Project no.	034084
Project acronym:	SELFMAN
Project title:	*Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components*

European Sixth Framework Programme Priority 2, Information Society Technologies

The Adventures of Selfman - Year Two

Due date of deliverable:	July 15, 2008
Actual submission date:	July 15, 2008
Start date of project:	June 1, 2006
Duration:	36 months
Dissemination level:	PU

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document presents all the deliverables of the second year of the SELFMAN project except the Periodic Activity Report which is submitted as a separate document. As an experiment, we have decided to bundle all the deliverables together, similar to a book. Each deliverable corresponds to a single chapter in this book, supplemented with the appendices (relevant papers) and bibliographic references. Note that each deliverable can be transformed into a separate document if necessary by extracting these pages with Adobe Acrobat. We decided to create a single book-length document for three reasons:

- Some of the relevant papers are part of more than one deliverable. Putting them in a single appendix removes this duplication.
- We have additional results that do not fit easily into a single deliverable, for example Raphaël Collet's Ph.D. dissertation and the release of Mozart 1.4.0 (see Appendix A.2) and the papers on feedback loop architectures (see Chapter 2 and Appendix A.1). These results are not listed in the Description of Work but they are logically part of the project. They would have been listed if we would have been prescient enough when we originally defined the project.
- The organization in book form shows the coherence of the project. The deliverables form an ascending ladder from low-level structures (the SONs, Workpackage 1), followed by the component models (Workpackage 2), the transaction protocol (Workpackage 3), the merge protocol and other self-* services (Workpackage 4), and finally the application scenarios (Workpackage 5). We have tried to cross-reference the deliverables to clarify the connections between these parts.

Chapter 2 starts the document by giving an overview of the SELFMAN project, summarizing the second-year results in a way that is strongly flavored by our ultimate vision (which is further explained in Appendix A.1).

Chapter 2

An overview of the SELFMAN project

This paper derives from a talk given originally in the Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2007), held in CWI, Amsterdam, from Oct. 24-26, 2007. The paper was written in March 2008 and will appear in the revised postproceedings. It gives an overview of the SELFMAN vision and a summary of the main results of the project in its second year. For a more far-ranging vision based on the concept of reversible phase transitions, we refer you to Appendix A.1. The network merge algorithm developed in SELFMAN is an example of a robust software system that shows reversible phase transitions.

Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project

Peter Van Roy[1], Seif Haridi[2], Alexander Reinefeld[3], Jean-Bernard Stefani[4], Roland Yap[5], and Thierry Coupaye[6]

[1] Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium
[2] Royal Institute of Technology (KTH), Stockholm, Sweden
[3] Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Berlin, Germany
[4] Institut National de Recherche en Informatique et Automatique (INRIA), Grenoble, France
[5] National University of Singapore (NUS)
[6] France Télécom Recherche et Développement, Grenoble, France

Abstract. As Internet applications become larger and more complex, the task of managing them becomes overwhelming. "Abnormal" events such as software updates, failures, attacks, and hotspots become frequent. The SELFMAN project will show how to handle these events automatically by making the application self managing. SELFMAN combines two technologies, namely structured overlay networks and advanced component models. Structured overlay networks (SONs) developed out of peer-to-peer systems and provide robustness, scalability, communication guarantees, and efficiency. Component models provide the framework to extend the self-managing properties of SONs over the whole application. SELFMAN is building a self-managing transactional storage and using it for three application demonstrators: a machine-to-machine messenging service, a distributed Wiki, and an on-demand media streaming service. This paper provides an introduction and motivation to the ideas underlying SELFMAN and a snapshot of its contributions midway through the project. We explain our methodology for building self-managing systems as networks of interacting feedback loops. We then summarize the work we have done to make SONs a practical basis for our architecture: using an advanced component model, handling network partitions, handling failure suspicions, and doing range queries with load balancing. Finally, we show the design of a self-managing transactional storage on a SON.

1 Introduction

It is now possible to build applications of a higher level of complexity than ever before, because the Internet has reached a higher level of reliability and scale than ever before using computing nodes that are more powerful than ever before. Applications that take advantage of this complexity cannot be managed directly by human beings; they are just too complicated. In order to build them, they

need to manage themselves. In that way, human beings only need to manage the high-level policies.

The SELFMAN project targets one part of this application space: applications built on top of structured overlay networks. Such networks are already self managing in the lower layers: they self organize around failures to provide reliable routing and lookup. We are building a service architecture on top of the overlay network using an advanced component model. To make it self managing, the service architecture is designed as a set of interacting feedback loops. Furthermore, by studying several application scenarios we find that support for distributed transactions is important. We are therefore building a replicated transactional storage as a key service on top of the structured overlay network. We will build three application demonstrators that use the service architecture and its transactional storage.

SELFMAN is a specific targeted research project (STREP) in the Information Society Technologies (IST) Strategic Objective 2.5.5 (Software and Services) of the European Sixth Framework Programme [30]. It started in June 2006 for a duration of three years with a budget of 1.96 MEuro. The project has seven partners: Université catholique de Louvain, Kungliga Tekniska Högskolan, INRIA (Grenoble), France Télécom Recherche et Développement (Grenoble), Konrad-Zuse-Zentrum für Informationstechnik (Berlin), National University of Singapore, and Stakk AB in Stockholm. This paper gives an overview of the motivations of SELFMAN, its approach, and its contributions midway through the project. The paper consists of the following six sections:

- Section 2: Motivation for self-managing systems. We give a brief history of system theory and cybernetics. We then explain why programs must be structured as systems of interacting feedback loops.
- Section 3: Presentation of the SELFMAN project. We present SELFMAN's decentralized service architecture and its three demonstrator applications.
- Section 4: Understanding and designing feedback structures. We explain some techniques for analyzing feedback structures and we give two realistic examples taken from human biology: the human respiratory system and the human endocrine system. We infer some design rules for feedback structures and present a tentative architecture and methodology for building them.
- Section 5: Introduction to structured overlay networks. We explain the basic ideas of SONs and the low-level self-management operations they provide. We then explain how they need to be extended for self-managing systems. We have extended them in three directions: to handle network partitions, failure suspicions, and range queries.
- Section 6: The transaction service. From our application scenarios, we have concluded that transactional storage is a key service for building self-managing applications. We are building the transaction service on top of a SON by using symmetric replication for the storage and a modified version of the Paxos nonblocking atomic commit.
- Section 7: Some conclusions. We recapitulate the progress that has been made midway through the project and summarize what remains to be done.

2 Motivation

2.1 Software complexity

Software is fragile. A single bit error can cause a catastrophe. Hardware and operating systems have been reliable enough in the part so that this has not unduly hampered the quantity of software written. Hardware is verified to a high degree. It is much more reliable than software. Good operating systems provide strong encapsulation at their cores (virtual memory, processes) and this has been polished over many years. New techniques in fault tolerance (e.g., distributed algorithms, Erlang) and in programming (e.g., structured programming, object-oriented programming, more recent methodologies) have arguably kept the pace so far. In fact we are in a situation similar to the Red Queen in Through the Looking-Glass: running as hard as we can to stay in the same place [7].

In our view, the next major increase in software complexity is now upon us. The Internet now has sufficient bandwidth and reliability to support large distributed applications. The number of devices connected to the Internet has increased exponentially since the early 1980s and this is continuing. The computing power of connected devices is continuously increasing. Many new applications are appearing: file sharing (Napster, Gnutella, Morpheus, Freenet, BitTorrent, etc.), information sharing (YouTube, Flickr, etc.), social networks (LinkedIn, FaceBook, etc.), collaborative tools (Wikis, Skype, various Messengers), MMORPGs (Massively Multiplayer On-line Role-Playing Games, such as World of Warcraft, Dungeons & Dragons, etc.), on-line vendors (Amazon, eBay, PriceMinister, etc.), research testbeds (SETI@home, PlanetLab, etc.), networked implementations of value-added chains (e.g., in the banking industry). These applications act like services. In particular, they are supposed to be long-lived. Their architectures are a mix of client/server and peer-to-peer. The architectures are still rather conservative: they do not take full advantage of the new possibilities.

The main problem that comes from the increase in complexity is that software errors cannot be eliminated [2, 38]. We have to cope with them. There are many other problems: scale (large numbers of independent nodes), partial failure (part of the system fails, the rest does not), security (multiple security domains) [18], resource management (resources tend to be localized), performance (harnessing multiple nodes or spreading load), and global behavior (emergent behavior of the system as a whole). Of these, global behavior is particularly relevant. Experiments show that large networks show behavior that is not easily predicted by the behaviors of the individual nodes (e.g., the power grid [11]).

2.2 Self-managing systems

What solution do we propose to these problems? For inspiration, we go back fifty years, to the first work on cybernetics and system theory: designing systems that regulate themselves [37, 4, 5]. A *system* is a set of components (called subsystems) that are connected together to form a coherent whole. Can we predict the system's behavior from its subsystems? Can we design a system with

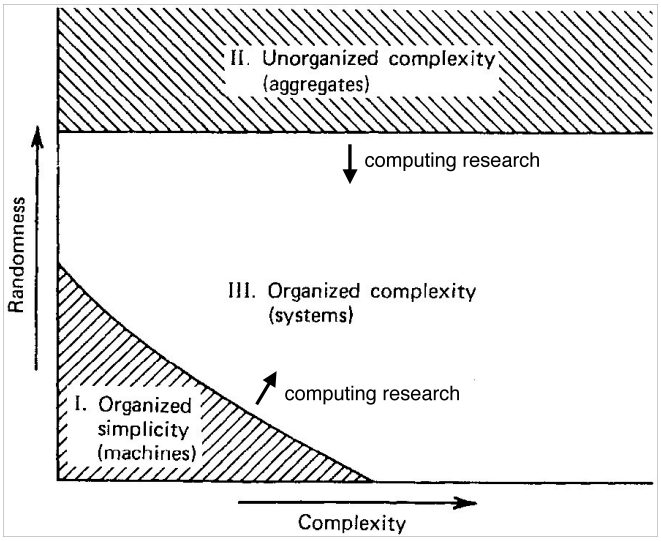


Fig. 1. Randomness versus complexity (taken from Weinberg [35])

desired behavior? These questions are particularly relevant for the distributed systems we are interested in. No general theory has emerged yet from this work. We do not intend to develop such a theory in SELFMAN. Our aim is narrower: to build self-managing *software* systems. Such systems have a chance of coping with the new complexity. Our work is complementary to [17], which applies control theory to design computing systems with feedback loops. We are interested in distributed systems with many interacting feedback loops.

Self management means that the system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies. In a sense, human intervention is *lifted* to the level of the policies. Typical self-management operations include adding/removing nodes, performance tuning, failure detection & recovery, intrusion detection & recovery, software rejuvenation. It is clear that self management exists at all levels of a system: the single node level, the network routing level, the service level, and the application level. For large-scale systems, environmental changes that require recovery by the system become normal and even frequent events. "Abnormal" events (such as failures) are normal occurrences.

Figure 1 (taken from [35]) classifies systems according to two axes: their complexity (the number of components and interactions) and the amount of randomness they contain (how unpredictable the system is). There are two shaded areas that are understood by modern science: machines (organized simplicity) and aggregates (unorganized complexity). The vast white area in the middle is poorly understood. We extend the original figure of [35] to emphasize that computing

research is the vanguard of system theory: it is pushing inwards the boundaries of the two shaded areas. Two subdisciplines of computing are particularly relevant: programming research (developing complex programs) and computational science (designing and simulating models). In SELFMAN we do both: we design algorithms and architectures and we simulate the resulting systems in realistic conditions.

2.3 Designing self-managing software systems

Designing self-managing systems means in large part to design systems with feedback loops. Real life is filled with variations on the feedback principle. For example:

- Bending a plastic ruler: a system with a single stable state. The ruler resists with a force that increases with the degree of bending, until equilibrium is reached (or until the ruler breaks: a change of phase). The ruler is a simple self-adaptive system with a single feedback loop.
- A clothes pin: a system with one stable and one unstable state. It can be kept temporarily in the unstable state by pinching. When the force is released, it will go back to (a possibly more complex) stable state.
- A safety pin: a system with two stable states, open and closed. Within each stable state the system is adaptive like the ruler. This is an example of a feedback loop with management (see Section 4): the outer control (usually a human being) chooses the stable state.

In general, anything that has continued existence is managed by a feedback loop. Lack of feedback means that there is a runaway reaction (an explosion or an implosion). This is true at all size scales, from the atomic to the astronomic. For example, binding of atoms to form a molecule is governed by a negative feedback loop: when perturbed it will return to equilibrium (or find another equilibrium). A star at the end of its lifetime collapses until it finds a new stable state. If there is no force to counteract the collapse, then the star collapses indefinitely (at least, until it goes beyond our current understanding of physics). If the star is too heavy to become a neutron star, then it becomes a black hole, which in our current understanding is a singularity.

Most products of human civilization need an implicit management feedback loop, called "maintenance", done by a human. For example, changing lightbulbs, replacing broken windows, or tanking a car. Each human mind is at the center of an enormous number of these feedback loops. The human brain has a large capacity for creating such loops; they are called "habits" or "chores". Most require very little conscious awareness. Repetition has caused them to be programmed into the brain below consciousness. However, if there are too many feedback loops to manage then the brain is overloaded: the human complains that "life is too complicated"! We can say that civilization advances by reducing the number of feedback loops that have to be explicitly managed [36]. A dishwashing machine reduces the work of washing dishes, but it needs to be bought, filled and emptied, maintained, replaced, etc. Is it worth it? Is the total effort reduced?

Software is in the same situation as other products of human civilization. In the current state, most software products are very fragile: they require frequent maintenance by a human. This is one of the purposes of SELFMAN: to reduce this need for maintenance by designing feedback loops into the software. This is a vast area of work; we have decided to restrict our efforts to large-scale distributed systems based on structured overlay networks. Because they have low-level self management built in, we consider them an ideal starting point. SONs have greatly matured since the first work in 2001 [33]; current SONs are (almost) ready to be used in real systems. We are adapting them in two directions for SELFMAN. First, we are extending the SON algorithms to handle important network issues that are not handled in the SON literature, such as network partitioning (see Section 5). Second, we are rebuilding the SON using a component model [1]. This is needed because the SON algorithms themselves have to be managed and updated while the SON is running, for example to add new basic functionality such as load balancing or new routing algorithms. The component model is also used for the other services we need for self management.

3 The SELFMAN project

The SELFMAN project is designing a decentralized service architecture and using it to build three demonstrator applications. Here we introduce the service architecture and the demonstrator applications. We also mention two important inspirations of SELFMAN: IBM's Autonomic Computing Initiative and the Chord system. Section 4.3 explains how the service architecture is used as a basis for self management.

3.1 Decentralized service architecture

SELFMAN is based on the premise that there is a synergy between structured overlay networks (SONs) and component models:

- SONs already provide low-level self-management abilities. We are reimplementing our SONs using a component model that adds lifecycle management and hooks for supporting services. This makes the SON into a substrate for building services.
- The component model is based on concurrent components and asynchronous message passing. It uses the communication and storage abilities of the SON to enable it to run in a distributed setting. Because the system may need to update and reorganize itself, the components need introspection and reconfiguration abilities. We have designed a process calculus, Oz/K, that has these abilities in a practical form [23].

This leads to a simple service architecture for decentralized systems: a SON lower layer providing robust communication and routing services, extended with other basic services and a transaction service. Applications are built on top of this

service architecture. The transaction service is important because many realistic application scenarios need it (see Section 3.2).

The structured overlay network is the base. It provides guaranteed connectivity and fast routing in the face of random failures (Section 5). It does not protect against malicious failures: in our current design we must consider the network nodes as trusted. We assume that untrusted clients may use the overlay as a basic service, but cannot modify its algorithms. See [42] for more on security for SONs and its effect on SELFMAN. We have designed and implemented robust SONs based on the DKS, Chord#, and Tango protocols [13, 29, 8]. These implementations use different styles and platforms, for example DKS is implemented in Java and uses locking algorithms for node join and leave. Tango is implemented in Oz and uses asynchronous algorithms for managing connectivity (Section 5.2). We have also designed an algorithm for handling network partitions and merges, which is an important failure mode for structured overlay networks (Section 5.1).

The transaction service uses a replicated storage service (Section 6). The transaction service is implemented with a modified version of the Paxos nonblocking atomic commit [15] and uses optimistic concurrency control. This algorithm is based on a majority of correct nodes and eventual leader detection (the so-called partially synchronous model). It should therefore cope with failures as they occur on the Internet.

This simple service architecture is our starting point for building self-managing applications. Section 4.3 shows how this service architecture is used to build the feedback structures that are needed for self management.

Application	Self-* Properties	Components	Overlays	Transactions
M2M Messaging	++	++	+	+
Distributed Wiki	++	+	++	++
P2P Media Streaming	++	+	++	
J2EE Application Server	++	++		+

Table 1. Requirements for selected self-managing applications

3.2 Demonstrator applications and guidelines

Using this self-management architecture, we will build three application demonstrators [12]:

– A machine-to-machine messaging application (specified by partner France Télécom). This is a decentralized messaging application. It must recover on node failure, gracefully degrade and self optimize, and have transactional behavior.

- A distributed Wiki application (specified by partner ZIB). This is a Wiki (a user-edited set of interlinked Web pages) that is distributed over a SON using transactions with versioning and replication, supporting both editing and search.
- An on-demand video streaming application (specified by partner Stakk). This application provides distributed live media streams with quality of service to large and dynamically varying numbers of customers. Dynamic reconfiguration is needed to handle the fluctuating structure.

Table 1 shows how much these applications need in four areas: self-* properties, components, overlay networks (decentralized execution), and transactions. Two pluses (++) mean strong need and one plus (+) means some need. An empty space means no need for that area according to our current understanding. All these applications have a strong need for self-management support. The table shows a fourth application that was initially considered, an application server written in J2EE, but we rejected it for SELFMAN because it does not have any requirements for decentralized execution.

At the end of the project we will provide a set of guidelines and general programming principles for building self-managing applications. One important principle is that these applications are built as a set of interacting feedback loops. A feedback loop, where part of the system is monitored and then used to influence the system, is an important basic element for a system that can adjust to its surroundings. As part of SELFMAN, we are carefully studying how to build applications with feedback loops and how feedback interacts with distribution.

3.3 Related work

The SELFMAN project is related to two important areas of work:

- IBM's Autonomic Computing Initiative [19]. This initiative started in 2001 and aims to reduce management costs by removing humans from low-level system management loops. The role of humans is then to manage policy and not to manage the mechanisms that implement it.
- Structured overlay network research. The most well-known SON is the Chord system, published in 2001 [33]. Other important early systems are Ocean Store and CAN. Inspired by popular peer-to-peer applications, these systems led to much active research in SONs, which provide low-level self management of routing, storage and smart lookup in large-scale distributed systems.

Other important related work is research in ambient and adaptive computing, and research in biophysics on how biological systems regulate and adapt themselves. For example, [21] shows how systems consisting of two coupled feedback loops behave in a biological setting.

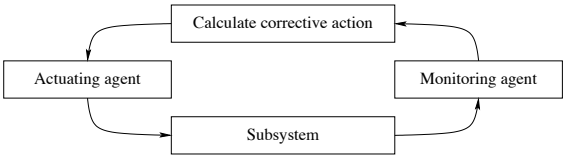


Fig. 2. A feedback loop

4 Understanding and designing feedback structures

A self-managing system consists of a large set of interacting feedback loops. We want to understand how to build systems that consist of many interacting feedback loops. Systems with one feedback loop are well understood, see, e.g., the book by Hellerstein *et al* [17], which shows how to design computing systems with feedback control, for example to maximize throughput in Apache HTTP servers, TCP communication, or multimedia streaming. The book focuses on regulating with single feedback loops. Systems with many feedback loops are quite different. To understand them, we start by doing explorations both in analysis and synthesis: we study existing systems (e.g., biological systems) and we design decentralized systems based on SONs.

A feedback loop consists of three parts that interact with a subsystem (see Figure 2): a monitoring agent, a correcting agent, and an actuating agent. The agents and the subsystem are concurrent components that interact by sending each other messages. We call them "agents" because they play specific roles in the feedback loop; an agent can of course have subcomponents. As explained in [34], feedback loops can interact in two ways:

- Stigmergy: two loops monitor and affect a common subsystem.
- Management: one loop directly controls another loop.

How can we design systems with many feedback loops that interact both through stigmergy and management? We want to understand the rules of good feedback design, in analogy to structured and object-oriented programming. Following these rules should give us good designs without having to laboriously analyze all possibilities. The rules can tell us what the global behavior is: whether the system converges or diverges, whether it oscillates or behaves chaotically, and what states it settles in. To find these rules, we start by studying existing feedback loop structures that work well, in both biological and software systems. We try to understand these systems by analysis and by simulation. Many feedback systems and feedback patterns have been investigated in the literature [34, 27, 22]. Sections 4.1 and 4.2 give two approaches to understanding existing systems and summarize some of the design rules we can infer from them. Finally, Section 4.3 gives a first tentative methodology for designing feedback structures.

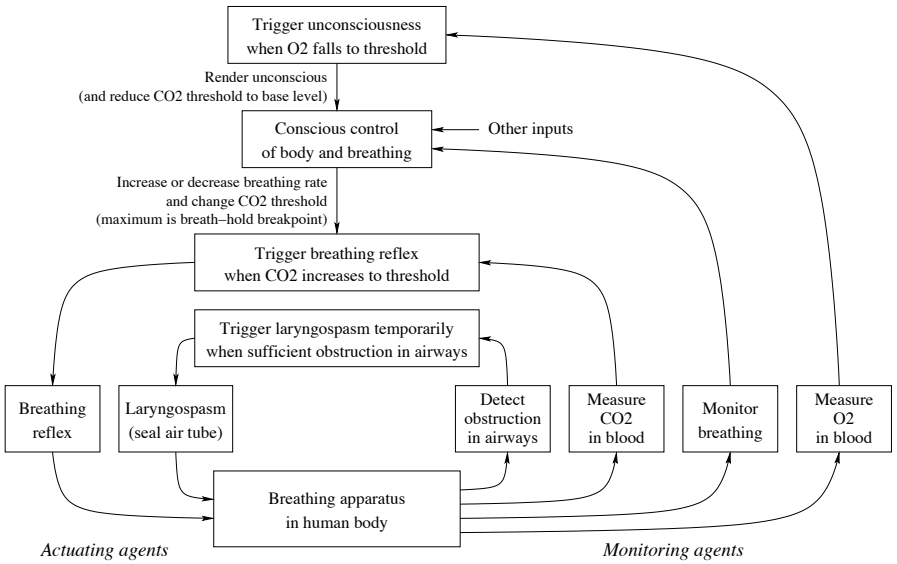


Fig. 3. The human respiratory system

4.1 Feedback structures in the human body

We investigate two feedback loop structures that exist in the human body: the human respiratory system and the human endocrine system. Figure 3 (taken from [34]) shows the human respiratory system, which has four feedback loops: three are arranged in a management hierarchy and the fourth interacts with them through stigmergy. This design works quite well. Laryngospasm can temporarily interfere with the breathing reflex, but after a few seconds it lets normal breathing take over. Conscious control can modulate the breathing reflex, but it cannot bypass it completely: in the worst case, the person falls unconscious and normal breathing takes over. We can already infer several design rules from this system: one loop managing another is an example of data abstraction, loops can avoid interference by working at different time scales, and since complex loops (such as conscious control) can have an unpredictable effect (they can be either stabilizing or unstabilizing) it is a good idea to have an outer "fail-safe" management loop. Conscious control is a powerful problem solver but it needs to be held in check.

The respiratory system is a simple example of a feedback loop structure that works; we now give a more complex biological example, namely the human endocrine system [10]. The endocrine system regulates many quantities in the human body. It uses chemical messengers called *hormones* which are secreted by specialized glands and which exercise their action at a distance, using the blood

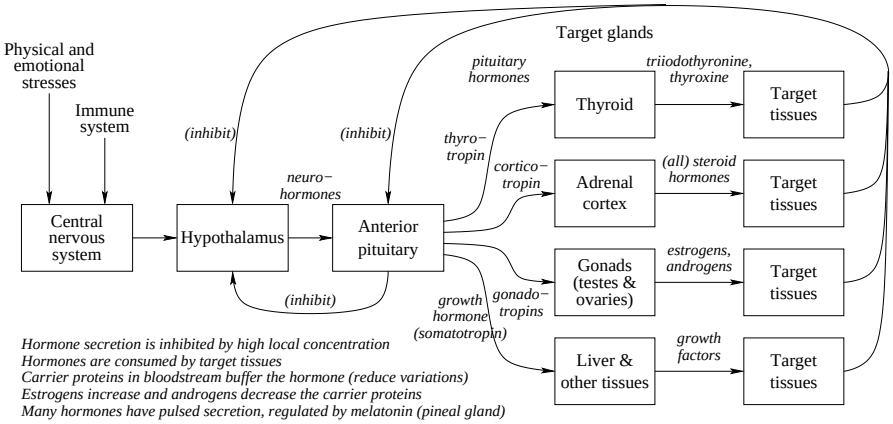


Fig. 4. The hypothalamus-pituitary-target organ axis

stream as a diffusion channel. By studying the endocrine system, we can obtain insights in how to build large-scale self-regulating distributed systems. There are many feedback loops and systems of interacting feedback loops in the endocrine system. It provides homeostasis (stability) and the ability to react properly to environmental stresses. Much of the regulation is done by simple negative feedback loops. For example, the glucose level in the blood stream is regulated by the hormones glucagon and insulin. In the pancreas, A cells secrete glucagon and B cells secrete insulin. An increase in blood glucose level causes a decrease in the glucagon concentration and an increase in the insulin concentration. These hormones act on the liver, which releases glucose in the blood. Another example is the calcium level in the blood, which is regulated by parathyroid hormone (parathormone) and calcitonine, also in opposite directions, both of which act on the bone. The pattern here is of two hormones that work in opposite directions (push-pull). This pattern is explained by [21] as a kind of dual negative feedback loop (an NN loop) that improves regulation.

More complex regulatory mechanisms also exist in the endocrine system, e.g., the hypothalamus-pituitary-target organ axis. Figure 4 shows its main parts as a feedback structure. This system consists of two superimposed groups of negative feedback loops (going through the target tissues and back to the hypothalamus and anterior pituitary), a third short negative loop (from the anterior pituitary to the hypothalamus), and a fourth loop from the central nervous system. The hypothalamus and anterior pituitary act as master controls for a large set of other regulatory loops. Furthermore, the nervous system affects these loops through the hypothalamus. This gives a time scale effect since the hormonal loops are slow and the nervous system is fast. Letting it affect the hormonal loop helps to react quickly to external events.

Figure 4 shows only the main components and their interactions; there are many more parts in the full system. There are more interacting loops, “short cir-

cuits", special cases, interaction with other systems (nervous, immune). Negative feedback is used for most loops, saturation (like in the Hill equations introduced in Section 4.2) for others. Realistic feedback structures can be complex. Evolution is not always a parsimonious designer! The only criterion is that the system has to work.

Computational architecture We can say something about the computational architecture of the human endocrine system. There are *components* and *communication channels*. Components can be both local (glands, organs, clumps of cells) or global (diffuse, over large parts of the body). Channels can be point-to-point or broadcast. Point-to-point channels are fast, e.g., nerve fibers from the spinal chord to the muscle tissue. Broadcast is slower, e.g., diffusion of a hormone through the blood circulation. Buffering is used to reduce variations, e.g., the carrier proteins in the bloodstream act as buffers by storing and releasing hormones. Regulatory mechanisms can be modeled by interactions between components and channels. Often there are intermediate links (like the carrier proteins). Abstraction (e.g., encapsulation) is almost always approximate. This is an important difference with digital computers. Biological and social abstractions tend to be leaky; computer abstractions tend not to be. This can have a large effect on the design. In biological systems security is done through a mechanism that is itself leaky, e.g., the human immune system. In computer systems, the security architecture tries to be as nonleaky as possible, although this cannot be perfect because of covert channels.

4.2 Analysis of feedback structures

How can we design a system with many interacting feedback loops like that of Figure 3? Mathematical analysis of interacting feedback loops is quite complex, especially if they have nonlinear behavior. Can we simplify the system to have linear or monotonic behavior? Even then, analysis is complex. For example, Kim *et al* [21] analyze biological systems consisting of just two feedback loops interacting through stigmergy. They admit that their analysis only has limited validity because the coupled feedback loops they analyze are parts of much larger sets of interacting feedback loops. Their analysis is based on Matlab simulations using the Hill equations, first-order nonlinear differential equations that model the time evolution and mutual interaction of molecular concentrations. The Hill equations model nonlinear monotonic interaction with saturation effects. We give a simple example using two molecular concentrations X and Y. The equations have the following form (taken from [21]):

$$\frac{dY}{dt} = \frac{V_X (X/K_{XY})^H}{1 + (X/K_{XY})^H} - K_{dY} Y + K_{bY}$$
$$\frac{dX}{dt} = \frac{V_Y}{1 + (Y/K_{YX})^H} - K_{dX} X + K_{bX}$$

Here we assume that X activates Y and that Y inhibits X. The equations model saturation (the concentration of a molecule has an upper limit) and ac-

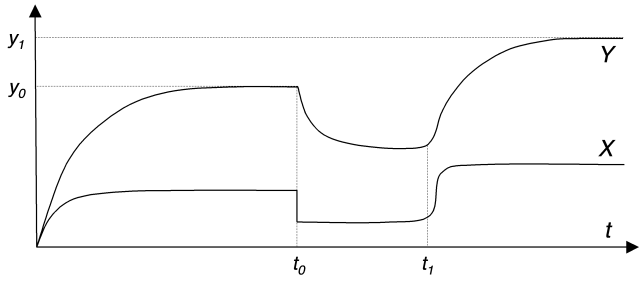


Fig. 5. Example of a biological system where X activates Y

tivation/inhibition with saturation (one molecule can affect another, up to a point). We see that X and Y, when left to their own, will each asymptotically approach a limit value with an exponentially decaying difference. Figure 5 shows a simplified system where X activates Y but Y does not affect X. X has a discrete step decrease at t_0 and a continuous step increase at t_1. Y follows these changes with a delay and eventually saturates. The constants K_{dY} and K_{bY} model saturation of Y (and similarly for X). The constants V_X, K_{XY}, and H model the activation effect of X on Y. We see that activation and inhibition have upper and lower limits.

By simulating these equations, Kim *et al* determine the effect of two coupled feedback loops, each of which can be positive or negative.

- A positive loop is bistable or multistable; it is commonly used in biological systems for decision making. Two coupled positive loops cause the decision to be less affected by environmental perturbations: this is useful for biological processes that are irreversible (such as mitosis, i.e., cell division).
- A negative loop reduces the effect of the environment; it is commonly used in biological systems for homeostasis, i.e., to keep the biological system in a stable state despite environmental changes. Negative loops can also show oscillation because of the time delay between the output and input. Two coupled negative loops can show stronger and more sustained oscillations than a single loop. They can implement biological oscillations such as circadian (daily) rhythms.
- A combined positive and negative loop can change its behavior depending on how it is activated, to become more like a positive or more like a negative loop. This is useful for regulation.

These results are interesting because they give insight into nonlinear monotonic interaction with saturation. They can be used to design structures with two coupled feedback loops.

Many patterns of feedback loops have been analyzed in this way. For example, [27] shows how to model oscillations in biological systems by cycles of feedback loops. The cycle consists of molecules where each molecule activates or inhibits the next molecule in the cycle. If the total effect of the cycle is a negative

feedback then the cycle can give oscillations. Given an oscillatory behavior, the topology of the cycle (the molecules involved and their interaction types) can be reconstructed. Many other patterns have been analyzed as well in biological systems, but there is as yet no general theory for analyzing these feedback structures. In SELFMAN we are interested in investigating the kinds of equations that apply to software. In software, the feedback structures may not follow the Hill equations. For example, they may not be monotonic. Nevertheless, the Hill equations are a useful starting point because they model saturation, which is an interesting form of nonlinearity.

4.3 Feedback structures for self management

From the examples given in the previous sections and elsewhere [34, 4, 37, 5, 35], we can give a tentative methodology for designing feedback structures. We assume that the overall architecture follows the decentralized structure given in Section 3.1: a set of loosely-coupled services built on top of a structured overlay network. We build the feedback structure within this framework. We envisage the following three layers for a self-managing system:

1. *Components and events.* This basic layer corresponds to the service architecture of Section 3.1: services based on concurrent components that interact through events [1, 9]. There can be publish/subscribe events, where any component that subscribes to a published type will receive the events. There is a failure detection service that is eventually perfect with suspect and resume events. There can be more sophisticated services, like the transaction service mentioned in Section 3.1 and presented in more detail in Section 6.
2. *Feedback loop support.* This layer supports building feedback loops. This is sufficient for cooperative systems. The two main services needed for feedback loops are a pseudoreliable broadcast (for actuating) and a monitoring layer. Pseudoreliable broadcast guarantees that nodes will receive the message if the originating node survives [13]. Monitoring detects both local and global properties. Global properties are calculated from local properties using a gossip algorithm [20] or using belief propagation [39]. The multicast and monitoring services are used to implement self management abilities.
3. *Multiple user support.* This layer supports users that compete for resources. This is a general problem that requires a general solution. If the users are independent, one possible approach is to use collective intelligence techniques (see Section 4.4). These techniques guarantee that when each user maximizes its private utility function, the global utility will also be maximized. This approach does not work for Sybil attacks (where one user appears as multiple users to the system). No general solution to Sybil attacks is known. A survey of partial solutions is given in [42]. We cite two solutions. One solution is to validate the identities of users using a trusted third party. Another solution is to use algorithms designed for a Byzantine failure model, which can handle multiple identical users up to some upper bound. Both solutions give significant performance penalties.

We now discuss two important issues that affect feedback structures: simple versus complex components (how much computation each component does) and time scales (different time scales can be independent). A complex component does nontrivial reasoning, but in most cases this reasoning is only valid in part of the system's state space and should be ignored in other parts. This affects the architecture of the system. At different time scales, a system can behave as separate systems. We can take advantage of this to improve the system's behavior.

Complex components A self-managing system consists of many different kinds of components. Some of these can be quite simple (e.g., a thermostat). Others can be quite complex (e.g., a human being or a chess program). We define a component as complex if it can do nontrivial reasoning. Some examples are a human user, a computer chess program, a compiler that translates a program text, a search engine over a large data set, and a problem solver based on SAT or constraint algorithms.

Whether or not a component is simple or complex can have a major effect on the design of the feedback structure. For example, a complex component may introduce instability that needs fail-safe protective mechanisms (see, e.g., the human respiratory system) or mechanisms to avoid "freeloaders" (see Section 4.4). Many systems have both simple and complex components. We have seen regulatory systems in the human body which may have some conscious control in addition to simpler components. Other systems, called social systems, have both human and software components. Many distributed applications (e.g., MMORPGs) are of this kind.

A complex component can radically affect the behavior of the system. If the component is cooperative, it can stabilize an otherwise unstable system. If the component is competitive, it can unstabilize an otherwise stable system. All four combinations of {simple,complex} × {cooperative,competitive} appear in practice. With respect to stability, there is no essential difference between human components and programmed complex components; both can introduce stability and instability. Human components excel in adaptability (dynamic creation of new feedback loops) and pattern matching (recognizing new situations as variations of old ones). They are poor whenever a large amount of precise calculation is needed. Programmed components can easily go beyond human intelligence in such areas. Whether or not a component can pass a Turing test is irrelevant for the purposes of self management.

How do we design a system that contains complex components? If the component is external to the designed system (e.g., human users connecting to a system) then we must design defensively to limit the effect of the component on the system's behavior. We need to protect the system from the users and the users from each other. For example, the techniques of collective intelligence can be used, as explained in Section 4.4. Getting this right is not just an algorithmic problem; it also requires social engineering, e.g., incentive mechanisms [28].

If the component is inside the system, then it can improve system behavior but fail-safe mechanisms must be built in to limit its effect. For example,

conscious control can improve the behavior of the human respiratory system, but it has a fail-safe to avoid instability (see Section 4.1). In general, a complex component will only enhance behavior in part of the system's state space. The system must make sure that the component cannot affect the system outside of this part.

Time scales Feedback loops that work at different time scales can often be considered to be completely independent of each other. That is, each loop is sensitive to a particular frequency range of system behavior and these ranges are often nonoverlapping. Wiener [37] gives an example of a human driver braking an automobile on a surface whose slipperiness is unknown. The human "tests" the surface by small and quick braking attempts; this allows to infer whether the surface is slippery or not. The human then uses this information to modify how to brake the car. This technique uses a loop at a short time scale to gain information about the environment, which is then used to help for a long time scale. The fast loop manages the slow loop.

4.4 Managing multiple users through collective intelligence

An important part of feedback structures that we have not yet explained is how to support users that compete for resources. A promising technique for this is collective intelligence [40, 41]. It can give good results when the users are independent (no Sybil attacks or collusion). The basic question is how to get selfish agents to work together for the common good. Let us define the problem more precisely. We have a system that is used by a set of agents. The system (called a "collective" in this context) has a global utility function that measures its overall performance. The agents are selfish: each has a private utility function that it tries to maximize. The system's designers define the reward (the increment in its private utility) given to each of the agent's actions. The agents choose their actions freely within the system. The goal is that agents acting to maximize their private utilities should also maximize the global utility. There is no other mechanism to force cooperation. This is in fact how society is organized. For example, employees act to maximize their salaries and work satisfaction and this should benefit the company.

A well-known example of collective intelligence is the El Farol bar problem [3], which we briefly summarize. People go to El Farol once a week to have fun. Each person picks which night to attend the bar. If the bar is too crowded or too empty it is no fun. Otherwise, they have fun (receive a reward). Each person makes one decision per week. All they know is last week's attendance. In the idealized problem, people don't interact to make their decision, i.e., it is a case of pure stigmergy! What strategy should each person use to maximize his/her fun? We want to avoid a "Tragedy of the Commons" situation where maximizing private utilities causes a minimization of the global utility [16].

We give the solution according to the theory of collective intelligence. Assume we define the global utility G as follows:

$$G = \sum_{w} W(w)$$
$$W(w) = \sum_{d} \phi_d(a_d)$$

This sums the week utility $W(w)$ over all weeks w. The week utility $W(w)$ is the sum of the day utilities $\phi_d(a_d)$ for each weekday d where the attendance a_d is the total number of people attending the bar that day. The system designer picks the function $\phi_d(y) = \alpha_d y e^{-y/c}$. This function is small when y is too low or too high and has a maximum in between. Now that we know the global utility, we need to determine the agents' reward function. This is what the agent receives from the system for its choice of weekday. We assume that each agent will try to maximize its reward. For example, [40] assumes that each agent uses a learning algorithm where it picks a night randomly according to a Boltzmann distribution distributed according to the energies in a 7-vector. When it gets its reward, it updates the 7-vector accordingly. Real agents may use other algorithms; this one was picked to make it possible to simulate the problem.

How do we design the agent's reward function $R(w)$, i.e., the reward that the agent is given each week? There are many bad reward functions. For example, Uniform Division divides $\phi_d(y)$ uniformly among all a_y agents present on day y. This one is particularly bad: it causes the global utility to be minimized. One reward that works surprisingly well is called Wonderful Life:

$$R_{WL}(w) = W(w) - W_{\text{agent absent}}(w)$$

$W_{\text{agent absent}}(w)$ is calculated in the same way as $W(w)$ but when the agent is missing (dropped from the attendance vector). We can say that $R_{WL}(w)$ is the difference that the agent's existence makes, hence the name Wonderful Life taken from the title of the Frank Capra movie [6]. We can show that if each agent maximizes its reward $R_{WL}(w)$, the global utility will also be maximized. Let us see how we can use this idea for building collective services. We assume that agents try to maximize their rewards. For each action performed by an agent, the system calculates the reward. The system is built using security techniques such as encrypted communication so that the agent cannot "hack" its reward.

This approach does not solve all the security problems in a collaborative system. For example, it does not solve the collusion problem when many agents get together to try to break the system. For collusion, one solution is to have a monitor that detects suspicious behavior and ejects colluding users from the system. This monitor is analogous to the SEC (Securities and Exchange Commission) which regulates and polices financial markets in the United States. Collective intelligence can still be useful as a base mechanism. In many cases, the default behavior is that the agents cannot or will not talk to each other, since they do not know each other or are competing. Collective intelligence is one way to get them to cooperate.

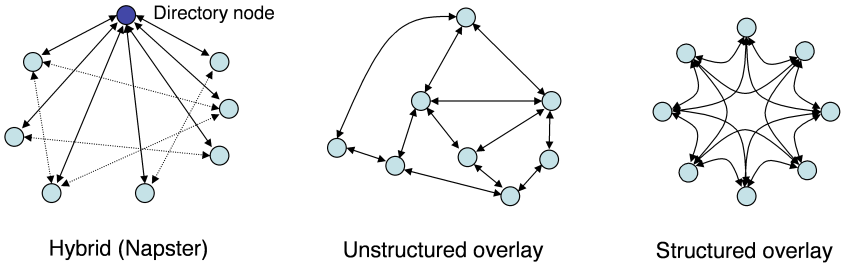


Fig. 6. Three generations of peer-to-peer networks

5 Structured overlay networks

Structured overlay networks are a recent development of peer-to-peer networks. In a peer-to-peer network, all nodes play equal roles. There are no specialized client or server nodes. There have been three generations of peer-to-peer networks, which are illustrated in Figure 6:

- The first generation is a hybrid: all client nodes are equal but there is a centralized node that holds a directory. This is the structure used by the Napster file-sharing system.
- The second generation is an unstructured overlay network. It is completely decentralized: each node knows a few neighbor nodes. This structure is used by systems such as Gnutella, Kazaa, Morpheus, and Freenet. Lookup is done by flooding: a node asks its neighbor, which asks its neighbors, up to a fixed depth. There are no guarantees that the lookup will be successful (the item may be just beyond the horizon) and flooding is highly wasteful of network resources. Recent versions of this structure use a hierarchy with two kinds of peer nodes: normal nodes and super nodes. Super nodes have higher bandwidth and reliability than normal nodes. This alleviates somewhat the disadvantages.
- The third generation is the structured overlay network. A well-known early example of this generation is Chord [33]. The nodes are organized in a structured way called an exponential network. Lookup can be done in logarithmic time and will guarantee to find the item if it exists. If nodes fail or new nodes join, then the network reorganizes itself to maintain the structure. Since 2001, many variations of structured overlay networks with different advantages and disadvantages have been designed: Chord, Pastry, Tapestry, CAN, P-Grid, Viceroy, DKS, Chord#, Tango, etc. In SELFMAN we build on our previous experience in DKS, Chord#, and Tango.

Structured overlay networks provide two basic services: name-based communication (point-to-point and group) and distributed hash table (also known as DHT, which provides efficient storage and retrieval of (key,value) pairs). Routing is

done by a simple greedy algorithm that reduces the distance of a message between the current node and the destination node. Correct routing means that the distance converges to zero in finite time.

Almost all current structured overlay networks are organized in two levels, a ring complemented by a set of fingers:

- *Ring structure.* All nodes are connected in a simple ring. The ring must always be connected despite node joins, leaves, and failures.
- *Finger tables.* For efficient routing, extra routing links called fingers are added to the ring. They are usually exponential, e.g., for the fingers of one node, each finger jumps twice as far as the previous finger. The fingers can temporarily be in an inconsistent state; this has an effect only on efficiency, not on correctness. Within each node, the finger table is continuously converging to a correct content.

Ring maintenance is a crucial part of the SON. Peer nodes can join and leave at any time. Peers that crash are like peers that leave but without notification. Temporarily broken links create false suspicions of failure.

We give three examples of structured overlay network algorithms developed in SELFMAN that are needed for important aspects of ring maintenance: handling network partitioning (Section 5.1), handling failure suspicions (Section 5.2), and handling range queries with load balancing (Section 5.3). These algorithms can be seen as dynamic feedback structures: they converge toward correct or optimal structures. The network partitioning algorithm restores a single ring in the case when the ring is split into several rings due to network partitioning. The failure handling algorithm restores a single ring in the case of failure suspicion of individual nodes. The range query algorithm handles multidimensional range queries. It has one ring per dimension. When nodes join or leave, each of these rings is adjusted (by splitting or joining pieces in the key space) to maintain balanced routing.

5.1 Handling network partitioning: the ring merge algorithm

Network partitioning is a real problem for any long-lived application on the Internet. A single router crash can cause part of the network to become isolated from another part. SONs should behave reasonably when a network partition arrives. If no special actions are taken, what actually happens when a partition arrives is that the SON splits into several rings. What we need to do is efficiently detect when such a split happens and efficiently *merge* the rings back into a single ring [31].

The merging algorithm consists of two parts. The first part detects when the merge is needed. When a node detects that another node has failed, it puts the node in a local data structure called the passive list. It periodically pings nodes in its passive list to see whether they are in fact alive. If so, it triggers the ring unification algorithm. This algorithm can merge rings in $O(n)$ time for network size n. We also define an improved gossip-based algorithm that can merge the network in $O(\log n)$ average time.

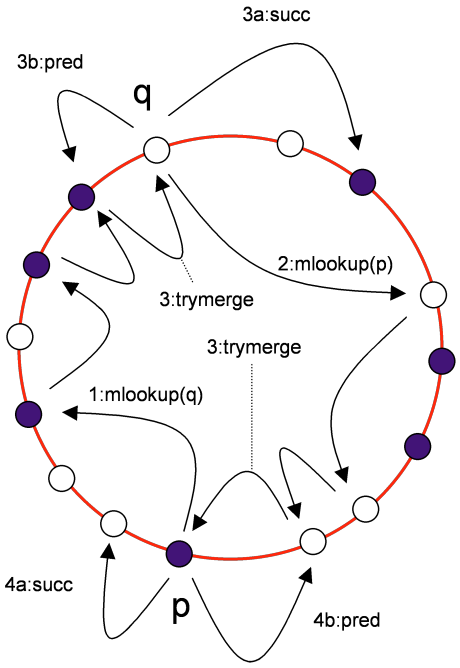


Fig. 7. The ring merge algorithm

Ring unification happens between pairs of nodes that may be on different rings. The unification algorithm assumes that all nodes live in the same identifier space, even if they are on different rings. Suppose that node p detects that node q on its passive list is alive. Figure 7 shows an example where we are merging the black ring (containing node p) and the white ring (containing node q). Then p does a modified lookup operation (`mlookup`(q)) to q. This lookup tries to reduce the distance to q. When it has reduced this distance as much as possible, then the algorithm attempts to insert q at that position in the ring using a second operation, `trymerge`(*pred*,*succ*), where *pred* and *succ* are the predecessor and successor nodes between which q should be inserted. The actual algorithm has several refinements to improve speed and to ensure termination.

5.2 Handling failure suspicions: the relaxed ring algorithm

A typical Internet failure mode is that a node suspects another node of failing. This suspicion may be true or false. In both cases, the ring structure must be maintained. This can be handled through the relaxed ring algorithm [24]. This algorithm maintains the invariant that every peer is in the same ring as its successor. Furthermore, a peer can never indicate another peer as the responsible node for data storage: a peer knows only its own responsibility. If a successor

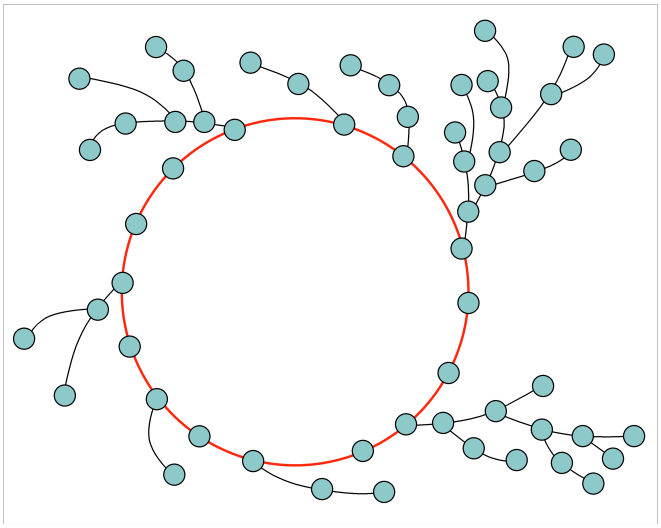


Fig. 8. The relaxed ring structure

node is suspected of having failed, then it is ejected from the ring. However, the node may still be alive and point to a successor. This leads to a structure we call the *relaxed ring*, which looks like a ring with "bushes" sticking out (see Figure 8). The bushes appear only if there are failure suspicions. At all times there is a perfectly connected ring at the core of the relaxed ring. The relaxed ring is always converging toward a perfect ring. The number of nodes in the bushes existing at any time depends on the churn (the rate of change of the ring, the number of failures and joins per time).

5.3 Handling multidimensional range queries with load balancing

Efficient data lookup is at the heart of peer-to-peer computing. Many SONs, including DKS and Tango, use consistent hashing to store (key,value) pairs in a distributed hash table (DHT). The hashing distributes the keys uniformly over the key space. Unfortunately, this scheme is unable to handle queries with partial information (such as wildcards and ranges) because adjacent keys are spread over all nodes. In this section, we argue that using DHTs is not a good idea in SONs. We support this argument by showing how to build a practical SON that stores the keys in lexicographic order. We have developed a first protocol, Chord#, and a generalization for multidimensional range queries, SONAR [29].

In SONAR the overlay has the shape of a multidimensional torus, where each node is responsible for a contiguous part of the data space. A uniform distribution of keys on the data space is not necessary, because denser areas get assigned more nodes. To support logarithmic routing, SONAR maintains, per dimension, fingers to other nodes that span an exponentially increasing number

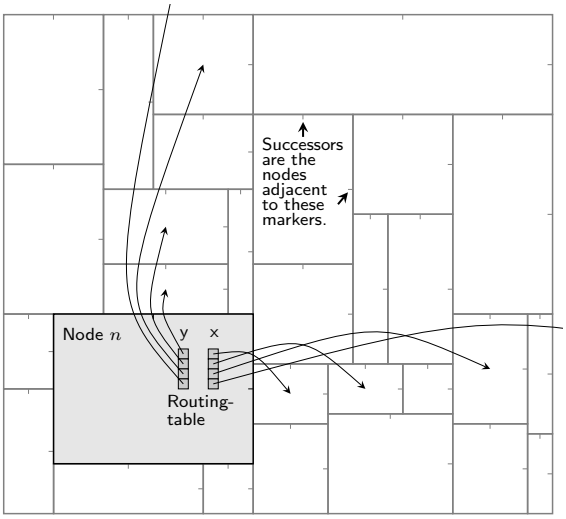


Fig. 9. Two-dimensional routing tables in SONAR

of nodes. Figure 9 shows an example in two dimensions. Most other overlays maintain such fingers in the key space instead and therefore require a uniform data distribution (e.g., which is obtained using hashing). SONAR, in contrast, avoids hashing and is therefore able to perform range queries of arbitrary shape in a logarithmic number of routing steps, independent of the number of system- and query-dimensions.

6 Transactions over structured overlay networks

For our three decentralized application scenarios, we need a decentralized transactional storage. We need transactions because the applications need concurrent access to shared data. We have therefore designed a transaction algorithm over SONs. We are currently simulating it to validate its assumptions and measure its performance [25, 26]. Implementing transactions over a SON is challenging because of churn (rate of node leaves, joins, and crashes and subsequent reorganizations of the SON) and because of the Internet's failure model (crash stop with imperfect failure detection).

The transaction algorithm is built on top of a reliable storage service. We implement this using replication. There are many approaches to replication on a SON. For example, we could use file-level replication (symmetric replication) or block-level replication using erasure codes. These approaches all have their own application areas. Our algorithm uses symmetric replication [14].

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of the Paxos algorithm [15]. In a companion paper, we have shown that majority techniques work well

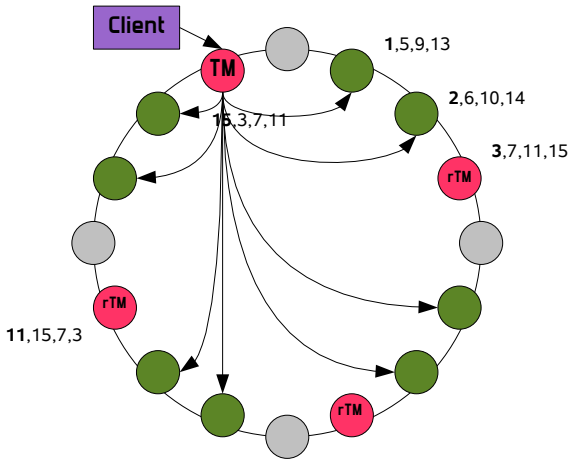


Fig. 10. Transaction with replicated manager and participants

for DHTs [32]: the probability of data consistency violation is negligible. If a consistency violation does occur, then this is because of a network partition and we can use the network merge algorithm of Section 5.1.

A client initiates a transaction by asking its nearest node, which becomes a transaction manager. Other nodes that store data are participants in the transaction. Assuming symmetric replication with degree f, we have f transaction managers and each other node participating gives f replicated participants. Figure 10 shows a situation with $f = 4$ and two nodes participating in addition to the transaction manager. Each transaction manager sends a Prepare message to all replicated participants, which each sends back a Prepared or Abort message to all replicated transaction managers. Each replicated transaction manager collects votes from a majority of participants and locally decides on abort or commit. It sends this to the transaction manager. After having collected a majority, the transaction manager sents its decision to all participants. This algorithm has six communication rounds. It succeeds if more than $f/2$ nodes of each replica group are alive.

7 Conclusions and future work

The SELFMAN project is using self-management techniques to build large-scale distributed systems. This paper gives a snapshot of the SELFMAN project at its halfway point. We explain why self management is important for software design and we give some first results on how to design self-managing systems as feedback loop structures. We show how to use structured overlay networks (SONs) as the basis of large-scale distributed self-managing systems. We explain how we have adapted SONs for our purposes by handling network partitioning, failure suspicions, and range queries with load balancing, and by providing a

transactional store service running over the SON. We present three realistic application scenarios, a machine-to-machine messaging application, a distributed Wiki, and an on-demand video streaming application. In the rest of the project, we will complete the transactional store and build the demonstrator applications. The final result will be a set of guidelines on how to build decentralized self-managing applications.

Acknowledgements

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). Peter Van Roy is the coordinator of the SELFMAN project. He acknowledges all the partners in the SELFMAN project for their insights and research results, some of which are summarized in this paper. He also acknowledges Mahmoud Rafea for encouraging him to look at the human endocrine system and Mohamed El-Beltagy for introducing him to collective intelligence.

References

1. Arad, Cosmin, Roberto Roverso, Seif Haridi, Yves Jaradin, Boris Mejias, Peter Van Roy, Thierry Coupaye, B. Dillenseger, A. Diaconescu, A. Harbaoui, N. Jayaprakash, M. Kessis, A. Lefebvre, and M. Leger. *Report on architectural framework specification*, SELFMAN Deliverable D2.2a, June 2007, `www.ist-selfman.org`.
2. Armstrong, Joe. "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, Nov. 2003.
3. Arthur, W. B. *Complexity in economic theory: Inductive reasoning and bounded rationality.* The American Economic Review, 84(2), May 1994, pages 406-411.
4. Ashby, W. Ross. "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): `pcp.vub.ac.be/books/IntroCyb.pdf`.
5. von Bertalanffy, Ludwig. "General System Theory: Foundations, Development, Applications," George Braziller, 1969.
6. Capra, Frank. "It's a Wonderful Life," Liberty Films, 1946.
7. Carroll, Lewis. "Through the Looking-Glass and What Alice Found There," 1872 (Dover Publications reprint 1999).
8. Carton, Bruno, and Valentin Mesaros. *Improving the Scalability of Logarithmic-Degree DHT-Based Peer-to-Peer Networks*, 10th International Euro-Par Conference, Aug. 2004, pages 1060–1067.
9. Collet, Raphaël, Michael Lienhardt, Alan Schmitt, Jean-Bernard Stefani, and Peter Van Roy. *Report on formal operational semantics (components and reflection)*, SELFMAN Deliverable D2.3a, Nov. 2007, `www.ist-selfman.org`.
10. Encyclopaedia Britannica. Article *Human Endocrine System*, 2005.
11. Fairley, Peter. *The Unruly Power Grid*, IEEE Spectrum Online, Oct. 2005.
12. France Télécom, Zuse Institut Berlin, and Stakk AB. *User requirements*, SELFMAN Deliverable D5.1, Nov. 2007, `www.ist-selfman.org`.
13. Ghodsi, Ali. "Distributed K-ary System: Algorithms for Distributed Hash Tables," Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, Oct. 2006.

14. Ghodsi, Ali, Luc Onana Alima, and Seif Haridi. *Symmetric Replication for Structured Peer-to-Peer Systems*, Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Springer-Verlag LNCS volume 4125, pages 74-85.
15. Gray, Jim and Leslie Lamport. *Consensus on transaction commit.* ACM Trans. Database Syst., ACM Press, 2006(31), pages 133-160.
16. Hardin, Garrett. *The Tragedy of the Commons*, Science, Vol. 162, No. 3859, Dec. 13, 1968, pages 1243–1248.
17. Hellerstein, Joseph L., Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. "Feedback Control of Computing Systems," Aug. 2004, Wiley-IEEE Press.
18. Hoglund, Greg and Gary McGraw. "Exploiting Online Games: Cheating Massively Distributed Systems," Addison-Wesley Software Security Series, 2008.
19. IBM. *Autonomic computing: IBM's perspective on the state of information technology*, 2001, `researchweb.watson.ibm.com/autonomic`.
20. Jelasity, Márk, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations*, Springer LNCS volums 3231, 2004, pages 79–98.
21. Kim, Jeong-Rae, Yeoin Yoon, and Kwang-Hyun Cho. *Coupled Feedback Loops Form Dynamic Motifs of Cellular Networks*, Biophysical Journal, 94, Jan. 2008, pages 359-365.
22. Kobayashi, Tetsuya, Luonan Chen, and Kazuyuki Aihara. *Modeling Genetic Switches with Positive Feedback Loops*, J. theor. Biol., 221, 2003, pages 379-399.
23. Lienhard, Michael, Alan Schmitt, and Jean-Bernard Stefani. *Oz/K: A Kernel Language for Component-Based Open Programming*, Sixth International Conference on Generative Programming and Component Engineering (GPCE'07), Oct. 2007.
24. Mejias, Boris, and Peter Van Roy. *A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks*, XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Nov. 2007.
25. Moser, Monika, and Seif Haridi. *Atomic Commitment in Transactional DHTs*, Proc. of the CoreGRID Symposium, Rennes, France, Aug. 2007.
26. Moser, Monika, Seif Haridi, Thorsten Schütt, Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. *First report on formal models for transactions over structured overlay networks*, SELFMAN Deliverable D3.1a, June 2007, `www.ist-selfman.org`.
27. Pigolotti, Simone, Sandeep Krishna, and Mogens H. Jensen. *Oscillation patterns in negative feedback loops*, Proc. National Academy of Sciences, vol. 104, no. 16, April 2007.
28. Salen, Katie, and Eric Zimmerman. "Rules of Play: Game Design Fundamentals," MIT Press, Oct. 2003.
29. Schütt, Thorsten, Florian Schintke, and Alexander Reinefeld. *Range Queries on Structured Overlay Networks*, Computer Communications 31(2008), pages 280-291.
30. SELFMAN: Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components, European Commission 6th Framework Programme, June 2006, `www.ist-selfman.org`.
31. Shafaat, Tallat M., Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*, Journal of Peer-to-Peer Networking and Applications, Springer-Verlag, 2008 (to appear).
32. Shafaat, Tallat M., Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks*, CoreGRID Integration Workshop, Heraklion, Greece, Springer LNCS, 2008.

33. Stoica, Ion, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, SIGCOMM 2001, pages 149-160.
34. Van Roy, Peter. *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS 2006), ENTCS volume 182, June 2007, pages 201-217.
35. Weinberg, Gerald M. "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).
36. Whitehead, Alfred North. Quote: *Civilization advances by extending the number of important operations which we can perform without thinking of them.*
37. Wiener, Norbert. "Cybernetics, or Control and Communication in the Animal and the Machine," MIT Press, Cambridge, MA, 1948.
38. Wiger, Ulf. *Four-Fold Increase in Productivity and Quality – Industrial-Strength Functional Programming in Telecom-Class Products*, Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems, 2001.
39. Wikipedia, the free encyclopedia. Article *Belief Propagation*, March 2008, `en.wikipedia.org/wiki/Belief_propagation`.
40. Wolpert, David H., Kevin R. Wheeler, and Kagan Tumer. *General principles of learning-based multi-agent systems*, Proc. Third Annual Conference on Autonomous Agents (AGENTS '99), May 1999, pages 77-93.
41. Wolpert, David H., Kevin R. Wheeler, and Kagan Tumer. *Collective intelligence for control of distributed dynamical systems*, Europhys. Lett., 2000.
42. Yap, Roland, Felix Halim, and Wu Yongzheng. *First report on security in structured overlay networks*, SELFMAN Deliverable D1.3a, Nov. 2007, `www.ist-selfman.org`.

Chapter 3

D1.2: Report on high-level self-management primitives for structured overlay networks

3.1 Executive summary

In order to build self-managing large-scale distributed systems, SELFMAN is aiming for a combination of component models and structured overlay networks. The goal is to achieve self management along four axes: self-configuration, self-healing, self-tuning and self-protection. This deliverable is mainly focused on self-configuration and self-healing properties of structured overlay network, defining the high-level primitives to be used by application running on top of the peer-to-peer infrastructure.

Structured overlay networks are used for building self-organising peer-to-peer systems with efficient routing. There are many ways of structuring these kind of networks. In SELFMAN, we focus our work on the ring topology, the relaxed-ring, and range queries. The ring and the relaxed-ring distribute the resources uniformly amount peers providing a Distribute Hash Table (DHT). If a uniform distribution of the resources is not feasible, it is recommended to use a topology like range queries. Results presented on this deliverable are related to these three kind of networks, and how applications interface them by using high-level primitives.

This deliverable is the continuation of the "Report on low-level self-management primitives for structured overlay networks" (D1.1), from year one. Results presented here achieved what was proposed as future work in Deliverable D1.1, and it presents the Application Programming Interface (API) that has been used to implement software deliverables D1.4 and D1.5,

presented in Chapters 5 and 6. It is also related with the results presented in Deliverable D3.3a, Chapter 14, where a replicated storage service is implemented on top of the results of this deliverable.

3.2 Contractors contributing to the Deliverable

Contributions measured on publications related to this deliverable are provided by Université catholique de Louvain UCL(P1), Zuse Institute Berlin ZIB(P5), Kungliga Tekniska Högskolan KTH(P2)and Peerialism. Contractors National University of Singapore NUS(P7) and Institut National de Recherche en Informatique et Automatique INRIA(P3), has contributed with analysis about how to improve the current results. Their related work in form of publications is presented in other deliverables. Partner Peerialism has contributed with a novel NAT traversal approach.

UCL(P1) has continued to work on the relaxed-ring in order to provide a stable self-managing peer-to-peer network to be used for the implementation of decentralized applications. It provides high-level primitives by defining the Application Programming Interface (API) to be used by other deliverables and work packages. This work is directly related to D1.5.

ZIB(P5) has continued its work on range queries, which provides an alternative solution to the ring topology. It has also developing work on sloppy management of structure peer-to-peer services. The work concerning the API is presented in Deliverable D3.3a.

KTH(P2) has also contributed defining the API by continuing the development of DKS. The work done on this deliverable is directly related to Deliverable D1.4.

Peerialism has contributed with a novel NAT traversal approach. This work might belong to low-level primitives, but since Peerialism has joined the project only since year two, we have also included this results because of their strong relationship with the rest of the project.

3.3 Results

This section is dedicated to report on the results of year two in the design of self-management primitives for structured overlay networks. Part of the results corresponds to concluding work on low-level primitives, and the other is the definition of an Application Programming Interface (API) as high-level primitives for developing services and applications on top of the structured overlay networks we have designed.

3.3.1 Introduction

Structured overlay networks are the basement for the development of self-managing large scale applications. They provide a self-configurable and self-healing network which is decentralized, avoiding the classical problem of single point of failure. SELFMAN has been working on different approaches to build such networks, targeting different and complementary issues. The work has been divided into low-level and high-level self-management primitives. The former has been reported on the first year of the project. The later are presented on this deliverable.

Since this is the continuation of the work reported in Deliverable D1.1, we analyze the future work of that deliverable. It was said that we will do other publications extending the results of multi-dimensional range queries in SONAR and the Relaxed-Ring. We have satisfactory achieved this task. The publications are described in Section 3.5, and the main concepts and ideas of these results are presented in the following sections of this deliverable.

We have also promised that KTH would continue with the development of DKS and UCL with P2PS, which is based on the Relaxed-Ring. These projects are presented as software deliverables D1.4 and D1.5, in chapters 5 and 6 respectively. The relation with this deliverable is based on the Application Programming Interface (API) we present here as high-level primitives. Both software deliverables implements big part of this API. Of course, all the results on the relaxed-ring are included in the development of P2PS.

In the following sections we presents the main concepts of the results achieved by this deliverable, and we also discuss ongoing and future work, and how it would be included in the next year of the project. We conclude by summarizing the publications which are added as appendices.

3.3.2 Range Queries

The results presented in this section targets the efficient handling of multiple range queries, which is actually not well achieved by our other results based

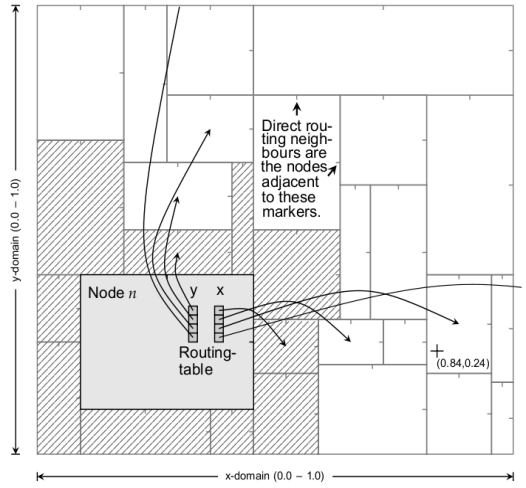


Figure 3.1: Routing fingers of a SONAR node in a two-dimensional data space.

on distributed hash-tables ring topology, as in P2PS and DKS. This is why this work is complementary to the other results presented on this deliverable.

We briefly summarize the results obtained by two structured overlay networks that support arbitrary range queries. The first one, named Chord$^{\#}$, has been derived from Chord [136] by substituting Chords hashing function by a key-order preserving function. It has a logarithmic routing performance and it supports range queries, which is not possible with Chord. Its $O(1)$ pointer update algorithm can be applied to any peer-to-peer routing protocol with exponentially increasing pointers.

Chord$^{\#}$ is extended to support multiple dimensions, resulting in SONAR, a Structured Overlay Network with Arbitrary Range queries. SONAR covers multi-dimensional data spaces and, in contrast to other approaches, SONARs range queries are not restricted to rectangular shapes but may have arbitrary shapes. Empirical results with a data set of two million objects show the logarithmic routing performance in a geospatial domain.

Figure 3.1 depicts a routing table in a two-dimensional data space. The keys are specified by attibute vectors (x, y) and hypercuboids cover the complete key space. The hypercuboids are presented in the figure as rectangular boxes which are managed by the nodes. Their different area is due to the key distribution, which confirms that this data would not be balanced in a ring architecture. In SONAR, at runtime, the load balancing scheme ensures that box holds about the same number of keys.

SONAR maintains two data structures, a neighbor list and a routing

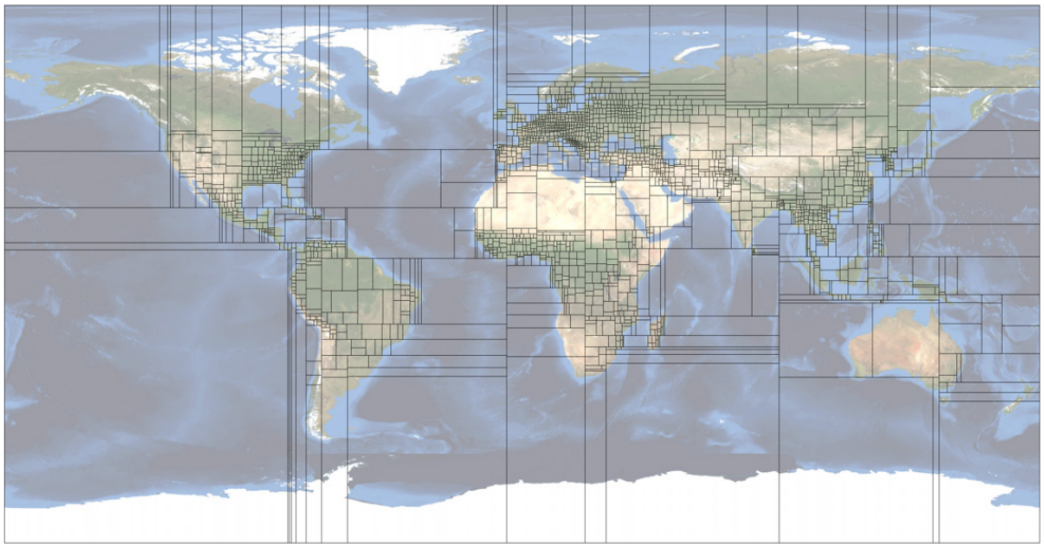


Figure 3.2: SONAR overlay network with 1.9 million keys (city coordinates) over 2048 nodes. Each rectangle represents one node.

table. The neighbor list contains links to all neighbors of a node. The node depicted by the grey box in Figure 3.1, for example, has ten neighbors. The routing table comprises d subtables, one for each dimension. Each subtable s with $1 \leq s \leq d$ contains fingers that point to remote nodes in exponentially increasing distances.

One application of SONAR is shown in Figure 3.2, which is also use as evaluation experiment. It represent a data set of a traveling salesman problem with the 1,904,711 largest cities worldwide. Their GPS locations follow a Zipf distribution, which is a common distribution pattern of many other application domains. In a preprocessing step we partitioned the globe into non-overlapping rectangular patches so that each patch contains about the same amount of cities. During the first year of the project we have achieved to manage the data with a netwok of 256 peers. The results of this year were able to scale to 2048 nodes, and still providing a very efficient routing. More specific results concerning this work can be found in Appendix A.3.

3.3.3 Relaxed-Ring

Continuing the work of D1.1, we have finished the design, analysis and validation of the Relaxed-Ring, the network topology used to implement P2PS. We have also define its high-level primitives which are included in the API

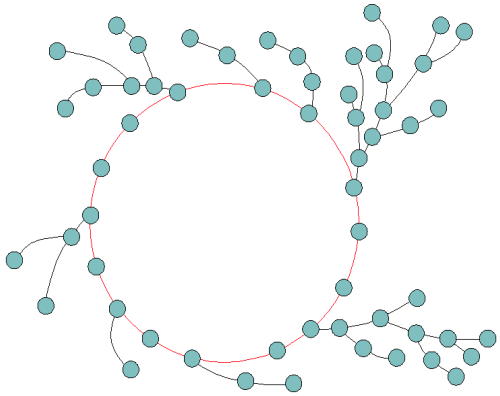


Figure 3.3: Extreme case of a relaxed-ring with many branches.

of Section 3.3.4. This section is dedicated to described the work done with respect to the analysis of the algorithm and its validation. The analysis has been done with the help of results coming from WP2, more precisely the definition of feedback loops in the design of self-managing software. The validation is done via simulations that will be described later on this section.

The Relaxed-Ring topology is a Chord-like [136] ring, where every peer has a successor (*succ*) and a predecessor (*pred*). It is a structured overlay network providing a Distributed Hash Table (DHT) where every peer is responsible for a certain range of hash-keys, which is delimited by its own key and the key of its predecessor, *pred*. In order to efficiently route messages in the network, every peer has a set of fingers to jump across the ring. When a new peer joins the network, it uses a hash key as identifier, joining between its corresponding *succ* and *pred*. In addition, the relaxed-ring allows loosely coupled peers that can be attached in branches when they cannot contact their predecessors. This property makes the system more robust and fault-tolerant. An extreme case of the relaxed-ring topology can be observed in Figure 3.3.

Taken from system theory, feedback loops can be observed not only in existing automated systems, but also in self-managing systems in nature. Several examples of this can be found in [145]. The loop consists out of three main concurrent components interacting with the subsystem. There is at least one agent in charge of monitoring the subsystem, passing the monitored information to a another component in charge of deciding a corrective action if needed. An actuating agent is used in order to perform this action in the subsystem. These three components together with the subsystem forms the

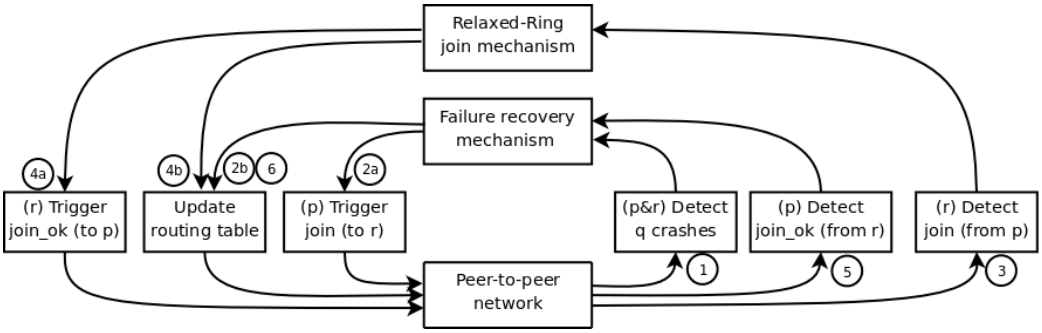


Figure 3.4: Failure recovery mechanism of the relaxed ring modeled as a feedback loop. The labels exemplifies the failure of peer q, placed in between peers p and r.

entire system. It has similar properties to PID-controllers, with the difference that the evolution of a running software application is measured discretely.

The results of modeling the relaxed-ring using feedback loops has been published in [101]. The work is focus on the fault-tolerant maintenance of the network under high churn, being the join and failure recovery mechanism the key parts of the architecture. Both feedback loops can be understood as independent loops, with the observation that we made the failure recovery mechanism reusing the join mechanism. This makes them easy to interact. In this deliverable we show the interaction of both loops, which is depicted in Figure 3.4. We use a concrete example to explain it.

Let us consider a particular section of the ring having peers p, q and r connected through successor and predecessors pointers. Figure 3.4 describes how the ring is perturbed and stabilised in the presence of a failure of peer q. Only relevant monitored and actuating actions are included in the figure to avoid a bigger and verbose diagram.

Initially, the crash of peer q is detected by peers p and r (1). Both peers will update their routing tables removing q from the set of valid peers (2b). But, since p is q's predecessor, only p will trigger the correcting event *join* (2a). This first iteration corresponds to a loop from the failure recovery mechanism. The *join* event will be monitored by peer r (3), starting an iteration in the join maintenance loop. The correcting action *join_ok* will be triggered (4a) together with the corresponding update of the routing table (4b). Then, the event *join_ok* will be monitored (5) by the failure recovery component in order to perform the correspondent update of the routing table (6). Since the *join_ok* event is also detected by the join loop, both loops will consider the network stable again.

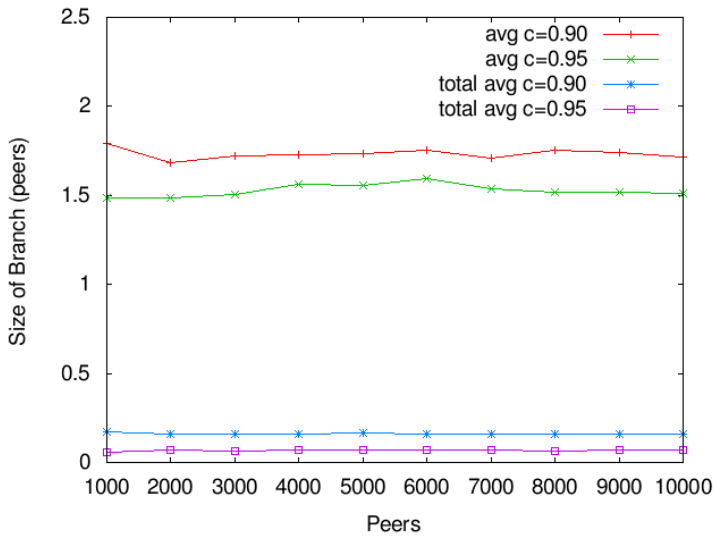


Figure 3.5: Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

One of the missing parts of the results of year one is the empirical validation of the relaxed-ring, and how the created branches influence the routing efficiency and the load of the network. To validate the relaxed-ring, we analyse four aspects: the amount of branches that can appear on a network, the size of branches, the number of messages generated by the ring-maintenance protocol, and the verification of lookup consistency on unstable scenarios.

The evaluation is done using a simulator implemented in Mozart, and that is released in software deliverable D1.5. In this simulator (called CiNiSMO, for Concurrent Network Simulator in Mozart-Oz), every node run autonomously on its own lightweight thread. Nodes communicate with each other by message passing using ports. We consider that these properties make the simulator more realistic. Every network is run several times using different seeds for random number generation. Charts are built using the average values of these executions. This deliverable will be focus on the size of branches, and the network traffic in comparison with Chord. Other results are presented in detailed in Appendix A.5.

Figure 3.5 shows the average size of branches depending on the quality of connections. The coefficient c represents the connectivity level of the network, where for instance $c = 0.95$ means that when a node contacts another one, there is only a 95% of probability that they will establish connection. A

value of $c = 1.0$ means 100% of connectivity.

The average size of branches appears to be independent of the size of the network. The value is very similar for both cases where the quality of the connectivity is poor. In none of the cases the average is higher than 2 peers, which is a very reasonable value. If we want to analyse how the size of branches degrades routing performance of the whole network, we have to look at the average considering all nodes that belong to the core ring as providing branches of size 0. This value is represented by the curves *totalavg* on the figure. In both cases the value is smaller that 0.25. Experiments with 100% of connectivity are not shown because there are no branches, so the average size is always 0.

We have also implemented Chord in our simulator. Experiments were only run in fault-free scenarios with full connectivity between peers, and thus, in better conditions than our experiments with the relaxed-ring. We have observed that lookup consistency can be maintained in Chord at very good levels if periodic stabilization is triggered often enough. The problem is that periodic stabilization demands a lot of resources.

Figure 3.6 depicts the load related to every different stabilization rate. Logically, the worse case corresponds to most frequently triggered stabilization. If we only consider networks until 3000 nodes, it seems that the cost of periodic stabilization pays back for the level of lookup consistency that it offers, but this cost seems too expensive with larger networks.

In any case, the comparison with the relaxed-ring is considerable. While the relaxed-ring does not pass 5×10^4 messages for a network of 10000 nodes, a stabilization rate of 7 on a Chord network, starts already at 2×10^5 with the smallest network of 1000 nodes. Figure 3.6 clearly depicts the difference on the amount of messages sent. The point is that there are too many stabilization messages triggered without modifying the network. On the contrary, every join on the relaxed-ring generate more messages, but they are only triggered when they are needed.

3.3.4 API

To be able to build services and applications on top of our structured overlay networks, it is necessary to define a simple high-level API that can express the functionality of the underlaying self-managing peer-to-peer network. The API we present and discuss here is inspired by OpenDHT [117], because it fits the ring topology used by DKS and P2PS, both presented as software deliverables D1.4 and D1.5. Those deliverables implement big part of the following API. The interface associated with Range queries and Chord$^{\#}$ is presented in detailed on its dedicated Deliverable D3.3a, Chapter 14.

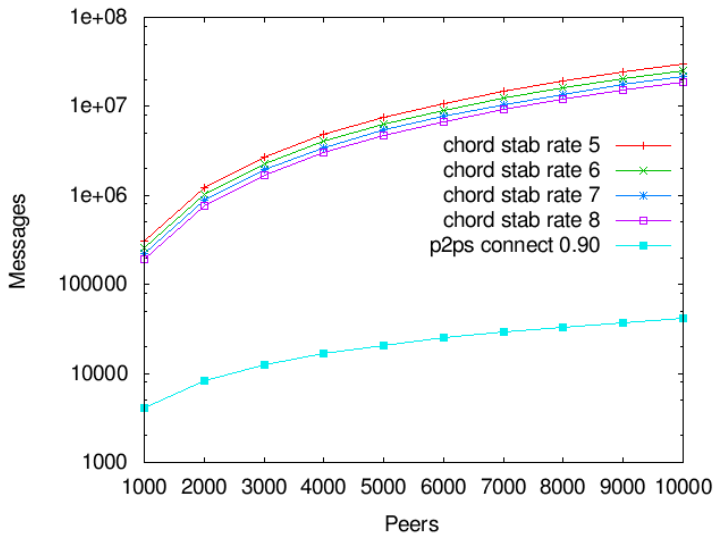


Figure 3.6: Load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

Basic functionality

The basic functionality that one can expect from a peer-to-peer network correspond to the ability of creating a node, make it join a network, lookup for other nodes in the network, and leave the network. We do not need to worry about neither detecting failures nor recovering from others peers crashes.

The API we provide is untyped. Its objective is to represent a language independent interface that can be implemented by different kind of peer-to-peer networks.

- `create_peer()`: Creates a node with an identifier *id*, and a *namespace*, which can be use to bootstrap a new network. Other peers can use this *namespace* to join this node.
- `join(id, namespace)`: It uses an identifier *id* to join a network identified by *namespace*.
- `lookup(key)` It finds the responsible peer for a given *key*. It returns the *id* of that peer. No specification about how the message is routed is given.

- `send(msg, key)` Reliable send of message *msg* to the responsible peer of the provided *key*.
- `direct_send(msg, id)` Reliable send of message *msg* to the particular peer identified by *id*. If the peer is not connected to the network, the message will not be deliver to the responsible of the *id*. This is why present *id* in contrast to *key*.
- `broadcast(msg, from, to)` It provides a pseudo-reliable broadcast of message *msg* to all peers responsible of keys in the range $[from, to]$. Since the functionality considers a range of keys, it can be used as a multicast. Using $[0, N-1]$ broadcast the message to all peers.
- `leave` This functionality explicitly disconnects the peer from the network. It is not necessary to implement because leave should be a particular case of failing. We include it to give the possibility to the implementing network to send leaving message for the sake of efficiency, not for correctness.

Basic storage functionality

This functionality is provided as the basic storage service that a DHT can offer. The implementing system is free to offer the replication guarantees that consider necessary. But, we assume that transactional operations, described in the following section, should offer the replication guarantees expected from a distributed data storage service, and that these group of functionalities should remain as a more basic set of operations without strong persistence guarantees. Still, all these functions requires lookup consistency provided by the underlaying network. The work presented in sections 3.3.2 and 3.3.3 target this issue.

We do not use secrets associated to keys, and we do not use time-to-live values either. This is because we are focused on applications such as the distributed wiki, described in WP3, where the data is not supposed to disappear after a fix TTL. The reason for not including secrets is because we are still analysing different approaches for security on DHTs, as it will be described in section 3.3.6

- `put(key, value)`: Stores de value *value* using the key *key*. The value will reside on the peer responsible of *key*.
- `get(key)`: Returns the value associated with *key*.
- `remove(key)`: Removes the value associated with *key*.

- `put_immutable(key, value)`: Store an immutable *value* associated with *key*. This means that no other value can be put using the same key, until the value is removed. Note that there is no need for an operation $get-immutable$, because its functionality is already covered by operation *get*.

Transactional functionality

This functionality is directly connected with the technical report "Transactional DHT Algorithms" [107] included in Appendix A.12, of Deliverable D3.1b. The objective of this interface is to provide access to transactional operations that use symmetric replication, as described in [57], given strong guarantees on persistence storage. If these operations are implemented following the mentioned technical report, they also provide strong guarantees with respect to failures of peers while the transaction is being process. If the majority of the peers is alive at the moment of committing the transaction, then, the transaction will commit.

- `begin_transaction` Returns a transaction identifier to be used in order to commit the transaction or to abort it. The transaction allows to atomically perform a set of *read* and *write* operations over one or more *items*.
- `write(item, value)` Write *value* to the item identified as *item*, using symmetric replication.
- `read(item)` Returns the value stored on the *item*.
- `commit(tid)` Commit the transaction identified by *tid*.
- `abort(tid)` Abort the transaction identified by *tid*.

3.3.5 Sloppy Management

As an unexpected result, we have observe the potential of sloppy management as an alternative approach for dealing with self-configuration and self-healing services on structured overlay networks. The proposal is that instead of explicit repair algorithms, systems could use continuous background probabilistic algorithms to handle non-functional management tasks such as routing table maintenance, while relying on the original structured algorithms for the functional tasks such as routing messages through a DHT. This probabilistic overlay maintenance is what we call "sloppy management". We still need to

further study the implications of such an approach. More details about this work can be found in Appendix A.4

3.3.6 Ongoing and Future work

The analysis on how to improve the results of this deliverable, contributed by NUS and INRIA, is presented here as ongoing work, motivating our future work on structured overlay networks. Being aware that Work Package 1 does not carries on during year 3, we expect to report progress on these ideas on WP4 Self-management services, more specifically on T1 self configuration and T4 self protection.

Security

The API presented in section 3.3.4 is based on the one of OpenDHT [117]. Therefore, our security analysis starts by spotting some issues on OpenDHT, and how they are related to our API.

First of all, with respect to availability, OpenDHT does not offer any load balancing mechanism to avoid data being discarded. Our system does offer symmetric replication by using the transactional procedures. Another difference is that storage quota in OpenDHT is offered per IP instead of per user, as in our system, where the quota is uniform, introducing different challenges we still have to investigate.

One risk from our system is that we have removed the time-to-live value from the stored entries. This is because our storage services are meant to be persistent, as in the Distributed Wiki presented in Deliverable D3.1b, Chapter 12. This implies that our stored data could grow forever, just as the size of Wiki can grow forever. Of course, this risk is driven by the application we want to implement.

With respect to Sybil attack, we are expecting to apply the results from Deliverable D1.3b, that propose the use of social networks to prevent it.

Slicing

The ongoing work on slicing is inspired on the results presented on [48]. The objective is to categorize the network by analyzing the available resources of each peer. The criteria to use considers processing power, storage capacity and locality, among other variables. By having a slicing service one can partition the network into groups that represent a controllable amount of some resource, where peers present homogeneous capacities.

Future Work

Considering what has been described as ongoing work, it is absolutely necessary to carry on with the work on security. Since there are other work packages specially dedicated to this topic, we believe that it is feasible to continue the work and to integrate it during the next year of the project. The work on slicing should also be continued in order to decide if it provides some benefit to our project.

With respect to the API, even when it is quite stable, it is expected that we will have to adapt it and extend it during the development of services and applications on top of the structured overlay networks we have developed on this deliverable. We consider such iteration as a normal process in the life cycle of any software development and research project. As mentioned in the beginning of this section, the results of the ongoing and future work is expected to be reported in other work packages, mainly in WP4, dedicated to self-management services.

We can conclude that the amount of work that remains to be done in this deliverable is reasonable small to be assimilated and integrated by other work packages in the next year of the project.

3.4 Novel P2P NAT Traversal Approach

In this section we present a new method for NAT Traversal designed in the context of Peerialism's peer-to-peer content distribution system. The suggested method is based on a widely adopted technique on current home routers: NAT Port Preservation. The method advances the current state-of-art of NAT Traversal by improving support for two widely deployed types of NAT: Restricted Cone and Port Restricted Cone.

3.4.1 Network Address Translators

Network Address Translators (NATs) have been deployed ubiquitously in today's Internet. Network home routers and corporate firewalls are the most common examples of devices which implement the Network Access Translation Technique. However, as a result of a lack of standardization of this technology, the behavior of NAT devices is often vendor-specific.

Network access translators have been mostly designed around the client-server paradigm. The client lies in a private network where access to the open Internet happens only through a NAT device. Servers are instead located ouside that network, in hosts which have public IP addresses and static DNS records. The great advantage of NAT-enabled devices is their ability to translate connections coming from their internal hosts such that they appear to the external servers to be issued from the NAT device, thus protecting the clients by hiding their real addresses in the private network. Network Access Translators also provides an additional form of protection, which is dropping of incoming unwelcome connections that were not first initialized by internal hosts. This approach has been designed to support Web or e-mail services, where it is always the hosts inside the private network that initiate the communication to the outside servers.

When peer-to-peer applications are involved instead, hosts behind NAT might need to accept connections from other peers and not only to initiate them. In this case, the NAT "client-server" approach might constitute a problem. A typical example of peer-to-peer application which from this problem is a VOIP system where calls need to be established between pairs of peers with the help of a central rendez-vous server. Typically, the peer would register to the rendez-vous server and obtain the address of another peer that it wants to communicate with. It would then try to connect to the latter to establish a call. If the destination peer is behind NAT, the address obtained by the rendez-vous server will be either the private address of the peer or the address of the NAT device, or even both. This is not enough to establish a connection between the two peers, since the NAT device will not

expose any open ports for the call initiator to contact the destination peer.

A number of methods have been studied to avoid this problem. The first step in solving the problem is to define the behavior of different NATs and group them together in different types. This has been done in the STUN RFC. We summarize the types as follows:

- Full cone. Once a host behind a NAT initiates a connection to another host outside its private network, and the NAT box has establishes a mapping for send packets back to the initial host.
- Restricted cone. When a private host initiates a connection to another for allowed to respond back to the host inside the private network. Any other packet coming from a different host will be dropped.
- Port restricted cone. If a private host initiates a connection to a host with address IP1 and port number N2, only packets coming from that same IP and port will be forwarded back by the NAT device to the private host.
- Symmetric. Each request coming from an internal host with a certain IP address and port to a specific destination IP address and port is mapped by the internal host initiates a connection from the same source address and port but to a different destination, a different mapping is used.

3.4.2 Current NAT Traversal methods

IGD and NAT-PMP

The Internet Gateway Device(IGD)[51] and the NAT Port Mapping(NAT-PMP)[32] protocols are Application Layer Protocols which allows for discovery and configuration of port forwarding rules in NAT-enabled devices. IGD is implemented via UPnP[79], it has been widely adopted by router producers but it is not an Internet Engineering Task Force document. NAT-PNP instead is an Internet Draft which has been filed by Apple and it's mainly used in Apple routers. Both protocols enables hosts to discover UPnP and NAT-PMP-compliant routers in their local network, learn their public IP address, retrieve and modify existing port mappings and assign corresponding lease times.

STUN

STUN[121] is a network protocol which allows a host to discover which type of NAT it is behind and which public IP address is associated with it. STUN

does not require any explicit interaction with the NAT device as in UPnP and NAT-PMP, it instead makes use of an external STUN server to test the behavior of the NAT device through a well-defined discovery process. The process consists in first contacting the STUN server to learn the observed public IP of the client and then in requesting the STUN server to send UDP packets back to the acquired IP using different combinations of source IPs and ports. The client will be able to understand which kind of NAT it is behind according to which packet it will be able to receive as different types of NATs handles incoming UDP packets in different ways. STUN provides also a method to discover which port has been opened by the Network Address Translator on the public interface. If the STUN client communicates this information to a public rendez-vous server, it can be used by other peers to establish a direct connection with the same host. For more information about the STUN protocol, please refer to[121].

TURN

Traversal Using Relay NAT (TURN)[120] is a protocol which enables a host behind a NAT or firewall to receive incoming data over TCP or UDP connections using relaying. It is mostly useful for hosts behind Symmetric NATs or firewalls with similar behavior. Clients running TURN must authenticate to a server in the public domain. When a TURN client wants to send data to another one, it sends it to the server which will in turn forward it to the the destination. Although TURN provides connectivity between clients in almost all the cases, it has the cost of relaying all the traffic through a public server. Therefore, TURN is used as last resort when all attempts to establish a connection with other protocols, such as STUN, have failed.

ICE

The Interactive Connectivity Establishment (ICE)[119] provides a comprehensive mechanism for NAT traversal. The ICE approach consists in deploying a number of STUN and TURN servers in the public network and automating the choice of which of the two protocols to use, such that it appears as transparent for the client running the ICE protocol. In particular, it is used to allow SIP-based VoIP clients to successfully traverse the variety of firewalls that may exist between a remote user and a network.

3.4.3 Peerialism's system

Peerialism's media distribution platform performs audio and video streaming using proprietary peer-to-peer technologies. The target customers of our products are home users. The customer would typically install a client application in its machine and start requesting content. Every running client application constitutes a peer in Peerialism's system. Peers collaborate to distribute content over an ad-hoc overlay network managed by a central coordinator, called Tracker. The Tracker is entitled with the task of periodically reorganizing the network such that the content distribution's load is balanced among all participating peers. The content itself is composed by a number of different streams which consist of RTP and RTSP packets sent over UDP. Given the characteristics of the system, it is therefore crucial that Clients are able to connect to each others and exchange parts of the stream. However, this is not always possible when peers lie on private networks behind NAT. In the next two paragraph we will explain the limitations of the current NAT technologies which have been discovered during the process of building our system. Then we will expose our simple approach to reach connectivity between peers using UDP.

3.4.4 Limitations of current approaches

STUN is sometimes referred as the universal solution for NAT Traversal when speaking about peer-to-peer applications. This is a common misconception. STUN, as mentioned in the STUN RFC, "..will work for any application for full cone NATs only. For restricted cone and port restricted cone NAT, it will work for some applications depending on the application. Application specific processing will generally be needed." However, the RFC does not define which kind of behavior the application should implement to cope with the limitations of the protocol. A comprehensive analysis concerning the state of peer-to-peer communication across NATs has been recently published[135]. It complements the STUN RFC by describing in details how a peer-to-peer application should behave when multiple levels of NATs are involved, when peers are behind the same NAT device or when using traffic relaying. The analysis underlines that the common approach used by peer-to-peer applications when dealing with NATs is to use a public host which behaves both as a rendez-vous server, like in STUN, and as a coordinator for connection establishment between peers. An earlier paper by Ford et al.[50] documents similar hole punching techniques and analyzes their reliability on a wide variety of deployed NAT. However, both documents suggest the use of a technique called port prediction[154][157] when Restricted, Port Restricted

and Symmetric NATs are involved. Port prediction works by analyzing the behavior of the NAT device and attempting to predict the public port numbers it will assign to future outgoing communication sessions. As mentioned in [157], this technique is not reliable as NAT devices might implement unpredictable behaviors when assigning public ports. Furthermore, it's almost impossible to predict which public port will be allocated for a certain session when many peers are behind the same NAT.

Two independent surveys[149][138] show the distribution of pre-configured NAT types in common home routers. The results reveal that Restricted and Port Restricted types of NAT account for a significant share of the market, as much as 40%. However, all tests take into consideration the type of NAT configured by default on the devices, which might be changed by the user as home routers usually implement more flavours of NAT on the same device. Even though the validity of the surveys might be questionable, the first one being unofficial and the second one being somehow outdated, statistics collected during our preliminary system tests show similar results on the home routers of our customers. It's therefore important to provide a good degree of support to those NAT types.

The only existing protocols which guarantee peer connectivity for all NAT types are IGD, NAT-PMP and TURN. IGD and NAT-PMP are usually disabled by default in home routers since they might expose internal hosts to serious security threats[60]. However, in our system we do make use of these protocols when available. TURN instead, and consequently ICE, suggest an approach which consists in relaying peer-to-peer traffic through a public server. In our system, this technique cannot be used since the cost of relaying multiple video and audio streams through a public server is too high. In fact, it would require an enormous amount of both computational and bandwidth resources on the public host. Therefore, when connectivity cannot be achieved between two peers, it is better to redirect the requester of the content to another provider which is publicly reachable instead of relaying traffic.

3.4.5 Our solution

We base our NAT Traversal solution on previous studies[135][50][157], which we extend to provide better support for Restricted and Port Restricted NATs. In our approach, we exploit a widely adopted technique in currently deployed home NATs: port preservation. We refer to port preservation as the attempt of a NAT box to use the same external port as the internal host used to establish a connection to an outside host. This means that, for instance, if Client A behind NAT tries to contact Server B in the public domain from its

local port 123, the NAT device will try to use port 123 on its public interface to communicate with Server B. Port preservation may be supported by all types of NATs except the Symmetric one. In fact, by definition, Symmetric NATs allocate a different port on the external interface every time a host tries to contact a different endpoint, where the endpoint is given by IP address and port. Port preservation is defined in [154] as "Exceptional Behavior". We argue that port preservation is becoming more and more common as NAT vendors adapt their products to support peer-to-peer applications. That is confirmed by tests we performed on a limited set of home routers currently on the market.

Given the assumption of port preservation, we derived a connectivity table, showed in Figure 3.7. It represents the compatibility between pairs of peers according to their NAT types. The compatibility is only theoretical since there is no guarantee that a certain NAT device will behave as assumed by the classification of NAT types or that it implements port preservation. Combinations of NAT types are grouped in five different classes in the table. Every class defines which method will be used when attempting to establish a connection between a pair of peers. The last class instead defines the combinations where port preservation cannot be used to obtain connectivity.

In our system, the connectivity table is used by the Tracker application to decide whether it is possible for a host to provide content to another one when NATs are involved. The Tracker is then aware of the NAT types of its clients. This because, when started, the Client application performs a STUN discovery and reports to the Tracker the following information:

- Its private IP address and listening UDP port.
- The public IP address of the NAT device and the external port mapped during the STUN discovery process.
- The type of NAT implemented by the gateway.

We now detail the aforementioned classes providing examples in the context of our system.

- **(I)** Only one of the two Clients is behind NAT. In this case, we implement a classic Client-Server behavior: the peer behind NAT connects to the peer in the public domain. The peers assume their role of content provider or content requester only after the connection has been established.
- **(II)** One or both peers are behind Full Cone NAT. In this case, both Clients start to send to each other's public endpoint at the same time.

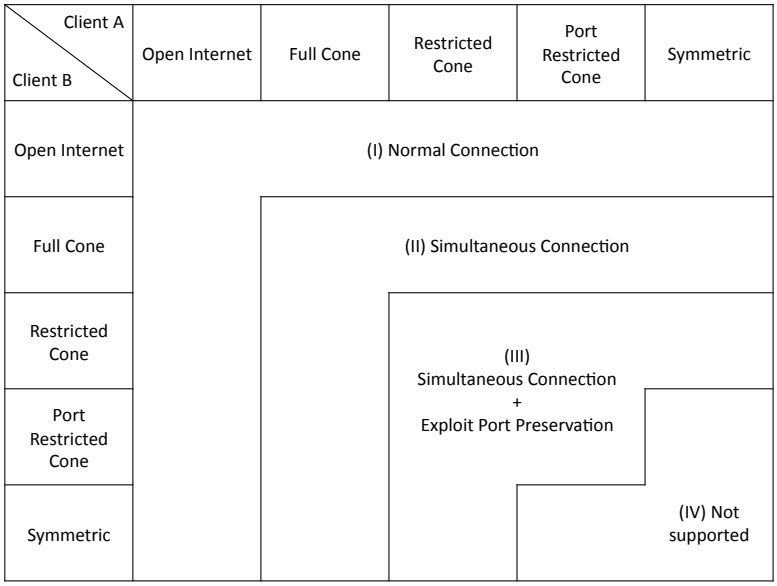


Figure 3.7: Connectivity Table which shows the compatibility between pairs of NAT types. It also dictates which approach should be used in establishing a connection between peers behind those types of NAT

This is necessary to create the translation entries in the two NAT devices and make hole punching possible. If a peer is behind Full Cone NAT, its endpoint is the public IP address and port learnt during the initial STUN discovery. Since a device implementing Full Cone NAT is expected to behave as endpoint-independent, the other peer will be able to use that same endpoint for contacting the peer behind Full Cone NAT.

Consider now the scenario in Figure 3.8. Both Client A and Client B are behind NAT, A is behind a Full Cone NAT and B is behind Port Restricted NAT. Both peers have previously performed a STUN discovery and reported the findings to the Tracker. Client A now makes a request for a certain stream sending a Content Request message through its TCP connection to the Tracker. The Tracker then decides that Client B should provide the content A requested. The Tracker issues notifications both to Client A and to Client B at the same time, in the figure called Receipt Notification and Delivery Notification. The message sent to Client B contains the public endpoint of Client A, and viceversa. Note that the public endpoint of Client A is composed by the NAT device's public address and the external port learnt through

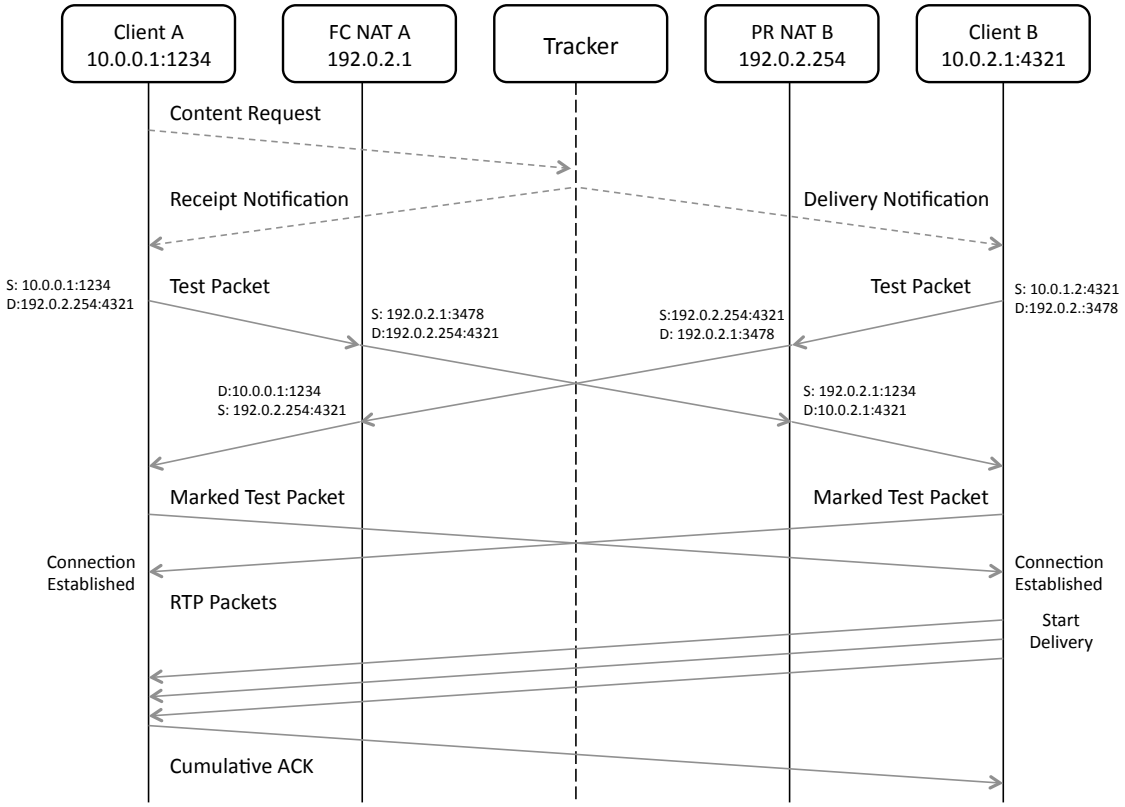


Figure 3.8: NAT Traversal: Connection establishment process of class II

the STUN discovery, in this case port 3478. Instead for Client B, the public endpoint is NAT B's public address, and the port is the one used locally by the peer, port 4321. The peers then start sending test packets to each other's public endpoint at the same time. As mentioned earlier, this is to guarantee that the NAT devices create their translation entries as soon as possible. If the entries are not present before the first packets are received, those packets will be dropped. However, next packets will be forwarded correctly to the private host.

Now, if the first test packet sent from Client B opens port 4321 on NAT B's external interface as expected by port preservation, test packets from Client A will be forwarded correctly to B. Port 3478 on NAT A instead is supposed to be still open since it was used for the STUN discovery. We make sure to keep this mapping active by sending periodic messages to the STUN server in the periods the Client A is not receiving or providing any content. If Client A receives a test packet, it marks

it as received and sends it back to the source, the same does Client B. When Client B receives the marked packet from A, it starts to deliver data. Note that Client B might have previously received an unmarked test packet from A but it will not start to deliver at that point. This is to guarantee symmetric connectivity between peers, since it might happen that Client A can send to Client B but it is unable to receive from him, in particular because of both Clients being behind NAT. Finally, Client A periodically acknowledges received stream packets to Client B sending cumulative ACK messages.

The aforementioned connection establishment process works even if NAT B does not implement port preservation. In that case what would happen is that Client A receives a test packet from an endpoint which has the public address of NAT B but has a different port than the one notified by the tracker. In that case, Client A would update Client B's endpoint and start sending test packets to the new endpoint.

- **(III)** Peers are either behind Restricted or Port Restricted Nat. In this case we rely on port preservation to know which port will be open by the NAT device on its external interface.

  Let us consider the scenario showed in Figure 3.9. Both Client A and Client B are behind Port Restricted NAT. As in the scenario showed in Figure 3.8, Client A makes a request for a certain stream to the Tracker. The Tracker then decides that Client B should provide the content. Again, the Tracker issues notifications both to Client A and Client B at the same time. However, in this case the public endpoint of Client A is 192.0.2.1:1234, which is NAT A's public address and the private port which Client A is listening on locally. Similarly, Client B's public endpoint is 192.0.2.154:4321. The peers then start their connection setup phase sending to each other's public endpoint. If NAT A and B behave as expected, they will open the same external port as the one their internal hosts are sending from, namely port 1234 for NAT A and port 4321 for NAT B. If this happens, both peers will be able to receive test packets and reply to their counter-parts. Client B will then start delivering content to Client A.

- **(IV)** This class is not supported since port preservation cannot be used to predict which port will be allocated by a Symmetric NAT on its external interface.

  The class includes two combinations of NAT types: Port Restricted-Symmetric and Symmetric-Symmetric. In the first combination, even

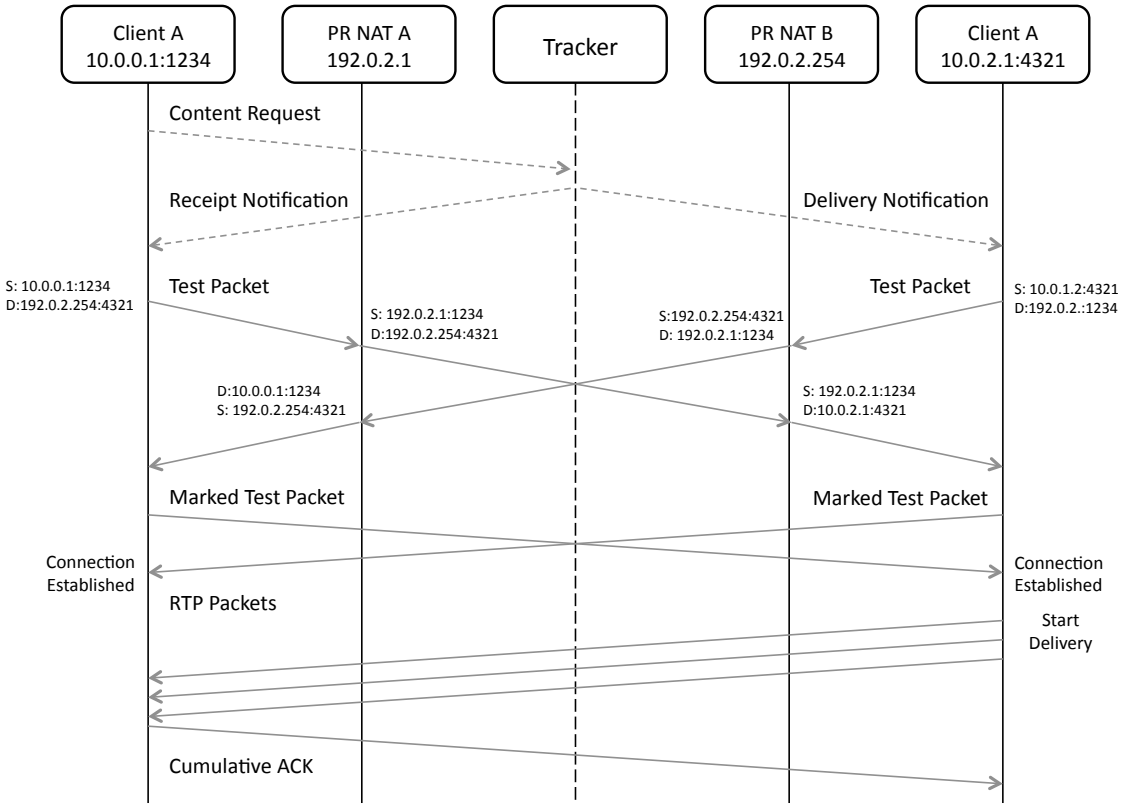


Figure 3.9: NAT Traversal: Connection establishment process of class III

if the NAT implementing a Port Restricted behavior supports port preservation, there is no way of knowing which endpoint should be used when contacting a client behind Symmetric NAT. The same is true for the second combination, where both endpoints are completely unpredictable.

To show the incompatibility between combinations of NAT types included in this class, let us assume that a Client A is behind Port Restricted NAT (NAT A) and wants to connect to another Client B behind Symmetric NAT (NAT B). The Tracker can provide Client A with NAT B's public address, the local port which Client B is listening on and the port which the Symmetric NAT allocated for the STUN discovery. Let us now say that Client A starts to send to the public endpoints given by the Tracker, first to NAT B's public address and Client B's local port, then always to NAT B's public address but to the port that Client B learnt during its STUN discovery. NAT B will drop the incoming packets since they come from an endpoint that was not previously

contacted by Client B. If Client B tries to establish a connection at the same time, as described in (II), NAT A will drop the incoming packets since the new endpoint that the Symmetric NAT allocated is not in NAT A's translation map.

Given the limitations of port preservation for combinations of NAT types included in class (IV), it is necessary to use the Port Prediction technique to obtain connectivity between pairs of peers whose NAT combination falls in this class.

3.4.6 Conclusion and Future Work

In this document we presented the current NAT Traversal technologies for peer-to-peer applications and their limitations. We then suggested and detailed a new method based on the Port Preservation technique to improve hole punching when Restricted Cone and Port Restricted Cone NAT are involved.

We now plan to evaluate the suggested NAT Traversal method in Peerialism's deployed platform. Before doing that, we intend to add support for Port Prediction, as defined by Takeda et al [157], to be used when Port Preservation alone does not provide connectivity. We would also like to provide valid statistical results on the distribution of NAT types in today's home routers as observed in a real system, such as Peerialism's platform. We would then like to combine those results with the success rate of our NAT Traversal technique depending on the various combinations of NAT types.

3.5 Papers and publications

This section is dedicated to give a brief introduction to the publications produced by the work done on this deliverable. Some of them are included as appendices of this book, as explained here below.

The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay

This paper has been accepted with revision in the Journal "Parallel Processing Letter". It is annexed in Appendix A.5, and it includes the results presented in two other publications mentioned in the Periodic Activity Report, but that are not included as appendixes: "A Relaxed-Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks" and "Improving the Peer-to-peer Ring for Building Fault-tolerant Grids". This article concludes the work of low-level primitives, motivating the work on high-level primitives for self-managing systems.

Range queries on structured overlay networks

Similarly to the article on the Relaxed-Ring, this paper concludes the work on low-level primitives for range queries, motivate the work on high level primitives. It is included in Appendix A.3

Sloppy Management of Structured P2P Services

This publication is the result of a collaboration between ZIB and the Vrije Universiteit Amsterdam. It proposes sloppy management for structured overlay networks using continuous background probabilistic algorithms instead of explicit repair strategies. This work is included in Appendix A.4

Chapter 4

D1.3b: Final report on Security in Structured Overlay Networks

4.1 Executive Summary

SELFMAN aims at building self-managing distributed systems. Self-management is achieved through mechanisms which provide: self-configuration, self-healing, self-tuning and self-protection. Although SELFMAN has self-protection, the objective of SELFMAN is not to develop highly-secure distributed systems, which would be too ambitious an objective. Rather the self-protection aspects are to provide mechanisms which can enhance security.

This deliverable presents results which provide security functionality which is useful for the self-management work in Workpackage 4 (WP4). Although security is not the primary focus, an attack on a distributed self-managing application can affect self-configuration, self-healing and self-tuning as well. Thus the results in this deliverable are both to do with mechanisms for self-protection and partly about making other self-management aspects of SELFMAN more resilient to attacks.

Most of the works in Structured Overlay Networks (SONs) are based on the notion of a Distributed Hash Table (DHT). Unfortunately, while DHT-based SONs has many desirable properties from a self management viewpoint, they also have many security drawbacks. The first report on security, D1.3a, suggested a promising direction which moves away from the fundamental drawbacks of DHT (from a security perspective) by employing Small World Networks (SWN). SWNs can get around the problems of identity which can plague DHTs and being more robust also simplify self-healing and

self-configuration.

We have built a testbed for investigating SWNs which provides the following functionalities: creation of various SWN, simulation of a network using the SWN, visualizations of the experiment including statistics. The most important feature is that the testbed makes it easy to test different algorithms which operate on the SWN. We have successfully run simulations of up to 100000 nodes without any problems. The intention of the SWN testbed is to investigate whether a SWN can be a suitable replacement for a DHT. Our experimental results show although routing guarantees are probabilistic, we found that the routing success rate to be around 100%.

Self-configuration and self-healing mean that a self-managing system will add/modify/update software components on the fly. We have implemented a software component authentication system which can guarantee that only authorized code can be used. This means that we can be sure of the provenance and identity of all software components in the system. Our prototype runs in Windows, the most common platform where many subtle attacks on components are possible. We protect against all attacks which try to load malware to replace a software component.

A SELFMAN application would communicate with others on the Internet. Software often contains bugs, and we expect SELFMAN applications not to be an exception, which can be exploited externally by attackers. As such, monitoring the behavior of multiple processes communicating over the network can be used to detect unexpected or illegal behavior. We have developed a monitoring infrastructure which can capture the behavior of a collection of processes and threads. It has low overheads and allows actions to be related to the software components which cause them. Low overheads means that permanent monitoring is feasible. We tested monitoring on Skype running on several machines. The results demonstrate that one can clearly see that Skype is a Peer-to-Peer (P2P) application from its network traffic flow and once can observe how Skype works.

4.2 Contractors contributing to the Deliverable

UCL (P1), KTH (P2), INRIA (P3), FT(P4) and NUS (P7) have contributed to this deliverable. Self protection issues are related to other self properties and the work here on basic underlying mechanisms for security are between NUS working on the security aspects with the cooperation of other partners who have contributed to design or requirements.

UCL (P1) UCL has cooperated with NUS to refine security issues for the low level SON.

KTH (P2) KTH has cooperated with NUS to refine security issues for the low level SON.

INRIA (P3) INRIA has cooperated with NUS in the design of aspects of the monitoring infrastructure which allows for secure monitoring for self-management.

FT (P4) France Telecom has contributed to the design of messaging requirements and use cases from an application standpoint which is the starting point used by NUS for the Small World Network routing testbed.

NUS (P7) NUS has designed and implemented the Small World Network testbed and simulator being a new low level SON-like infrastructure, the WinResMon monitoring infrastructure and the BinAuth software component authentication infrastructure.

4.3 Introduction

In general, making a non-distributed application secure is a difficult task. Making distributed self-managing systems and applications, as in SELFMAN, secure is correspondingly much more difficult. Nevertheless, in this age where attacks on software are common, some security mechanisms are necessary. The goals of SELFMAN with respect to security are not to provide high security, which would be simply too ambitious, rather to provide security mechanisms which can help self-protection in self-managing distributed systems. This includes increased self-protection for the overlay networks used in SELFMAN.

Security for a system needs to be approached from a holistic perspective. Self-protection for the (structured)[1] overlay network is one aspect. Other aspects include whether self-tuning, self-configuration or self-healing have been compromised. One also needs mechanisms to help determine whether malicious behavior is occurring in the distributed system/application.[2]

In the D1.3a deliverable, we surveyed security in P2P systems and identified that often the final level security properties desired are specific to the needs of the P2P application. In this workpackage, such domain/application specific security measures are not part of the low level infrastructure and are not dealt with here. We also identified Structured Overlay Networks (SON) such as Distributed Hash Tables (DHT) as having a number of fundamental drawbacks. The two most important are the difficulties of dealing with identities in a decentralized setting and the problem of maintaining the distributed data structures which comprise the SON. The former leads to the problem of Sybil attacks [47] which are problematic for SONs. In the latter, e.g. in a DHT like Chord, periodic stabilization is used for finger maintenance under churn, the effort for maintenance can itself lead to more security problems such as routing and data attacks.

We proposed in D1.3a that rather than trying to retrofit more security mechanisms to a DHT, there are already many proposals but they are not satisfactory, to take an orthogonal approach and avoid some of the drawbacks. A Social Network can automatically provide trust relationships which mean that the problem of Sybil attacks disappears since it is not feasible to create multiple identities easily. Social networks are a form of Small World Network (SWN). Routing is also simpler in SWNs which reduces the problem of routing attacks.

[1] We will later advocate studying small world networks which have a mix of random and structured properties.

[2] We cannot hope to guarantee that no malicious behavior occurs since that question is undecidable in general.

We built a SWN testbed for investigating the suitability of SWN as a replacement for a DHT. This is complementary to the other work in SELFMAN in task T1.1 which makes use of DHT-based SONs. The SWN testbed can generate and simulate several types of SWNs. It provides visualization as well as statistics. Our testbed is able to handle reasonably large networks, we have run simulations on the order of 100000 nodes. Our experiments show that although in a distributed setting routing algorithms based purely on local information only have probabilistic guarantees, in practice, experiments show that the probability of success is very high. We report further on self-protection in SWN in D4.4a.

The other security mechanisms described in this report serve to gain better self-protection and assist in the other self-* tasks in WP4. The monitoring infrastructure allows one to understand and analyze the behavior of a distributed application organized as a collection of processes and threads running on various machines. The software component authentication ensures that only those components which are trusted can be executed. This allows complete control over what software components are loaded and executed in a system.

4.3.1 Relating D1.3b and D1.3a

We take the opportunity to clarify the work performed between deliverable D1.3b and D1.3a. The SELFMAN project started late at NUS[3] and the work reported in D1.3a was for a period of three months. Given the short timeframe, the primary focus of D1.3a was to understand the major security issues in SONs and P2P systems. Some work was also started on the other tasks in WP1 in this timeframe which was not reported simply because they were preliminary. In deliverable D1.3b, as we shifted focus from SON to SWN, some of the initial effort was no longer relevant.

The usefulness of D1.3a was that it allowed us to identify some of the key problems which are pertinent to self protection in structured overlay networks. The work in this deliverable reflects the directions adopted from the results in D1.3a. We also identified from D1.3a that only two of the applications in SELFMAN would have relevance from a security standpoint, namely the M2M application and the Wiki scenario. The P2P TV application is being a closed system doesn't require security.[4] We remark that

[3]Due to late receipt of the SELFMAN funding, work on SELFMAN could only effectively start on Feb 2007. The work in D1.3a was for the timeframe of Feb to May 2007

[4]As far as we understood, they are interested in the self-management aspects other than self-protection.

the M2M application has some generic properties which can typify a more generic application for which security mechanisms can be used since it performs data collection and monitoring. Based on our existing experimental results on SWNs, we believe that use of a SWN may simplify and make the self-management and networking aspects of the M2M application more robust. The Wiki application being more specialized may ultimately need more domain specific mechanisms such as Wiki specific trust management.

4.4 Self Organizing Networks with Small World Networks

In the D1.3a survey, we found that structured overlay network such as those using DHTs have many drawbacks. The main ones arise from problems with identity in a P2P setting and the need for constant maintenance of the DHT data structures. One of the most problematic is the Sybil attack [95]. which exploit the lack of identity. Unfortunately, it is not possible to stop Sybil attacks while retaining the decentralized properties which make DHTs attractive.

The problem of identity in DHTs arises because of the decentralized and P2P nature of a DHT is not conducive to creating trust. One way of increasing trust is to have networks which are based on trust or friendship and these provide a natural defense against Sybil attack. One network which naturally has these properties is the so-called Social Networks. For example, in a social networking site, e.g. Facebook, Friendster, LinkedIn, etc, we usually add (real/unique) people that we know directly as our friends. The nature of the social network verifies the identity of the nodes. Thus a social network can be thought as "full" of identities. A social network is usually thought of as a kind of Small World Network (SWN).

The classic experiment by Milgram on social networks show that the chain of social acquaintances required to connect one arbitrary person to another arbitrary person anywhere in the world is generally short (this is the origin of the phrase "*six degrees of separation*" and concepts such as Erdos number). The motivation for investigating SWN is that the network has automatic identity properties which reduce the problem of Sybil nodes. In addition, SWN do not require much maintenance as the SWN graph does not change frequently. As we will see, it is possible to route efficiently in a SWN. In this section we introduce Small World Networks (since they might be less familiar). The SWN infrastructure described here is used later in deliverable D4.4a (Section 18).

4.4.1 Small World Network Models

A SWN as defined by Watts and Strogatz [155] is a graph with small diameter and a high clustering coefficient. Examples of SWN are: social networks, electric power grids, neural networks, telephone call graphs, paper authorship relations (Erdos number), etc. There are many ways to model a SWN, here, we describe two SWN models: the Watts-Strogatz model [155] and Kleinberg model [86].

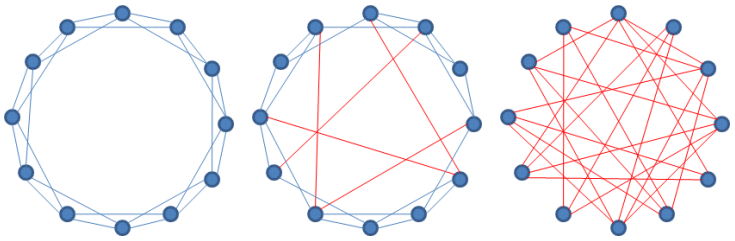


Figure 4.1: Watts and Strogatz Model

The Watts and Strogatz SWN construction depicted in Figure 4.1 first starts with a regular graph where each node is connected to its $k-$nearest neighbors (Figure 4.1 left-most graph). This graph has no shortcuts, high clustering coefficient, and large diameter. With probability p, the links of each node in the regular graph are rewired to a node chosen uniformly at random over the entire ring. This results in edges being rewired to act like shortcuts links to far away nodes. The middle graph in Figure 4.1 is called a Small-World network by Watts and Strogatz. It has quite a high clustering coefficient and diameter is small, expected $log(n)$. If we continue rewiring the links, the graph will no longer has high clustering coefficient. It now resembles a random graph, Figure 4.1 right-most graph. Note that the diameter can still be small.

The path length L is defined as the average shortest path length between any two vertices. The clustering coefficient C is defined as the average of the fractions of the neighbors of each node that know each other. $L(p)$ and $C(p)$ is the path length and the clustering coefficient of the graph that are constructed with the algorithm above with probability p. Figure 4.2 shows that with $p = 0$, the graph is a regular graph where all nodes are connected with it's $k - nearest$ neighbors which has high clustering and also large diameter. With $p \approx 0.01$, the graph still has high clustering but the path length has decreased significantly which is the SWN defined by Watts-Strogatz. With $p = 1$, the graph is totally rewired thus has no clustering.

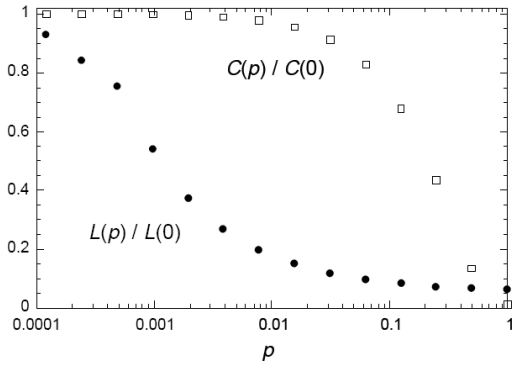


Figure 4.2: Characteristic Path Length $L(p)$ and Clustering Coefficient $C(p)$

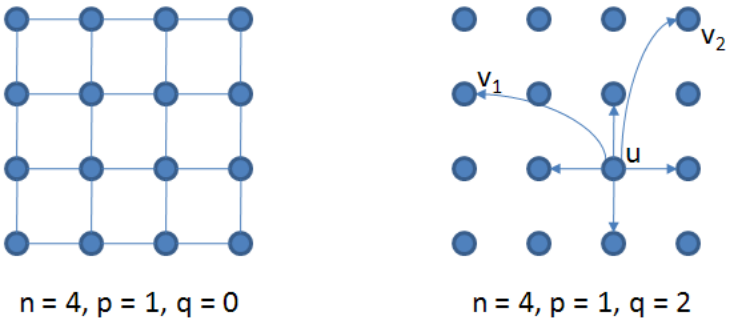


Figure 4.3: Kleinberg Model

The construction of the Kleinberg model [86] first starts with the links and nodes connected like a lattice (Figure 4.3 left-most graph). Then a constant number of non-local links are added. Kleinberg add two shortcuts with probability proportional to $d(u,v)^{-r}$ (Figure 4.3 right-most graph) with r as the dimension of the graph. The shortcuts make the diameter of the graph small with expected $log(n)$. The node identifiers are assigned based on the lattice coordinates and the cost of greedy routing is in expected $O(log^2(n))$ steps.

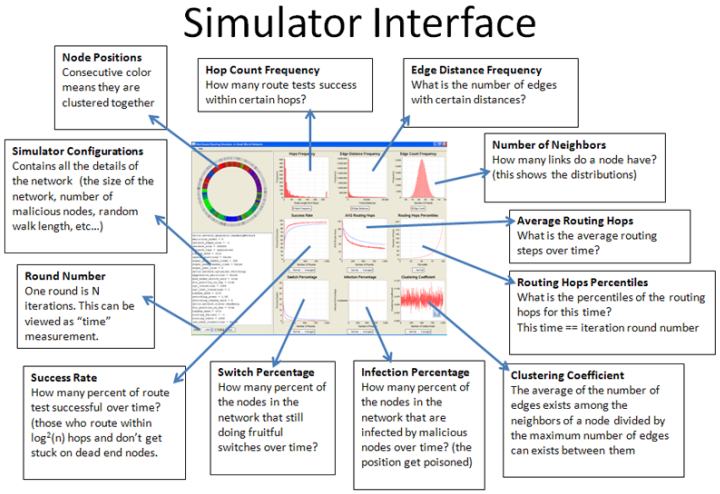


Figure 4.4: Simulator Interface

4.4.2 Small World Network Testbed and Simulator

We build a SWN testbed (see Figure 4.4) to allow us to experiment with SWNs. The testbed contains a number of generators for SWNs and also SONs. It has a simulator which monitors routing performance (average routing hops), success percentage (how many % of the routing are successful), node positions (visualized as a ring), routing hops percentiles (to tell how robust the routing for all percentiles), and other statistical distributions such as hop-counts, edge-distance, edge-count, etc. The simulator allows us to easily build new SWN algorithms and experiment with them. The SWN testbed GUI has extensive use of visualization and animation which is useful for understanding performance of the SWN.

To handle large networks, we have a parallel version of the simulator which runs on our cluster. We have run networks up to 100000 nodes and

depending on the type of experiment, typically these take between 1 minute to under an hour. Thus, our SWN testbed platform can handle realistic network sizes.

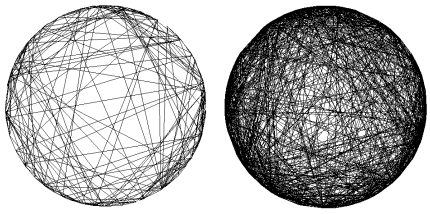


Figure 4.5: $log(n)$ and $6 * log(n)$ rings

4.4.3 Comparing Small World Networks

Typically, DHTs use $O(log\ n)$ or more links per node. Our experiments show that for a SWN, $O(log\ n)$ links also work reasonably well, We experiment with a Normal SWN (this is called "normal" as it has the same number of links as typical DHTs) with $log\ n$ links, and the Sandberg SWN which has $6log\ n$ links. Figure 4.5 shows a normal and Sandberg SWN witn $n = 100$. We also use the Kleinberg SWN with just 4 links.

To test the SWN simulator and to get some initial insights into the performance of SWN models and to determine what are the important factors, we conducted routing experiments on all the above three models. We use 100000 nodes and in order to establish baseline performance, this experiment assumes no node or link failure.

The results of the experiment are shown in Figure 4.6. It shows that SWN is more robust in terms of routing length if we add more edges to it. The Kleinberg model which has only 4 constant edges has a very large deviation in routing length between 1 and 80. The Normal model which has log(N) edges has moderate deviation of routing length from 1 to 30. The Sandberg model which has 6*log(N) edges has very small deviation of routing length from 1 to 10. We has 100% success rate for all the experiments.

This experiment shows that as a basic network, a SWN can have good performance, though it might need more edges than a DHT. The performance is actually rather good, since much less structure is needed and also only a few assumptions. For example, given a social network, one already automatically has a usable network. A DHT on the other hand is more complex. As this deliverable is concerned mainly with the low level infrastructure, more details on SWN are covered in deliverable D4.4a (Section 18).

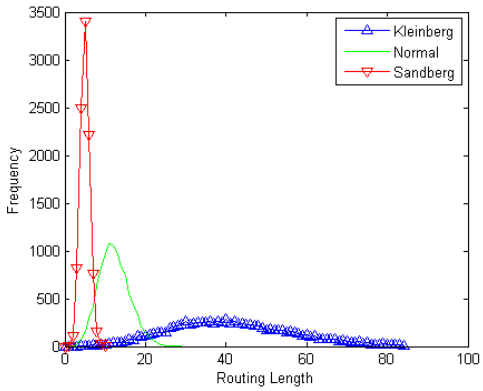


Figure 4.6: Routing Length Distribution

We intend to submit a position paper to the SELFMAN workshop (Decentralized Self Management for GRIDS, P2P, and user communities) at the SASO conference (Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems) on the potential of small world networks as an overlay network to replace more structured networks such as DHTs.

4.5 A Look at Security using Skype

In this section, we take a look at Skype as it is perhaps the largest and most well known P2P system which has extensive security mechanisms built in. The Skype network is a large distributed system with many nodes and users. The main concern of Skype is to make sure that only the Skype application makes use of the Skype network.

Skype achieves this by network traffic encryption and code obfuscation. Obfuscation is feasible since Skype is a closed source application. Network traffic encryption includes RC4 encryption using a key derived from the source address, destination address and message ID.

Many code obfuscation and anti-debugging mechanisms are adopted. For example, code is packed and encrypted in the executable file. Integrity checks are placed in the code to prevent debuggers from modifying the Skype executable. Other anti-debugging mechanisms include timing check and detection of known debuggers. Code obfuscation includes adding indirect calls, conditional jumps, and execution flow rerouting.

Even though Skype uses many techniques to hide the protocol, some parts of the protocol have been discovered by researchers. [15, 21, 17, 67, 74] For

example, it is possible to send a command to a remote Skype node to ping any host. This example shows that basic security mechanisms are useful, in this case, monitoring to understand Skype and integrity checking to ensure that only the unmodified Skype code is executed.

4.6 System-wide Monitoring Infrastructure

WinResMon [113] is a monitoring infrastructure for determining resource usage and interactions among programs in Microsoft Windows environments. Programmers can write system monitoring tools on top of the infrastructure of WinResMon. We enhanced WinResMon for monitoring network activity so that it is possible to explain how network activity (and others) is related to particular programs, processes and threads.[5] We also enhanced WinResMon so that it is possible to attribute activity due to software components. Although WinResMon is implemented for Windows, the general monitoring infrastructure can be applied to other operating systems such as Linux.

A monitoring tool using WinResMon can register a set of interested events. These events are reported to the monitoring tool whenever they occur. An event is characterized by the following information:

- **serial number** is a monotonically increasing integer to notify progress and missing events.
- **start time** and **finish time** are timestamps to show the period of the event. Some events (network, large I/O) take more time, while others (registry, small I/O) are quick. We provide high resolution timestamps based on the internal CPU performance counters. This allows to differentiate events even when they occur with high frequency.
- **process ID** and **thread ID** signifiex which process/thread causes the event.
- **program name** is the pathname of the executable.
- **user name** is the user which owns the process which causes the event.
- **return status** shows whether the event is a success or failure. If it is a failure, e.g. file-not-found, the return status gives the reason.

[5]We remark that we discovered that monitoring networking was much more complex in Microsoft Windows than we expected. This is because networking in Windows is not part of the core operating system kernel.

- **operation** is the resource type and the operation on the resource. E.g. `file_create`, `net_connect`, `net_send_datagram`.
- **resource path** is the pathname of resource. It only applies to file and registry.
- **parameters** are additional information related to each operation. For example, the parameters of `file_create` include file access flags, and file attributes. The parameters of `net_connect` include remote network address. The parameters of `net_send_datagram` include remote network address and packet size. Note that this field is flexible and we can include actual network data if needed.
- **api tracing** can be used to determine which software components call each other to generate an event. The stack trace is analyzed to determine this information.

WinResMon can be used in the composite probes framework described in Section 9.2 as basic probes to collect system information. As this is done at the operating system level, the information is guaranteed to be accurate and cannot easily be subverted. In addition to system-wide information such as network throughput and disk I/O, basic probes may use WinResMon to monitor a specific program or a specific component in a program, because WinResMon provides context information such process ID and component information.

WinResMon has the basic capabilities which can be used as a way of virtualizing some aspects of load testing (described in Section 17.3.2) which relate to the external environment, such as increasing the latency of operations under load and resource failures due to excessive resource usage.

4.6.1 An Example from Monitoring Skype

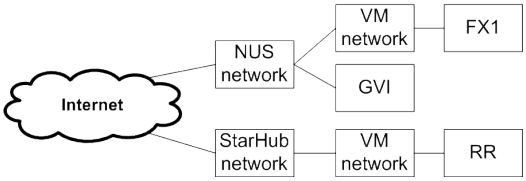


Figure 4.7: Network configuration of three Skype clients: GVI, FX1 and RR.

We use WinResMon to monitor network interaction in Skype. In the experiments, we installed Skype in three machines, *GVI*, *FX1* and *RR*. *GVI*

is directly connected to NUS network. *GVI* has a NUS NATed IP address 172.18.178.175. *FX1* is a VMWare virtual machine running in another machine (different from *GVI*) in NUS network. *FX1* has a VMWare NATed IP address 192.168.0.5. *RR* is a VMWare virtual machine running in a machine connected to the Singapore ISP, StarHub. *RR* has a VMWare NATed IP address 192.168.0.6. Figure 4.7 describes the network configuration.

We did three experiments.

- **idle**

  It is known that Skype will sometimes behave as a supernode unless a registry flag is explicitly set to prevent this. It is also known that Skype participates in routing traffic for other nodes. In our experiment, we wanted to see the traffic flow when Skype was run for an extended period.

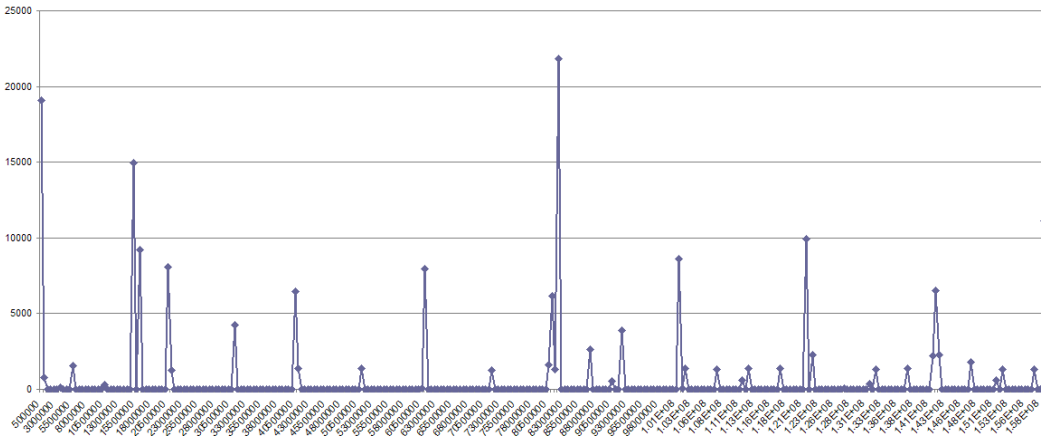


Figure 4.8: Idle Skype network traffic over 16 hours. The X axis is time measured in CPU clocks, and the Y axis is the network traffic.

  We ran Skype in *GVI* and let it idle for a period of 16 hours. The average traffic was 3.1B/s and 30% of the traffic was made to just 2 IP addresses. There were a total of 126 different IP addresses connected to *GVI*. Figure 4.8 shows the network traffic over time.

- **two-way call**

  We had 2 machines (*RR* and *FX1*) in different networks calling each other and simultaneously used WinResMon to monitor them. The idea was to see whether the machines will directly communicate with each other or go through a few hops. *FX1* is the call initiator. We found

that there was no direct connection between both machines. We also found that about 95% of the traffic from both machines were to 2 IP addresses located in Japan. Figure 4.9 shows the network of the two machines over time.

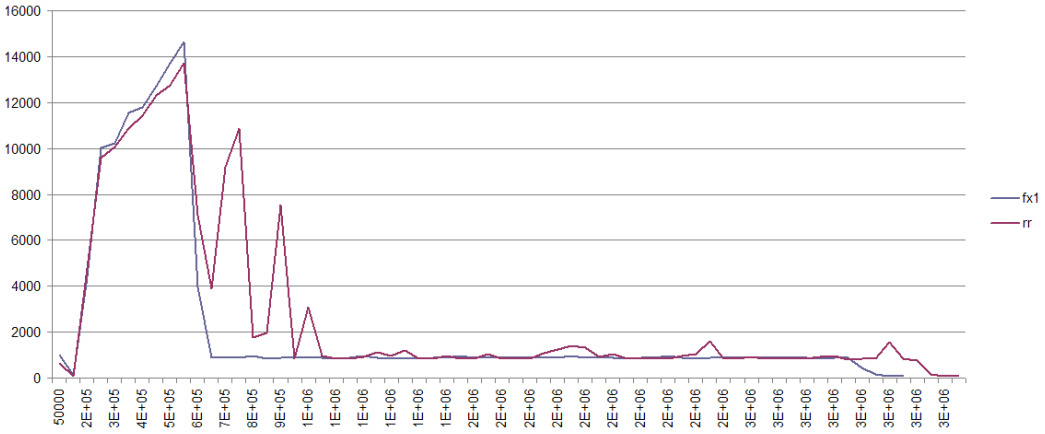


Figure 4.9: Skype network traffic during two party calling. The X axis is time measured in CPU clocks, and the Y axis is the network traffic of the two machines.

- **three-way call**

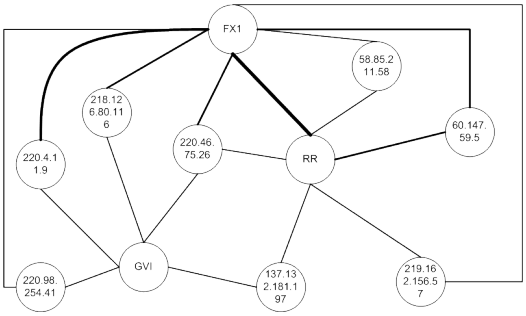


Figure 4.10: Network connectivity graph in a three-way conference call

We have three machines in a three-way conference call. The machine *FX1* is the call initiator. Figure 4.10 shows most of the network traffic. Some minor network traffics such as DNS query are removed from the graph.

Concluding, we show that using WinResMon on a distributed application such as Skype leads to discovering interesting traffic flows.

4.7 Authenticating Software Components and Version Management

In SELFMAN, the self-managing aspect of software components may itself lead to attacks on SELFMAN applications. For example, some researchers used the Storm Worm self-update mechanism to remove Storm Worm itself. (The Storm worm itself uses P2P techniques for self-management and control).

We developed BinAuth [68] which is software component authentication system in Windows. BinAuth ensures that only binaries whose data integrity have been verified can be executed. Software components in Windows include all kinds of binaries which include the main program executables (EXE), dynamic linked libraries, (DLL and others), and kernel drivers (SYS). Because the BinAuth authentication system works at the operating system level, it can guarantee that only software components which pass the authentication test can be used and executed. This stops most kinds of malware attacks which attempt to subvert an application with foreign code. Interestingly, we note that Skype works perfectly fine under BinAuth but it would fail under many systems which use dynamic code instrumentation.

For Selfman, component distribution and updating can be authenticated by BinAuth by signing library files, assuming that each component consists of several library files. We use a flexible signing mechanism which doesn't need to modify the format of existing binaries. The component authentication mechanism also helps to control the versions of software components on a system. Often, the real system relies on many software components, some of which are third party, and others we may simply not even know about. BinAuth allows all the components to be managed so that automatic patching, updating can be achieved without the patching and updating mechanism itself being the attack vector.

4.8 Papers

The papers which describe the work here are as follows:

- Felix Halim, Rajiv Ramnath, Sufatrio, Yongzheng Wu and Roland H.C. Yap. "A Lightweight Binary Authentication System for Windows". In *IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security.* It is attached in Appendix A.7.
- Felix Halim, Yongzheng Wu and Roland H.C. Yap, "Small World Networks as Self Organizing Networks". This is intended to be submit-

ted to *Workshop on Decentralized Self Management for Grids, P2P, and User Communities (SELFMAN)* held in conjunction with *SASO 2008: Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2008.

Chapter 5

D1.4: Java library of SELFMAN structured overlay network

5.1 Executive summary

In this deliverable we present the Java prototype for the SELFMAN structured overlay network. This has been implemented as a set of components, using the Kompics [9] component model, presented in D2.1b (see Chapter 7), and its prototype implementation delivered as D2.1c (see Chapter 8).

We have used the Kompics component framework to implement the Java version of the SELFMAN structured overlay network. We have devised a generic component architecture for a peer-to-peer system that supports multiple virtual peers in one address space. This includes components like: network, timer, virtual peer, bootstrap client and server, monitoring agent and server, failure detector, ring based overlay, web server, web handler, etc.

Our architecture allows for the various components to be written once and then executed both in a simulation scenario or in a real deployment. This is made possible by replacing the network and application components with components for network simulation and user simulation.

The SELFMAN structured overlay network prototype is available as a public release at `http://kompics.sics.se/p2p`.

5.2 Contractors contributing to the Deliverable

KTH(P2) has contributed to this deliverable.

KTH(P2) KTH has been and is still currently implementing and testing components of the SELFMAN structured overlay network using the Kompics component model, presented in D2.1b (see Chapter 7), and its prototype implementation delivered as D2.1c (see Chapter 8).

5.3 The Kompics P2P architecture for the SELFMAN structured overlay network

The latest release of the Kompics P2P architecture, that contains the Java implementation of the SELFMAN structured overlay network, is publicly available at `http://kompics.sics.se/p2p`. The release includes technical documentation, source code, API documentation, the binary library and user guide.

Figure 5.1 shows the Kompics peer-to-peer system architecture. The architecture allows many virtual peers in one address space which makes it possible to execute the system both in a simulation scenario (where all peers live in the same process) or in a real deployment. This is made possible by replacing the network and application components with components for network simulation and user simulation.

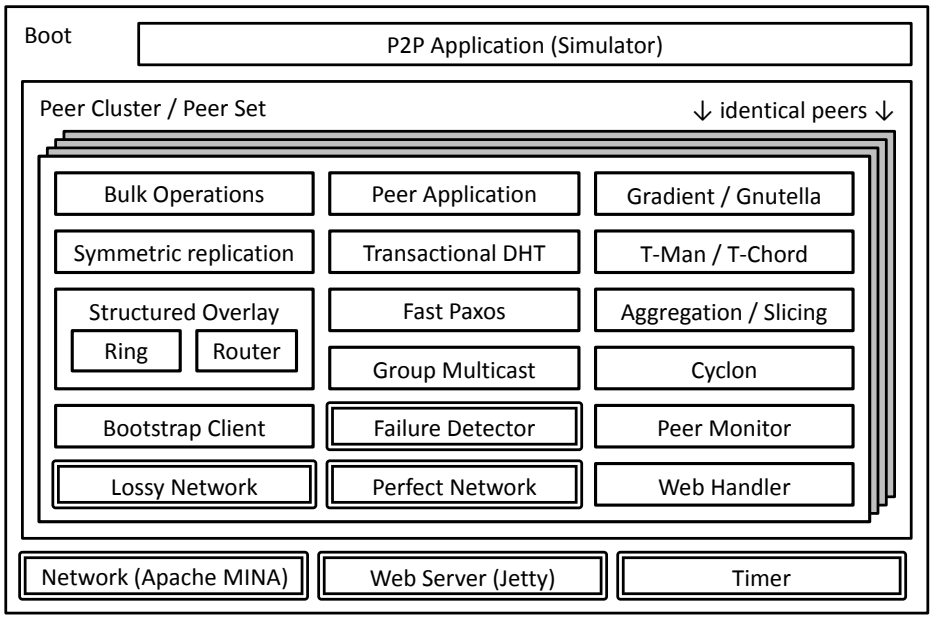


Figure 5.1: Kompics peer-to-peer system architecture.

Chapter 6

D1.5: Mozart library of SELFMAN structured overlay network

6.1 Executive summary

The objective of this deliverable is to implement the conceptual results achieved by other deliverables, as a proof of concepts. This deliverable consists of three softwares: P2PS [44], PEPINO [45] and CiNiSMO [100], all three of them being released as free software under Mozart Public License.

P2PS, Peer-to-Peer System, is a library implemented in Mozart [108] providing a distributed hash table (DHT) on top of a structured overlay network (SON), using the Relaxed-Ring topology [101]. It implements the Application Programming Interface (API) presented in Deliverable D1.2, Chapter 3.

PEPINO, PEer-to-Peer network INspectOr, is an end-user software implemented using the P2PS library. It provides a dynamic visualizer that can inspect a running network, as well as simulating one for research purposes.

CiNiSMO, Concurrent Network Simulator in Mozart-Oz, is a programming framework used for running network simulations with a realistic scenario. This is achieved by running every node in its own lightweight-thread with its own memory scope, as if it where an independent process.

6.2 Contractors contributing to the Deliverable

UCL(P1) and ZIB(P5) has contributed to this deliverable.

UCL(P1) UCL has implemented, documented and tested P2PS, PEPINO and CiNiSMO. It has also contributed by submitting demonstrator proposal that are included as Appendices. Web pages dedicated to make the software available are also developed by UCL.

ZIB(P5) ZIB has contributed by testing PEPINOand helping to understand the transactional algorithms presented in Work Package 3. ZIB has co-author one of the demonstrator proposals, which is dedicated to study decentralized transactions.

6.3 Introduction

This software deliverable is intended as a proof of concepts for the results achieved in other deliverables, in particular, we get a lot of input from deliverables D1.1 and D1.2 (Chapter 3) provide the low-level and high-level self-management primitives for building structured overlay networks. We also get input from Work Package 3, dedicate to provide a distributed storage service, which is integrated in the Application Programming Interface (API) that has been used to implement the software presented here.

The deliverable is composed of a programming library for building Peer-to-Peer systems, called P2PS. An end-user tool for visualizing and inspecting running and simulated networks, called PEPINO. And a programming framework that has been used to evaluate many of the concepts presented in Deliverable D1.2. The programing framework is called CiNiSMO.

6.4 P2PS

This is a library implementing the API described in Deliverable D1.2, see Chapter 3, which provides mean for building a peer-to-peer network using the Relaxed-Ring topology [101]. It is implemented in Mozart [108], and it is meant for programming in Mozart as well.

We have created a dedicate web site for documentation of P2PS, where the library can also be downloaded. The URL is `http://p2ps.info.ucl.ac.be`. We believe that the best way to test what P2PS provides is by using PEPINO, a graphical application for inspecting peer-to-peer networks, which is described in the following section.

As an example of how this library can be used in Mozart, here is an sample code that shows how to create a peer, join a network, put a value in the network and recover it for display. The API corresponds to a Mozart style implementation of what is presented in Deliverable D1.2

```
declare
Peer = {New P2PSNode init} % creates a node
{Peer join(RingRef)} % the ring reference is supposed to be known
{Peer put(key:'hello' value:'world')} % put a value under key 'hello'
{Show {Peer get(key:'hello' value:$)}} % show the recovered value
```

As we have have mentioned above, the best way to test what P2PS can provide is by running PEPINO, which is presented in detail in the next section.

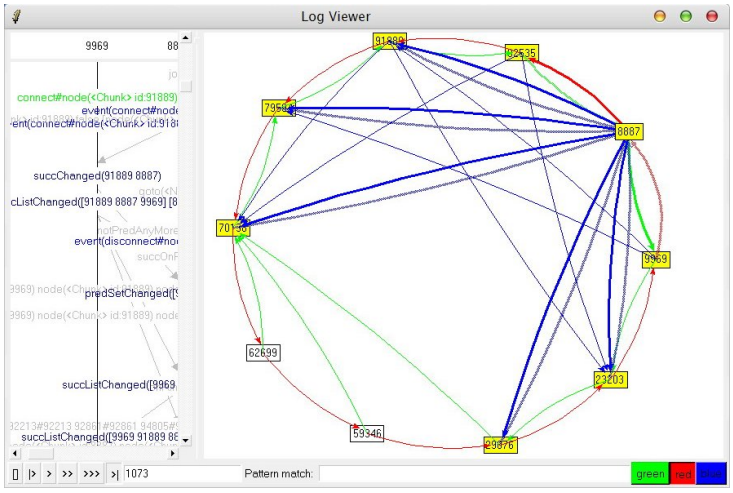


Figure 6.1: A peer-to-peer network visualized by PEPINO

6.5 PEPINO

PEPINO is a graphical PEer-to-Peer network INspectOr running on top of P2PS, a structured overlay network using the Relaxed-Ring topology. PEPINO has been built in order to monitor an existing network or to simulate one in order to study the system. The inspection of the network is done by detecting failures, and by observing the messages sent between peers. A dynamic and self-organizing view of the network is presented to the user, who can interacts with it by injecting failures or by sending messages to arbitrary peers.

Since many systems implement DHT in different ways - in particular by choosing the finger table with a different strategy - PEPINO also helps to study three different strategies, in particular finger tables following the strategy of DKS [57], Tango [29], and Chord [136].

Some screen-shots are presented on this deliverable to depict some of the features of PEPINO. Figure 6.1 shows a ring composed by 10 nodes. Arrows have different colours in order to present meaningful information. Green arrows represent successor pointers. Red ones correspond to predecessors. Blue arrows are fingers. On the bottom-right corner there are 3 buttons to organized the ring according to a particular colour. In the case of the image, predecessor pointers are followed to verified that no inconsistency is presented (correct sharing of responsibility). Fingers and other arrows are highlighted when the mouse is focus on top of a particular node.

One of the main features that differentiate the relaxed-ring from other

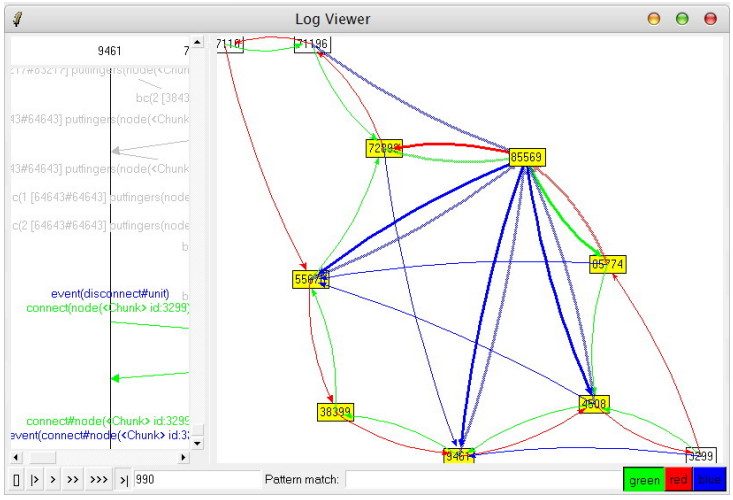


Figure 6.2: Testing relaxed-ring's branches with PEPINO

structured overlay networks, is the ability of accepting nodes with connectivity problems forming branches annexed to the core ring. Figure 6.2 depicts such a branch. The screen-shot shows the network organizing the visualization with respect to the successor pointer (green arrow). It is possible to observe peers painted in yellow as members of the cor ring, and white peers belonging to branches.

PEPINO not only visualizes the network as a ring of peers. It also shows the messages exchanged between nodes as it is depicted on the screen-shot of Figure 6.3. This feature is place at the resizable left side of the application. If the mouse is place over a message exchanged between to peers, all the other messages between them will be highlighted.

A demonstrator of the main features of PEPINO was shown in the Seventh IEEE International Conference on Peer-to-Peer Computing (P2P'07). The abstract published in the proceedings of the conference is presented in Appendix A.8. Two other demonstrator proposal were submitted to the eighth version of the IEEE P2P conference 2008. Both submissions propose an extension to PEPINO, one related to distributed transactions, and the other one related to network partitioning and merging.

The submission included in Appendix A.9 proposes a demonstration of the transactional DHT algorithm based on a modified version of the Paxos consensus algorithm. The whole algorithm for transactions is described in Deliverable D3.1b, Chapter 12, Appendix A.12. The implementation done in P2PS, and used by PEPINO, is completely based on the result of the mentioned deliverable. The design of the transactional DHT is mainly the

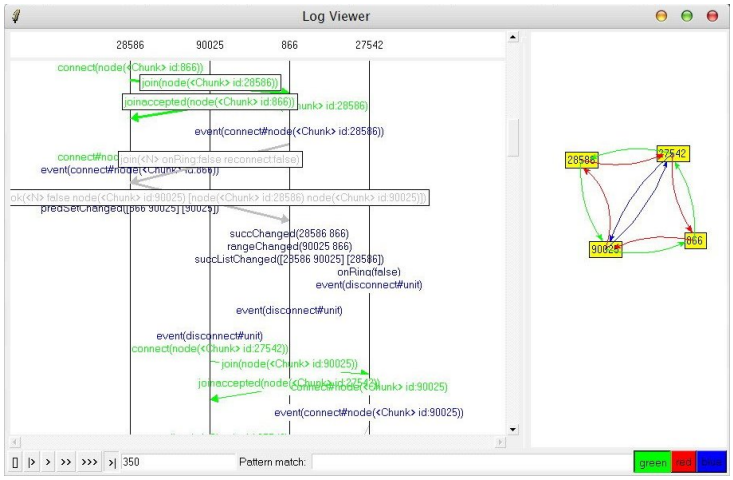


Figure 6.3: Events displayed with PEPINO

result of the collaboration between partners ZIB(P5) and KTH(P2). The demonstrator is the result of the collaboration between partners UCL(P1) and ZIB(P5). Report on the interface of the transaction algorithm is presented in Deliverable D1.2, Chapter 3.

In order to strength the robustness of out algorithm for transactional DHT, the demonstrator compares it to two-phase commit, which is one of the most popular choices for implementing distributed transactions, being used since the 1980s. The use of two-phase commit on peer-to-peer networks is very inefficient because it relies on the survival of the transaction manager, and therefore, it has to be use relying a robust server.

The second submitted proposal, included in Appendix A.10, proposes a demonstration of the merge algorithm [129], which is a result of the first year of SELFMAN. The demonstrator allows the user to inject a network partition to observe how the system survives forming two independent rings. The network partition is simulated because in other case, the PEPINO application would be able to observer only one of the rings. Once the rings are reorganized, the connection between the two set of nodes can be recovered in order to observe the merging algorithm.

6.5.1 Using PEPINO

PEPINO is implemented in Mozart 1.3.2, and it is currently being ported to Mozart 1.4.0. This means that it runs in many platforms, being Linux, Mac OS X and Windows the main ones. The following instructions are meant

for unix based systems with some specifique instructions for Windows users. This commands can usually be executed in a terminal in any of this systems (`cmd` for Windows).

Getting the Software

The software is available on `http://p2ps.info.ucl.ac.be/pepino` on the download section. The user can chose between a compiled version or the source code. The release is open to everyone under Free Software license. It is not internal to SELFMAN.

Requirements

For running PEPINO, you need to install Mozart [108]. Below we specifie some versioning issues. For building PEPINO from the source, GNU Make is needed.

Building and Running PEPINO

Once you uncompress the downloadable file, you will find a directory called `pepino` inside directory `p2ps`. If you get the compiled version, run PEPINO as follows (on Windows, use `pepino.exe` instead of `pepino`):

```
cd p2ps/pepino/
./pepino
```

If you want to compile the sources, then do as follows:

```
cd p2ps
make all
cd pepino
./pepino
```

Running PEPINO like that will open the network inspector with the default values, which is a network simulation of 11 peers. You can try out the different arrows to play the simulation at different speeds, and to move the position of the peers in the network with the mouse. You can also reorganize the network according to different colours of the arrows. Right clicking with the mouse on top of a peer can trigger a temporary or permanent failure to test failure recovery.

PEPINO can be run with different options to study the network from different points of view. We recommend the following options:

To create branches in the Relaxed-ring, run

```
./pepino -b
```

If it does not create enough branches, increment the probability by running for instance

```
./pepino -b --prob=40
```

To create a network of a bigger size, run

```
./pepino -s 42
```

To see all possible options, ask for help as follows

```
./pepino --help
```

That will return the following help menu that can give you an idea about which can of options you can use

```
Usage: ./pepino [option]
Options:
  -b, --branches BOOL  Create randomly branches
      --prob NUM       Probability of having a branch
  -d, --dist ATOM      Determines the type of session [dl, dss, sim (default)]
  -k, --maxkey NUM     Maximun value for a key (default 100000)
  -l, --logfile FILE   Log file name (default test.log)
  -n, --network ATOM   Name of the network (default guinness)
  -o, --ozstore FILE   Ticket to OzStore (default OzStoreTicket)
  -p, --logport FILE   File to store the logger port (default logger.tket)
  -s, --size NUM       Size of the network (default 11, minimun 4)
      --version        Version number
      --viewer         Viewer mode. Just read a log file
  -h, -?, --help       This help
               This PEPINO comes without mayonnaise
```

Looking at the options, you can log the experiment to reproduce it later by running

```
./pepino --viewer --logfile=test.log
```

Running real networks

Instead of running just a simulation using lightweight threads, you can run a real network created with different unix processes. Since the new Mozart version comes with a new distributed implementation, you will have to chose your distribution for running PEPINO accordingly. If you want to use Mozart 1.3.2, you have to run

```
./pepino -dist=dl
```

If you want test PEPINO with Mozart 1.4.0, you run it as follows.

```
./pepino -dist=dss
```

Note that as we mentioned above, PEPINO is currently being ported to Mozart 1.4.0 and this version might be very unstable.

6.6 CiNiSMO

CiNiSMO is a Concurrent Network Simulator implemented in Mozart-Oz. In has been used for evaluating the claims made about the Relaxed-Ring in Deliverable D1.2, and we continue to use it for ongoing research with other network topologies. In CiNiSMO, every node run autonomously on its own lightweight thread. Nodes communicate with each other by message passing using ports. We consider that these properties make the simulator much more realistic. We have released it as a programming framework that can be use to run other tests with other kinds of structured overlay networks. Another motivation for releasing CiNiSMO is to allow other researchers to reproduce the experiments we have run to generate our conclusions.

The general architecture of CiNiSMO is described in Figure 6.4. At the center, we observe the component called "CiNetwork". This one is in charge of creating n peers using the component "Core Node". The core node delegates every message it receives to another component which implements the algorithms of a particular network. Currently, we have implemented in CiNiSMO Chord, P2PS, Fully connected networks and Palta. To add a new kind of network to this simulator it is sufficient to create the correspondent component that handles the messages delegated by the core node.

Every core node transmit information about the messages it receives to a component called "Stats", which can summarize information such as how many lookup messages were generate, or how many crash events were triggered. The component that typically demands this kind of information is the "Test". This is another component that can be implemented to define the size of the network and the kind of event we want to study. Only one CiNetwork is created per every Test. When the

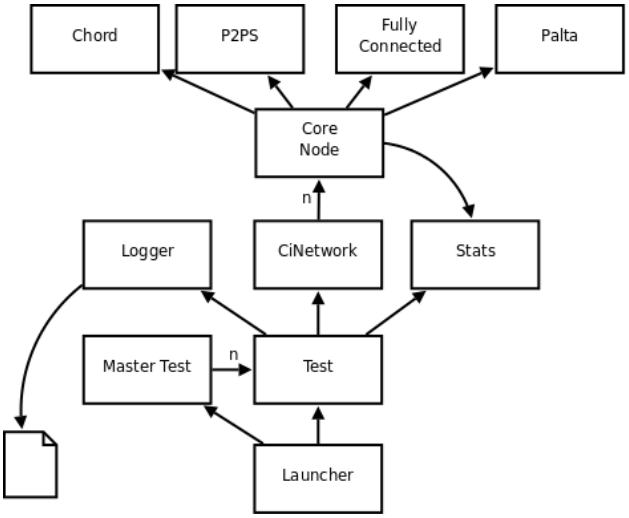


Figure 6.4: Architecture of CiNiSMO

relevant information is gathered, it sent to a "Logger", which outputs the results into a file.

Since it is cumbersome to run every test individually many times, it is possible to implement the component called "Master Test", which can organize the execution of many testing, changing the seed for random generation numbers, or a parameter that is used for the creation of the CiNetwork.

Figure 6.5, which is presented as a result in Deliverable D1.2, can give us an idea of the limits of execution of CiNiSMOThe data is generated by CiNiSMO, but the plot is generated by another specific software called gnuplot. Note that we can run networks of 10000 peers, which means 10000 threads running simultaneously exchanging constantly messages between them. The Y-axis shows as that $1e+07$ messages where created on the most loaded networks.

6.6.1 Using CiNiSMO

The source code and documentation of CiNiSMO is available on its dedicated website `http://p2ps.info.ucl.ac.be/cinismo`. It is released as Free Software. Since it is a programming framework, we provided as source code. To compile it you need Mozart and GNU Make. It can run on Linux, Mac OS X and Windows, among other operative systems. Even when it is meant for programming your own test, here are instructions for building and run some of the tests we have perform to validate our conceptual results. The instructions are given for running CiNiSMO on a unix based system.

Once you uncompress the downloadable files, you must execute the following steps:

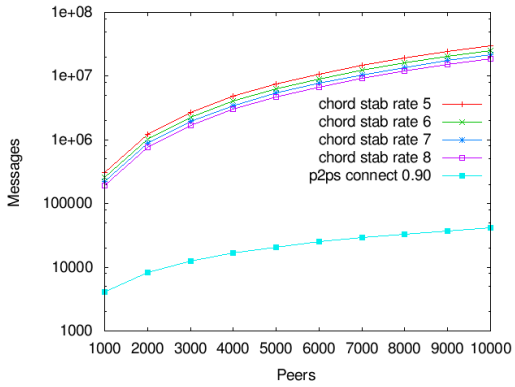


Figure 6.5: Data comparing the network traffic of different instances of Chord and P2PS.

```
cd CiNiSMO
make all
./cinismo --help
```

That will produce the following output, which is a help menu that gives an idea about the different possibilities offered by CiNiSMO.

```
CiNiSMO is the Concurrent Network Simulator in Mozart-Oz
Usage: ./cinismo [option]
Options:
  -k, --maxkey NUM       Maximun value for a key (default 666)
  -l, --logfile FILE     Log file name (default nolog)
      --logger FILE      Logger's Port ticket (default nolog)
  -n, --netsize NUM      Network size (default 7)
  -o, --omega NUM        Omega value for PALTA (default 666)
      --prob NUM         Probability of having a broken connection
  -s, --seed NUM         Seed for random generator (default 1)
      --stabrate NUM     Stabilization rate (default 0)
Choosing tests:
      --mastertest ATOM  Master Test you want to run
      --test ATOM        Test you want to run

      --version          Version number
  -h, -?, --help         This help
CiNiSMO also stands for Cynical Network Simulator in Mozart-Oz,
where 'i' stands for "ignored letter"
```

Then, you can check which kind of tests or master tests are already implemented by running

```
./cinismo list test
```

Which will produce the output

```
List of possible tests you can run

    chord_lookup
    full_connectivity
    p2ps_branches
    p2ps_lookup
    palta_build
    palta_hops
    revereendo_hops
    revereendo_test
```

Let us run the first one, saving the result into a file called `output.log`. Some results will be also printed on the standard output buffer, but we are not interested on that right now.

```
./cinismo --test chord_lookup -l output.log
```

6.7 Publications and Submissions

This section is dedicated to give a brief introduction to the demo proposals we have submitted using the results of this deliverable. The documents are included as appendices.

PEPINO: PEer-to-Peer network INspectOr

This is the first proposal we have submitted with the goal of shown the main features of our network inspector. The demonstrator was accepted and presented in the Seventh IEEE Peer-to-Peer Conference. The article included in Appendix A.8 appears in the proceedings of the conference.

Visualizing Transactional Algorithms for DHTs

The focus of this demonstrator is on the study of algorithms for implementing transactions on peer-to-peer networks. Their visualization contributes to the analysis and test of the protocols, verifying their tolerance to failures. In particular, we compare a DHT running two-phase commit and the Paxos consensus algorithm. The submission has been accepted on the Eighth IEEE Peer-to-Peer Conference, to take place in September of 2008. The article is included in Appendix A.9

Partitioning and Merging the Ring

This demonstrator offers a graphical way to study how tolerant a system is with respect to network partitions, and how efficient is the merging of the network when the network partition disappears. The article, included in Appendix A.10, has been submitted to the Eighth IEEE Peer-to-Peer Conference.

Chapter 7

D2.1b: Report on computation model with self-management primitives

7.1 Executive summary

In SELFMAN we aim to construct long-running, self-manageable and self-configurable dynamic distributed systems. In this deliverable we present aspects of the SELFMAN computation and programming model that facilitates the construction of such complex, dynamic and reactive systems. Systems of this type, where many software modules execute concurrently, reactively and interact in complex ways, are cumbersome to build, maintain and manage without a rigorous architectural and computational model.

The architectural and management aspects of SELFMAN systems are catered for by the guidelines provided by the Fractal [27] component model. In Fractal, the software is organized into components that are reflective, hierarchical and dynamically reconfigurable. However, Fractal is agnostic with respect to the model of execution and interaction between components.

KTH(P2) has worked on Kompics [9], a reactive component model that is compatible with Fractal but provides a concrete execution and interaction model for components, that is particularly aimed at components that implement distributed abstractions. Kompics components are reactive/event-driven, concurrent, and readily exploit multi-core architectures. They are fault-tolerant and can form flexible fault supervision hierarchies. Kompics components provide basic primitives for self-healing and self-configuration.

In this deliverable we first present a recapitulating overview of the Fractal component model in Section 7.3. We introduce the Kompics model for component execution and interaction in Section 7.4. In Section 7.5 we report our ongoing work on integrating Kompics and Fractal.

7.2 Contractors contributing to the Deliverable

KTH(P2), INRIA(P3) and FT R&D(P4) have contributed to this deliverable.

KTH(P2) KTH has contributed by defining and implementing the Kompics reactive component model for building distributed protocols as components. KTH has worked in cooperation with INRIA on integrating the Kompics component model with the Fractal component model.

INRIA(P3), FT R&D(P4) INRIA and FT have worked in cooperation with KTH on integrating the Kompics component model with the Fractal component model.

7.3 An overview of the Fractal component model

Fractal [27] is an advanced component model and associated on-growing programming and management support devised initially by France Telecom and INRIA since 2001. Most developments are framed by the Fractal project inside the ObjectWeb open source middleware consortium[1]. The Fractal project targets the development of a reflective component technology for the construction of highly adaptable and reconfigurable distributed systems.

In this section, we first focus on the concepts that form the component model itself and then we give some elements about Fractal implementations and tools that are relevant for the Selfman project in the line of defining an event-based component model (Kompics) as a "Fractal personality".

7.3.1 Component model

Classical concepts The Fractal component model relies on some classical concepts in the CBSE area.

Components are runtime entities that conforms to the component model. The component model defines specific interaction and composition standards. Components are units of development (design, modeling, implementation, test), deployment and management. Components do exist as such during execution and can be manipulated as such for management purposes. Components do not have predefined granularity (as in EJB for instance where components have a fixed and big granularity): Fractal components may be of arbitrary size, from a pool to a complete DBMS through service, resources, protocols stacks, name servers, application servers... Also, Fractal does not have dedicated "targets" such as typically "technical" or "applicative/business" components.

Interfaces (somehow similar to "ports" in other component models) are the only interaction points between components. Interfaces express dependencies between components in terms of required/client and provided/server interfaces. A client interface can typically emits operation invocations (or signals, or events), while a server interface can receive operation invocations (resp. signals, events).

Bindings are communication channels between component interfaces that can be primitive or composite. Primitive bindings are local communication channels between components (interfaces) that reside in a same address space. Primitive binding are typically implemented as Java references or C pointers. Composite bindings are specialized assemblies of components and bindings dedicated to advanced communication channels such as distributed, secured or transactional communications. From the Fractal point of view, "connectors" or "adaptors" as used in Architecture Description Languages (ADL) are just specialized bindings,

[1] cf. http://fractal.objectweb/org

i.e. bindings with a predefined semantics (communication, type matching, etc.). These specialized semantics can typically be implemented as binding components in Fractal. Structurally, they are not different from other components.

Types Fractal components can be typed. The Fractal specification defines a basic type system in which a component type is defined a set of interface types. Fractal interfaces are not to be confused with language interfaces: a java interface, which defines essentially a list of operations (methods) on a class of objects, is referred to as a interface signature in Fractal - while the term "interface" in Fractal designates the actual runtime entity that can be named and accessed by components. A Fractal interface type defines a interface signature and additional properties (constraints): role (client or server), cardinality (singleton or collection) and contingency (optional or mandatory). A sub-typing relation based on "substitutability" between components is defined by the model.

Factories Instantiation (creation) of components in Fractal can be done using factories. Factories are typically dedicated factory components (i.e. factories are themselves implemented as components which implies of course the need for special bootstrap factories). 3 kinds of factories are defined in the Fractal specification. A generic (or parametric) factory can create components of arbitrary types given as inputs and a description of control and content. A standard factory can create components of one specific type, i.e. the factory is explicitly programmed to do so. Templates can create components that are similar (isomorphic) to themselves. Templates are very useful to instantiate at once complex (hierarchical) component assemblies.

Original concepts Fractal also exhibits more original (in the sense of "less common") concepts.

A component is the composition of a membrane and a content. A membrane exercises an arbitrary control over its content. It embodies the control behavior associated to a particular component:

- it can provide explicit and causally representation of its content (sub-components),
- it can intercept oncoming and outgoing operation invocations targeting or originating from its content and superpose a control behavior: suspending, checkpointing, resuming activities, reifying or changing operation invocations parameters, managing transparently technical services (e.g. persistency or security), managing QoS (memory consumption, garbage collection),
- or it can do nothing at all!

Control is based on reflection. Reflection is defined as the ability of a component (seen as a program) to manipulate as data the entities that represent its execution state during its own execution. This manipulation can take two forms:

- introspection : the ability of a component (seen as a program) to observe and reason about its own execution state ;
- intercession : the ability of a component (seen as a program) to alter its own execution state ; or to alter its interpretation or semantics.

The model is recursive (hierarchical) with sharing at arbitrary levels. The hierarchical structure involves the associated notions of sub-components and super-components and an import/export mechanism based on complementary interfaces used to bind super and sub components. Complementary interfaces is a couple made of an external and an internal interfaces (of the same component) of the same type but with symmetrical role (client/server). Internal interfaces only exist as complementary interfaces (with the basic type system). Only sub components can be bind to internal interfaces of their super component(s). The recursion stops with base components that have an empty content. Base components encapsulate entities in an underlying programming language (e.g. objects in java).

A component can be shared by multiple enclosing components: a component can be a sub component of several super components. The behavior of a shared component C is under the control of the direct enclosing component of C super components. Sharing is intrinsic to resource management: without sharing encapsulation would have to be enforced by applications and/or with complex mechanisms (e.g. replication) in pure hierarchical settings. Sharing may be used for other purposes such as activity (e.g. transactions, processes) management, domains management (e.g. security, faults, administrative domains).

Organization of the model The model specification is organized by "levels of control". The "foundations level" defines base components with no reflexive capabilities (legacy code), an IDL (Interface Description Language) and a naming and binding API (which defines Name, NamingContext, Binder interfaces signatures). The "introspection level" defines the component and interface API and allows for introspection of components boundaries. The "configuration level" provides structural introspection and intercession through the Attribute, Content, Binding, Lifecycle control API. These represent predefined reflexive control of (white-box) components structure but arbitrary control features may be defined. The model also defines the type system and instantiation APIs (factories).

The model is programming language independent and open: everything is optional and extensible[2] in the model, which only defines some "standard" API for controlling bindings between components, the hierarchical structure of a component system or the components life-cycle (creation, start, stop, etc).

Fractal principles The Fractal component model enforces a limited number of very structuring architectural principles. Components are runtime entities con-

[2]This openness leads to the need for conformance levels and conformance test suites so as to compare distinct implementations of the model.

forming to a model and do have to exist at runtime per se for management purposes.

There is a clear separation between interfaces and implementations which allow for transparent modifications of implementations without changing the structure of the system. Bindings are programmatically controllable: bindings/dependencies are not "hidden in code" but systematically externalized so as to be manipulated by (external) programs.

Fractal systems exhibits a recursive structure with composite components that can overlap, which naturally enforces encapsulation and easily models resource sharing. Components exercise arbitrary reflexive control over their content: each component is a management domain of its own.

Altogether, these principles make Fractal systems self-similar (hence the name of the model): architecture is expressed homogeneously at arbitrary level of abstraction in terms of bindings an reflexive containment relationships.

Finally, the model - including type systems, controllers and forms of bindings - Julia platform and some tools such as Fractal ADL are open and extensible - which make us think Fractal is very suitable in the convergence effort with the Kompics model with a foreseeable result of defining Kompics as a event-based Fractal personality.

7.3.2 Fractal ecosystem

We give a partial snapshot of the existing Fractal implementations, languages, tools and component libraries collectively known as the Fractal ecosystem.

Implementations There exist currently 8 implementations (a.k.a. execution platforms)[3] providing support for Fractal components programming in 8 programming languages. We focus here on the Julia platform that is considered in Selfman in the sense that some building blocks of the platform may be used in the implementation of the Kompics component model.

Julia was historically (2002) the first Fractal implementation[4], provided by France Telecom. Since its second version, Julia makes use of AOP-like techniques based on interceptors and controllers built as a composition of mixins. It comes with a library of mixins and interceptors mixed at load time (Julia relies very much on load-time bytecode transformation as the main underlying technique thanks to the ASM Java bytecode Manipulation Framework).

The latest evolutions of the platform, thanks to a joint work between INRIA and France Telecom on the AOKell platform, allows for i) AOP-based (Aspect-Oriented Programming) programming on Fractal membranes based on standard AOP technologies (static weaving with AspectJ and load-time weaving with Spoon)

[3]Julia, AOKell, ProActive and THINK are available in the ObjectWeb code base. FracNet, FractTalk and Flone are available as open source on specific web sites.

[4]And sometimes considered for this reason as "the reference implementation" in Java.

instead of mixins ; and ii) implementation of component-based membranes: Fractal component controllers can themselves be implemented as Fractal components. The design of Julia cared very much for performance: the goal was to prove that component-based systems were not doomed to be inefficient compared to plain Java. Julia allows for intra-components and inter-components optimizations which altogether exhibit very acceptable performance.

Languages and tools A large number of R&D activities are being conducted inside the Fractal community around languages and tools, with the overall ambition to provide a complete environment covering the complete component-based software life cycle covering modeling, design, development, deployment and (self-) management.

A relevant list for Selfman but not exhaustive list of such activities is the following:

- development of formal foundations for the Fractal model, typically by means of calculi, essentially by INRIA Sardes,
- development of basic and higher levels (e.g. transactional) mechanisms for trusted dynamic reconfigurations, by France Telecom, INRIA Sardes and Ecole des Mines de Nantes (EMN),
- support for configuration, development of ADL support and associated tool chain, by INRIA Sardes, Jacquard, France Telecom, ST Micoelectronics,
- support for packaging and deployment, by INRIA Jacquard, Sardes Oasis, IMAG LSR laboratory, ENST Bretagne,
- development of navigation and management tools, by INRIA Jacquard and France Telecom,
- development of architectures that mix components and aspects (AOP), at the component (applicative) level and at the membrane (technical) level, by INRIA, France Telecom, ICS/Charles University Prague,

The most mature among these works are typically incorporated as new modules into the Fractal code base. Examples of such modules relevant for Selfman are the following:

- Fractal RMI is a set of Fractal components that provide a binding factory to create synchronous distributed bindings between Fractal components ("à la Java RMI"). These components are based on a re-engineering process of the Jonathan framework.
- Fractal ADL (Architecture Description Languages) is a language for defining Fractal configurations (components assemblies) and an associated retargetable parsing tool with different back-ends for instantiating these configurations on different implementations (Julia, AOKell, THINK, etc.). Fractal

ADL is a modular (XML modules defined by DTDs) and extensible language to describe components, interfaces, bindings, containment relationships, attributes and types - which is classical for an ADL - but also to describe implementations and especially membrane constructions that are specific to each Fractal implementation, deployment information, behavior and QoS contracts or any other architectural concern. Fractal ADL can be considered as the favorite entry point to Fractal components programming (its offers a much higher level of abstraction than the bare Fractal APIs) that embeds concepts of the Fractal component model[5].

- FScript is a scripting language used to describe architectural reconfigurations of Fractal components. FScript includes a special notation called FPath (loosely inspired by XPath) to query, i.e. navigate and select elements from Fractal architectures (components, interfaces...) according to some properties (e.g. which components are connected to this particular component? how many components are bound to this particular component?). FPath is used inside FScript to select the elements to reconfigure, but can be used by itself as a query language for Fractal.

Component library and real-life usage We would like to emphasize the maturity of the Fractal technology as a whole. Fractal is not a "paper" or "toy" component model: it has been used effectively to build several middleware and operating system components (several are available in ObjectWeb code base) including CLIF, a framework for performance testing, load injection and monitoring (management of blades, probes, injectors, data aggregators, etc.), that may be concerned/used by evaluation campaigns in the last year of Selfman WP5.

Some of these components that embed Fractal technology are used operationally, for instance JOnAS, Speedo and CLIF by France Telecom: JOnAS is widely used by France Telecom[6] for its service platforms, information systems and networks by more than 200 applications including vocal services including VoIP, enterprise web portals, phone directories, clients management, billing management, salesman management, lines and incidents management.

[5]It is worth noticing that Fractal ADL is not (yet) a complete component-oriented language (in the Turing sense), hence the need for execution support in host programming languages a.k.a. "implementations".

[6]See http://jonas.objectweb.org/success.html for a more comprehensive list of operational usage of JOnAS.

7.4 Kompics: Reactive component model for distributed computing

The Kompics component model targets the development of reliable and adaptable long-lived, dynamic, and self-managing distributed systems. Such systems are composed of many software modules which implement various distributed protocols (e.g. failure detectors, reliable group communication, agreement protocols, gossip protocols, etc.) and interact in complex ways.

Kompics aims to facilitate the construction of complex distributed systems by providing a computation model that accommodates their reactive nature and makes their programming as easy as possible. The Kompics run-time system provides primitives for self-configuration, self-healing, and self-tuning of component architectures. Components are executed concurrently and multi-core hardware architectures are exploited with no extra effort.

7.4.1 Component model

In Kompics, distributed abstractions are encapsulated into components that can be composed into hierarchical architectures of composite components. Subcomponents can be safely shared by multiple components at any level in the component hierarchy. Kompics components interact by passing asynchronous data-carrying events and they are decoupled by a flexible event publish-subscribe system.

The concepts of the Kompics model are: components, events, channels, event handlers, event subscriptions, component types, component membranes, component sharing, management and fault isolation.

Components A component is a unit of functionality and management. Components are active entities that interact with each other by triggering (sending) and handling (receiving) events. Components react to events by executing event specific procedures to handle the received events. Components are decoupled (by channels) which makes them independent and reusable.

Every component contains some internal state and a set of event-handling procedures. Composite components also contain subcomponents and thus form a component hierarchy. We sometimes call subcomponents *children* components and the containing composite component *parent* component. We say that the parent component is at a higher level in the component hierarchy than its children components.

Events Events are passive objects that contain a set of immutable attributes. Events are typed and they can form type hierarchies. Components subscribe for events to channels and publish events into channels.

Channels Channels are interaction links between components. They carry events from publisher components to subscriber components. Every channel is parameterized with a set of event types that can be subscribed for or published into the channel. Channels exist in the context of the composite components which create them. However, references to channels can be passed between components through events.

Event handlers An event handler is an event-specific procedure that a component executes as a reaction to a received event. An event handler is a component method that takes as argument one event of a certain type. While being executed, event handlers may trigger new events. Event handlers can be guarded by boolean guards.

Event subscriptions Components subscribe their event handlers to channels by registering event subscriptions at the respective channels. Event subscriptions can be made either by event type or by both event type and event attributes, whereby a subscription contains a set of (attribute, value) attribute filters. Events published into a channel are delivered to all subscriber components which registered at the channel subscriptions which match the published events. Components can publish or subscribe for subtypes of the event types carried by the channel.

Component types A component interacts with its environment (other components) by triggering (output) and handling (input) events. The component is subscribed for the input events to input channels and publishes output events into output channels. Kompics components are parameterized by their input and output channels. The types of input and output events of a component together with the input and output channel parameters that carry them represent the component's type. A composite component expresses its dependencies on subcomponents in terms of the component types of the subcomponent.

Component membranes A component membrane is the runtime incarnation on the component's type. The component membrane is a set of references to the actual channels that the component is using as input and output channel parameters. The membrane maps every pair (event type, in/out direction) to an actual channel reference.

Component sharing A component is shared between multiple composite components essentially by sharing the channels in its membrane. To share one of its subcomponents, a composite component registers the subcomponent's membrane under a name, in a registry of shared components. Other composite components can retrieve the membrane (by name) from the registry and use its channels to communicate with the subcomponent. This registry is hierarchical in the following sense: (1) names registered at some level in the component hierarchy (the level of

the parent component of the shared component) are not visible at higher levels, and (2) names registered at a lower level in the component hierarchy shadow the same names registered at a higher levels.

Component management The management of Kompics components is event-based, i.e., synchronous with the handling of functional events. Every component has an associated built-in control channel. A component manager publishes management events into the control channel and they are handled either by built-in default management handlers or by management handlers programmed explicitly by the component developer. Dynamic reconfiguration operations like adding or removing components and channels, replacing the channels in a component's membrane, replacing components, or changing component subscriptions provide basic primitives for self-configuration.

Component fault isolation Any error or exception that is not caught within an event handler is isolated by the runtime system and wrapped into a fault event which is published into the component's control channel. A supervisor component is subscribed to the faulty component's control channel and handles fault events. As a reaction to fault events, the supervisor component can manage/reconfigure the faulty component. Flexible fault supervision hierarchies [10] (possibly different from the component ownership hierarchy) can be formed. Hence, Kompics provides basic primitives for self-healing.

Figure 7.1 shows an example graphical representation of Kompics components. Here we have two components: A and B, and a channel carrying events of type E_1. Component A has an output formal channel parameter and component B has an input formal channel parameter. Both components use the same actual channel for their formal (input, respectively output) channel parameter. Component B has an event handler that is subscribed to B's input channel and handles events of type E_1. Component A has an event handler that publishes events of type E_1 in A's output channel. Both components A and B and the actual channel exist in the context of a parent component, $Node$.

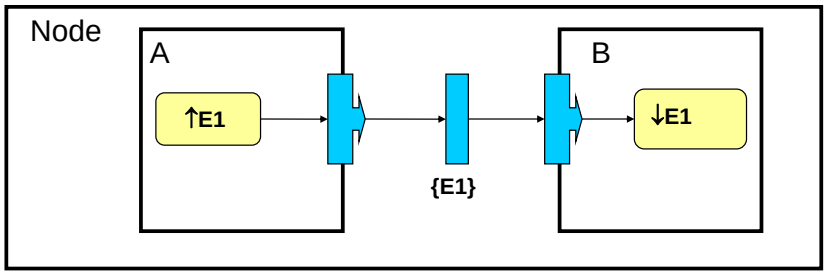


Figure 7.1: Graphical representation of Kompics components.

Figure 7.2 shows an example graphical representation of a Kompics composite component, namely a Best-Effort Broadcast (BEB) [66] component that contains a Perfect Point-to-point Links (PP2P) [66] subcomponent. The BEB component is parameterized by an input channel carrying *BebBroadcast* events and an output channel carrying *BebDeliver* events. The PP2P component is parameterized by an input channel carrying *Pp2pSend* events and an output channel carrying *Pp2pDeliver* events. The BEB component contains two local channels that are used as the actual channels that parameterize the PP2P subcomponent and two event handlers that handle *BebBroadcast* and *Pp2pDeliver* events respectively and trigger *Pp2pSend* and *BebDeliver* events respectively. The arrows indicate subscriptions and publications.

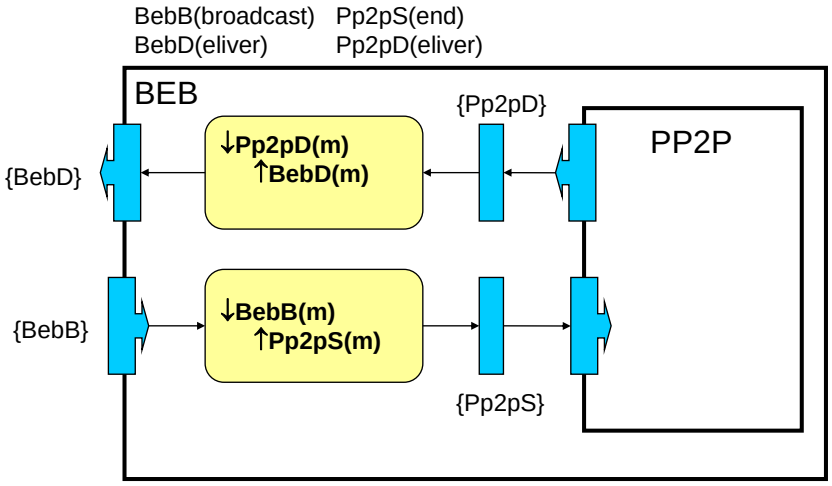


Figure 7.2: Graphical representation of a Kompics composite component.

Figure 7.3 shows an example of two composite components sharing a common subcomponent. A Perfect Point-to-point Links (PP2P) [66] component and a Fair-Loss Point-to-point Links (FLP2P) [66] component share a Network subcomponent. The shared component as well as the channels in its membrane are represented with double borders. The Network component accepts $NetSend$ events (in the sender process) and triggers $NetDeliver$ events (in the receiver process).

$NetSend$ events triggered by PP2P in the sender process are to result in $NetDeliver$ events handled only by PP2P in the receiver process. Similarly, $NetSend$ events triggered by FLP2P in the sender process are to result in $NetDeliver$ events handled only by FLP2P in the receiver process. Filtering $NetDeliver$ events between PP2P and FLP2P is done by event subtyping, i.e., PP2P and FLP2P subscribe to the Network component's output channels for different subtypes of the $NetDeliver$ event type. Events of these subtypes are respectively encapsulated in $NetSend$ events.

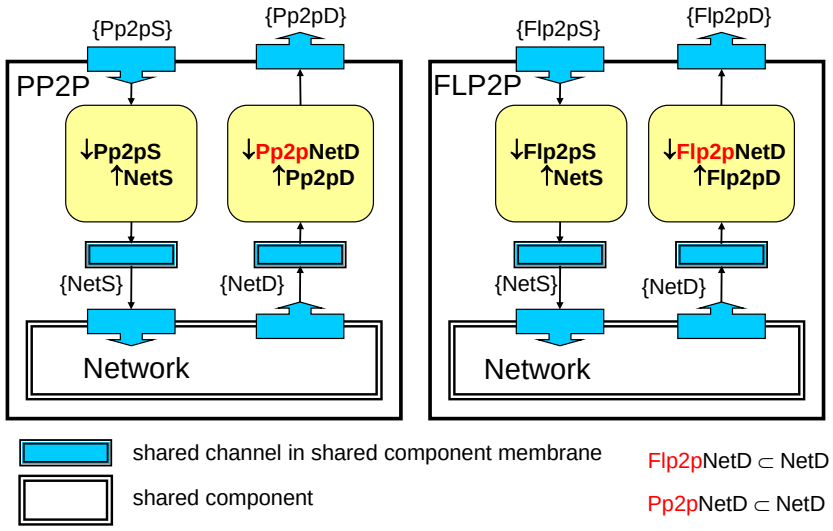


Figure 7.3: Two composite components sharing a subcomponent.

7.4.2 Component execution and interaction semantics

Typical Kompics components do not have execution threads of their own. Their event handlers are executed on their behalf by worker threads from a *worker pool*. Components that have received events are scheduled for execution to one of the worker threads.

Concurrent component execution The event handlers of the same component instance are guaranteed to be executed sequentially, but different component instances can execute event handlers concurrently (or in parallel on multi-core machines). In other words, the event handlers of the same component instance are mutually exclusive, while the event handlers of different component instances are not. However, the execution of an event handler is not atomic (in the all-or-nothing sense). That means that events triggered by one event handler are visible to the corresponding subscriber components (thus, executable) immediately after they are triggered. This entails that the execution of an event handler is neither failure atomic, i.e., it can fail before completion with observable partial side effects (some of the events that were supposed to be triggered by the handler are indeed triggered while others are not).

Event subscription Components subscribe their event handlers to input channels for a particular event type. When component a subscribes an event handler h for events of type T to channel x, a subscription of the form (a, T, h) is registered at channel x. At the same time, a FIFO *work queue*, q_x is created at a and associated with channel x (if it is not already existing from a previous subscription of a to x). A channel y has associated work queues q_y in every component that is

subscribed to it. A component can subscribe more than one of its event handlers to the same channel.

Component scheduling A component a can be in exclusively one of the following three scheduling states: BUSY, READY, or IDLE. We say that a is BUSY if one of the worker threads is being actively executing one of a's event handlers. We say that a is READY if it is not BUSY and at least one of its work queues q_x is not empty, so a is ready to execute some event. We say that a is IDLE if it is not BUSY and all its work queues q_x are empty, so a has no event to execute.

Event publication While executing event handlers, components may publish events into output channels. Assume component a triggers event e of type T in channel x. Let S be the subset of all subscriptions of the form (b, T', h), to channel x, where T' is either T or a super-type of T. For each subscription (b, T', h) in S, a work item of the form (e, h) is enqueued at subscriber b in work queue q_x and if b was IDLE then b becomes READY.

Channel FIFO guarantees The execution model guarantees the following FIFO semantics for channels. Each component a subscribed to a channel x, receives events published in x, in the same order in which they are published. Events triggered sequentially by one component instance will be published in the channel in the order in which they were triggered. A channel serializes the concurrent publication of events into the channel, i.e., events published concurrently in the same channel by different component instances.

This means that all subscribers to channel x for event type T observe the same order of publications of events of type T in their local work queues q_x.

Event handler execution Worker threads execute event handlers on behalf of components. Worker threads atomically pick READY components and make them BUSY. When a worker picks READY component a, it immediately makes a BUSY. An invariant of the execution model is that at this point a has at least one work queue q_x that is not empty. After making a BUSY, the worker dequeues one work item (e, h) from some work queue q_x selected according to some fairness criteria. Thereafter, the worker proceeds to execute a's event handler h by passing it as an argument the event e. Upon completing the execution of h, if all a's work queues q_x are empty, then the worker makes a IDLE. Otherwise it makes a READY.

Worker threads loop Worker threads wait for components to become READY. When a component a becomes READY, a worker w picks it and executes one work item (e, h), the head of some work queue q_x of a. The execution of event handler h may trigger new events e_i of types T_i, published in channels x_i. All components subscribed to channels x_i for event types T_i become READY if they were not BUSY. Upon completing the execution of event handler h, worker w picks another

READY component, if one exists, and it repeats the above steps. If no component is READY, worker w starts waiting for a component to become READY and it repeats the above steps.

Worker pools The number of workers in the worker pool can be proportional to the number of hardware processing cores. Locking is needed only on channels and on the component work queues q_x. No coarse-grained component locking is needed. This enables Kompics component architectures to efficiently exploit multi-core hardware architectures at no extra cost.

Additional worker pools of various sizes can be created and destroyed. Each component is a member of one worker pool at any one time and it shares the workers in the pool with the other member components. A component can be moved from one worker pool to another. Groups of "hot" components can be placed in their own worker pools in order to prioritize their execution. Hence, Kompics provides basic primitives for self-tuning.

Threaded components Typical Kompics event handlers are "short" and do not block. To facilitate programming of protocols in a continuation style, we provide a blocking *receive* primitive that allows an event handler to block waiting for an event with an expected type and/or attribute values. Upon receiving an expected event, the handler continues its execution.

Components that make use of the *receive* primitive use a private thread for executing event handlers which allows them to wait for an event without blocking one of the workers in the worker pool. However, from an observational point of view, threaded components look just like typical components: they accept and trigger events.

Security Components can have references to other (children or parent) components. Also, channel references can be passed between components inside events. References to Kompics components and channels embed fine-grained revocable capabilities (cf. caretaker pattern [116]). For example, component a can give component b a reference to a channel that only allows b to publish events into the channel but not to subscribe for events to the channel. Hence, Kompics provides basic mechanisms for the Principle of Least Authority (POLA) [106], thus it provides some basic primitives for self-protection.

7.4.3 Example component architectures

In this section we present as examples two component architectures that have been implemented with Kompics.

Reliable distributed abstractions architecture The example in Figure 7.4 shows stacking and composition of protocols in Kompics. For example

the Abortable Consensus component makes use of Best-Effort Broadcast and Perfect Links and the Consensus Instance uses the Leader Detector. The example also shows how components can be used to cater for functional and non-functional aspects. The Consensus Instance implements a Paxos uniform consensus algorithm. The Consensus Port offers to an application component a sequence of consensus instances while garbage collecting the already decided instances. The Consensus Service component allocates consensus ports to different applications.

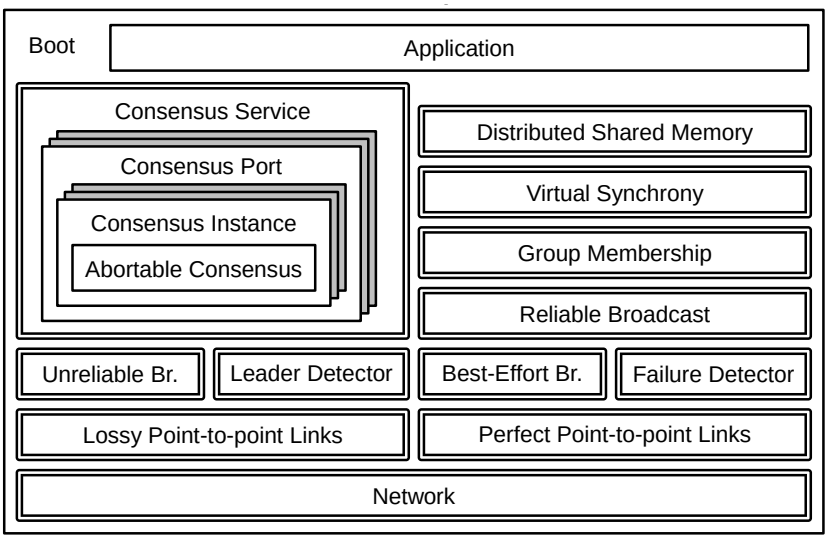


Figure 7.4: Static membership distributed abstractions system architecture.

Peer-to-peer system architecture The example in Figure 7.5 shows a peer-to-peer system architecture that supports multiple virtual peers in one node. This is an example of hierarchical sharing where for example we want to share the Perfect Network abstraction among the protocols of one virtual peer, but have different Perfect Networks in different virtual peers. On the other hand, the Network component is shared and used by all Perfect Network abstractions. In long-running systems like this, where peers build a communication structure among them (the overlay network) we would like to reconfigure peers without restarting them (thus making them forget the structure). This motivates the need for dynamic reconfiguration capabilities in the framework.

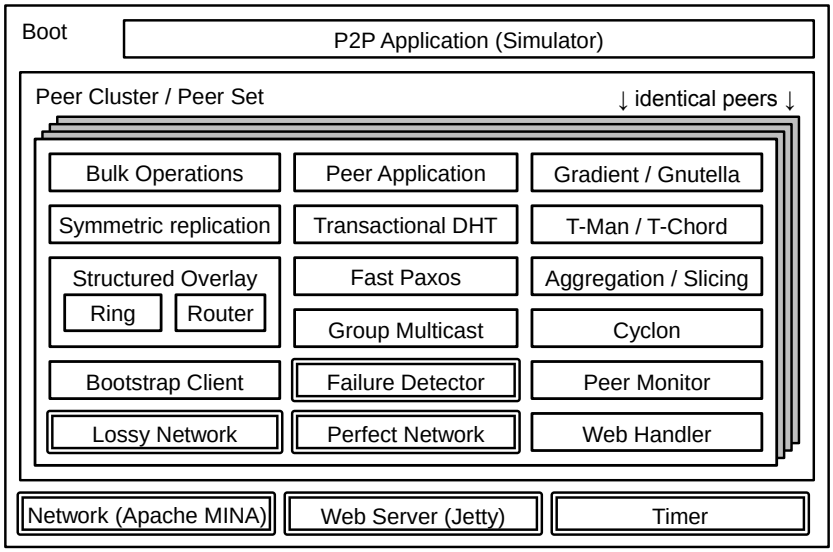


Figure 7.5: Kompics peer-to-peer system architecture.

7.5 Kompics and Fractal integration

The Fractal [27] component model allows the specification of components that are reflective, hierarchical, and dynamically reconfigurable. However, the Fractal model is agnostic with respect to the execution model of components. Kompics [9] is a reactive component model that is similar to Fractal but it enforces a particular execution and component interaction model. Hence, at a high level, Kompics can be regarded as a specialization of Fractal. Here we document the process of "fractalizing" Kompics components, essentially giving Kompics a Fractal personality and making it compatible with Fractal. We present the conceptual mapping between the concepts of the two models and the design choices made for the implementation of their integration.

We start by introducing a toy example of an architecture with 2 composite components that share a primitive component. We use this example as a support for introducing the concepts of the two component models and to discuss the conceptual mapping between these concepts. We assume the reader has some familiarity with the Fractal and Kompics component models.

7.5.1 Example component architecture with sharing

Let us consider the software architecture depicted in Figure 7.6. This is a possible subset of the architecture of a process participating in a distributed system. We have 2 composite components, Leader Elector (LE) and Remote Procedure Call (RPC), that share a primitive component, Failure Detector (FD). The FD component is a subcomponent of both the LE and RPC components.

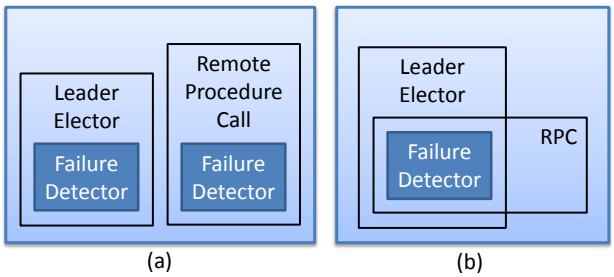


Figure 7.6: Example software architecture. A Leader Elector component and a Remote Procedure Call component share a Failure Detector component: (a) architectural view, (b) sharing view.

The FD component accepts requests to start or stop monitoring the liveness of other processes, and when it detects the crash of a monitored process it triggers a crash notification. We assume both the LE and RPC components use the FD component. The LE component needs to be notified when the elected leader crashes, to initiate a new leader election, hence it requests the FD to monitor the leader process after every election. The RPC component needs to be notified when the remote process to which it issued a procedure call, crashes, so that it can throw an exception for that remote procedure call, hence it requests the monitoring of the remote process for every RPC invocation.

A realistic architecture would also contain a Network component that is a subcomponent of, and shared by all three components. However, we omit it here for simplicity of presentation.

More concretely, the FD component accepts Start and Stop requests and delivers Crash notifications. The LE component delivers NewLeader notifications. The RPC component accepts RemoteCall requests. These calls either return successfully or throw an exception when the remote process crashes during the invocation.

Kompics architecture

The Kompics architecture that corresponds to our abstract example architecture is depicted in Figure 7.7. The FD component is parameterized by 2 channels: a `request` channel and a `notification` channel. The sharing of the FD component between the LE and the RPC components is done by sharing these 2 channels. The `request` channel carries Start and Stop events, and the `notification` channel carries Crash events.

The LE component is parameterized by a `notification` channel. LE has one event handler that is subscribed to the FD `notification` channel and handles Crash events. Upon handling a Crash event, the handler executes a leader election protocol to elect a new leader. Thereafter, it publishes a Start event in

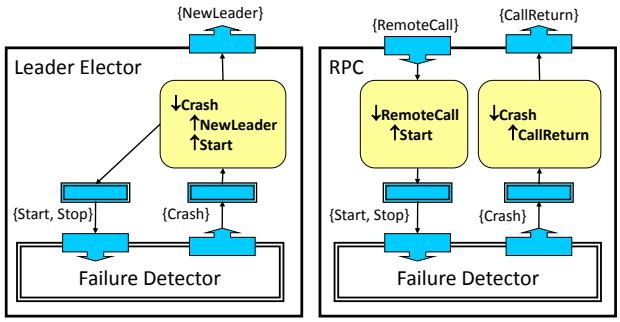


Figure 7.7: Example Kompics architecture.

the FD `request` channel to request the FD to monitor the newly elected leader. (At the same time the handler publishes a NewLeader event in the LE `notification` channel.)

The RPC component is parameterized by an `invoke` channel and a `return` channel. RPC has one event handler that is subscribed to the RPC `invoke` channel and handles RemoteCall events, and one event handler that is subscribed to the FD `notification` channel and handles Crash events. Upon handling a RemoteCall event, the handler publishes a Start event in the FD `request` channel in order to ask the FD to monitor the remote process to which the RemoteCall refers. We omit here the details of RPC including the case where the call returns successfully. In the case when the remote process is detected by the FD to have crashed during a remote call, FD publishes a Crash event in the FD `notification` channel, which is handled by the Crash event handler of RPC. This handler publishes a CallReturn event in the RPC `return` channel. (This CallReturn event would contain a process crashed exception.)

Fractal architecture

The Fractal architecture that corresponds to our abstract example architecture is depicted in Figure 7.8. The FD component is a primitive component shared by the LE/LE and RPC/RPC composite components. (Once again, in a realistic architecture FD, LE, and RPC would be composite components that would encapsulate a Network component but we abstract that for the purpose of this presentation.)

The FD component has a `request` server interface and a `notification` client interface. The `notification` client interface is a collection interface, thus the FD can be bound to more than one component that needs to receive crash notifications. In our example, both LE and RPC have a `crash` server interface, to which the FD's `notification` interface is bound. Both LE and RPC have a `start` client interface which is bound to FD's `request` server interface.

The LE/LE composite component exports LE's `leader` client interface. The RPC/RPC composite component exports RPC's `call` server interface and `return`

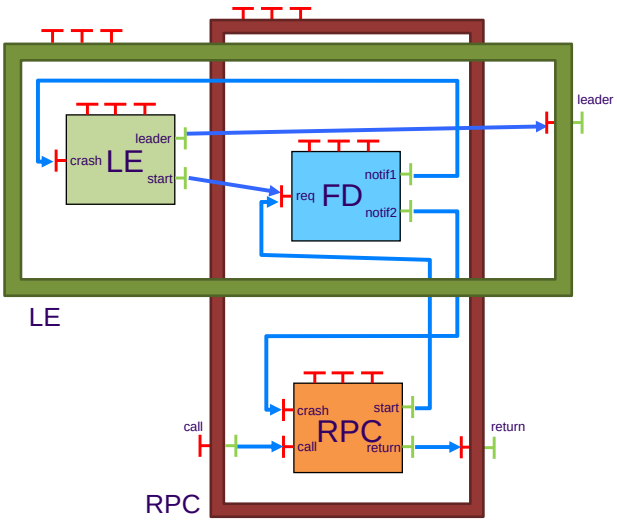


Figure 7.8: Example Fractal architecture.

client interface.

Notice that Kompics components communicate by event-passing. The Fractal equivalent of this type of component interaction is bindings with an asynchronous invocation semantics. Hence, all bindings in Figure 7.8 are asynchronous invocation bindings.

7.5.2 Conceptual mapping of model entities

The Kompics concepts are: *component*, *channel*, *event handler*, the *subscription* of an event handler to a channel and the *publication* of an event to a channel. Notice that our purpose is to give Kompics a Fractal personality, i.e., to map the Kompics concepts to Fractal concepts in order to make it possible for Kompics architectures to be handled by tools that were designed to handle Fractal architectures. Therefore, in the following we are going to look only at the Fractal concepts that are needed to represent Kompics components: *component*, (client/server, internal/external) *interface*, and (primitive) *binding*. For each Kompics concept we are now discussing the Fractal equivalent construction.

Component Fractal components interact though interfaces. Kompics components interact by passing events though channels. Typically, Kompics components are parameterized by the input and output channels through which they interact with their environment, by subscribing for events or publishing events, respectively. In order to reflect this component parameterization at the Fractal architecture level, we represent the input and output channel parameters of a Kompics component as server and client interfaces of the corresponding Fractal component.

There is a direct mapping between a Kompics component and a Fractal component. Every Kompics component, regardless of being primitive or composite, will have a composite Fractal counterpart. The Fractal composite reflects the Kompics component's input/output channel parameters, as server/client interfaces (both external and internal). The Fractal primitive reflects the Kompics component's event handlers as server interfaces. The bindings between the Fractal composite's internal client interfaces and the Fractal primitive's server interfaces reflect Kompics event handlers' subscription to formal input channel parameters. This is motivated by the need to reflect the Kompics component parameterization, on the one hand, and to encapsulate the information about Kompics component's event handlers but at the same time reflect their subscription to channel parameters as Fractal bindings, on the other hand.

We reflect the Kompics component's output channel parameters as client interfaces both of the Fractal primitive and Fractal composite component. We always have a hardwired Fractal binding between the primitive's client interface representing the output channel parameter and the internal interface of the composite that corresponds to the external interface representing the same output channel parameter.

Channel The channel is a concept specific to Kompics. We choose to represent a Kompics channel as a Fractal component for two reasons. First, channels are objects that can be created and destroyed in the context of a Kompics composite component, thus they resemble subcomponents. Also, a subscription to a channel resembles a Fractal binding. Second, channels have a life-cycle management interface similar to that of components.

A Kompics channel is represented at the Fractal architecture level as a primitive component with five Fractal management interfaces and two functional interfaces. The management interfaces are: BindingController, LifeCycleController, SuperController, NamingController and AttributeController. The functional interfaces are: a `publish` server interface, to which client interfaces of components, representing output channel parameters are bound, and a `handle` client interface which is to be bound to server interfaces of components, representing input channel parameters. The `handle` interface is a collection interface so that it is possible to bind it to multiple component input channel server interfaces.

Event handler We chose to represent the event handlers of a Kompics component as server interfaces of the corresponding Fractal primitive. This makes them visible at the Fractal architecture level, and their subscription to input channel parameters is visible as Fractal bindings (from client interfaces internal to the Fractal composite, representing input channel parameters to the server interfaces of the Fractal primitive, representing the event handlers). At the same time the handlers information is encapsulated in the corresponding Fractal composite.

Subscription Event handler subscriptions to input channel parameters are represented as a Fractal bindings from the client interfaces internal to the corresponding Fractal composite (representing the input channel parameters) to the server interfaces of the corresponding Fractal primitive (representing the event handlers).

Publication The publication of an event in a channel is the equivalent of an invocation over a Fractal binding. More concretely, in out mapping, the publication of an event into an output channel (including the delivery of the event to the component that has the same channel as an input parameter), is represented by an invocation on the following Fractal binding path: from the client interface of the Fractal primitive (representing the output channel parameter) to the internal server interface of the Fractal composite (representing the output channel parameter), from the external client interface of the Fractal composite (representing the output channel parameter) to the channel's `publish` server interface, and from the channel's `handle` client interface to the external server interface of the Fractal composite (representing the input channel parameter).

A simple example

Let us now clarify the previous by means of an example. Consider the Kompics primitive component depicted in Figure 7.9. It is a simple server component parameterized by an input and an output formal channel. It has one event handler that is subscribed to the input channel, handles In events and publishes Out events in the output channel. Figure 7.9 also shows the actual channels that are used for the formal channel parameters.

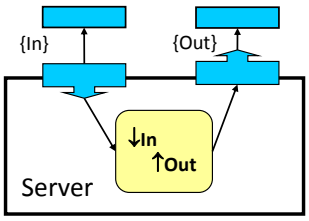


Figure 7.9: Simple primitive component with two channel parameters in Kompics.

In Figure 7.10 you can observe the Fractal counterpart as a composite component. The input channel parameter is represented as an external (internal) server (client) interface and the output channel is represented as an external (internal) client (server) interface of the Fractal composite. The event handler of the Kompics component is represented as a server interface of the Fractal primitive component and its subscription to the formal input channel parameter is represented as a Fractal binding from the internal interface of the Fractal composite to the server interface of the Fractal primitive representing the event handler. We always have

a hardwired Fractal binding from the client interfaces of the Fractal primitive to the internal server interface of the Fractal composite, both representing a Kompics output channel parameter.

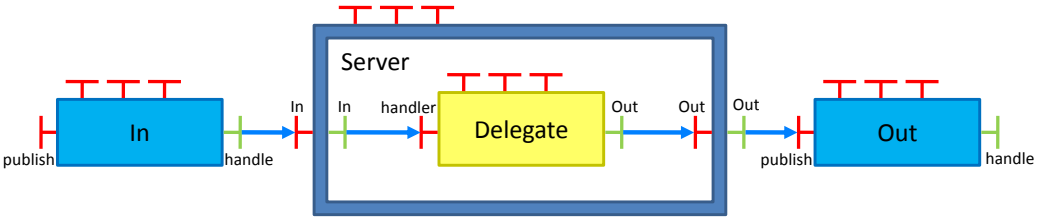


Figure 7.10: Simple Kompics primitive component with two channel parameters in Fractal.

7.5.3 Component sharing example revisited

Let us now look again at the example we introduced in Section 7.5.1. We have two composite components, LE and RPC, sharing the FD component. We have seen the Kompics architecture in Figure 7.7 and the Fractal architecture in Figure 7.8. Figure 7.11 shows the mapped Fractal architecture of the Kompics components according to our conceptual mapping. The details of the FD component are hidden. Notice that the LE and RPC composite components are represented as Fractal composites that contain a `delegate` primitive component whose server interfaces represent the event handlers of the Kompics composite components.

7.5.4 Implementation aspects

Both Fractal and Kompics have implementations in the Java programming language. The next step after the conceptual mapping is to make the Kompics compatible with Fractal at the Java implementation level. This is desired because it results in making a Kompics architecture (in Java) able to be introspected and manipulated by Fractal tools, which normally operate on Fractal architectures though the (Java) API that Fractal components implement.

The goal of our exercise is to give Kompics a Fractal personality, without changing the programming style of Kompics components. This allows giving a Fractal personality to already existing Kompics components preserving backward compatibility.

We had two choices in making the Kompics Java implementation Fractal compatible. One choice was to build a wrapper to the core of a Kompics component which would implement the Fractal Java APIs, and thus make Kompics components able to be handled by Fractal tools. At the same time, these wrappers would form a hierarchy parallel to the actual Kompics component hierarchy. We believe that implementing management operations that change the component hierarchy

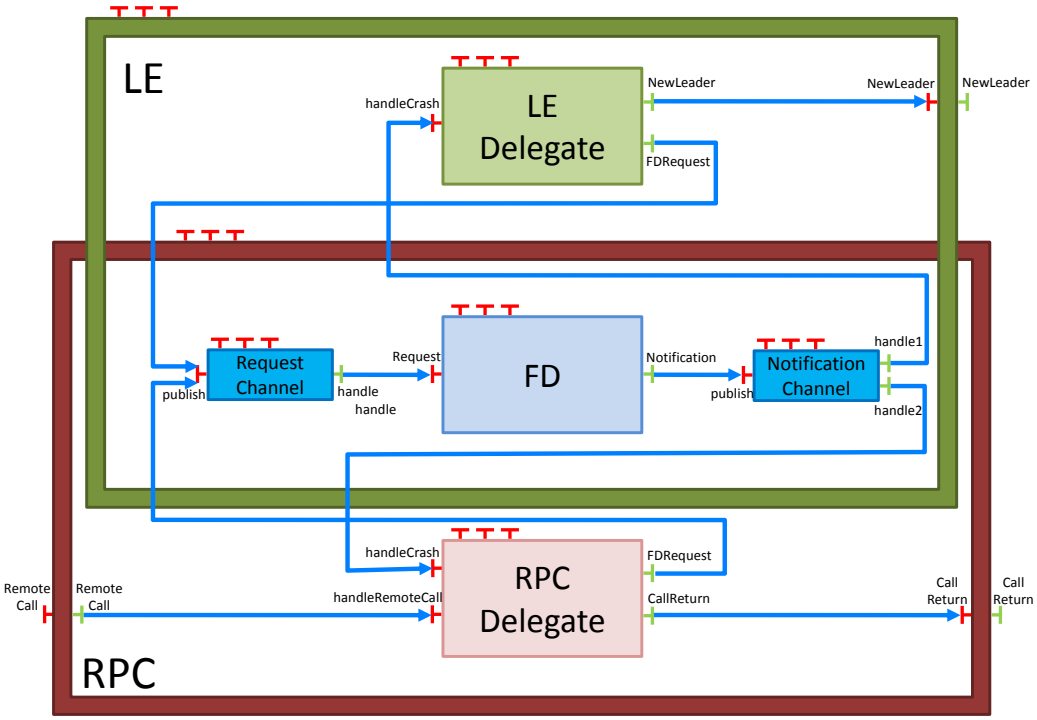


Figure 7.11: Fractalized example Kompics architecture.

would be very difficult since they would have to keep the two parallel hierarchies in sync, while being invoked from either side.

The second choice, was to directly make Kompics components compatible with Julia [4] components, the reference Java implementation for Fractal components. We took this choice because it avoids the difficulty mentioned with the other choice and at the same time, it makes Kompics component inherit some of the Fractal concepts that do not exist yet in Kompics. This means that the Kompics component core and channel core are re-engineered as sets of Julia mixins.

Event-based management

A Kompics component's management interface is event-based, in the form of the component's control channel. Internal component faults are reported to fault supervisors as fault events on the control channel. Also, management commands are sent to the component through the control channel and they are handled by management event handlers, either default or programmed by the component developer.

Because in Fractal management operations are invoked on the management interfaces of a component, our Julia custom (for Kompics) implementation of the various management interfaces consists of publishing corresponding Kompics management events in the Kompics component's control channel.

Chapter 8

D2.1c: Component-based computation model

8.1 Executive summary

In deliverable D2.1b (see Chapter 7) we reported on the SELFMAN component-based architectural and computational model. In this deliverable we present a Java prototype of the SELFMAN component-based computation model, the Kompics [9] framework and run-time system for specifying, composing, and executing distributed protocols as reactive components.

KTH(P2) has worked on Kompics, a reactive component model that is compatible with Fractal [27] but provides a concrete execution and interaction model for components, that is particularly aimed at components that implement distributed abstractions. Kompics components are reactive/event-driven, concurrent, and readily exploit multi-core architectures. They are fault-tolerant and can form flexible fault supervision hierarchies. Kompics components provide basic primitives for self-healing and self-configuration.

We have used the Kompics component framework to implement the Java version of the SELFMAN structured overlay network. This includes components like: network, timer, virtual peer, bootstrap client and server, monitoring agent and server, failure detector, ring based overlay, web server, web handler, etc. We report that work in deliverable D1.4 (see Chapter 5).

An earlier release of the Kompics component framework was also successfully used as a teaching tool in the Advanced Distributed Systems course (ID2203) at KTH. The framework was used as support for student assignments that required the implementation of distributed abstractions as reactive components. Distributed abstractions [66] implemented in Kompics include: perfect failure detector, eventually perfect failure detector, eventual leader elector, best-effort broadcast, reliable broadcast, uniform reliable broadcast, probabilistic broadcast, multiple-writer atomic register, abortable consensus, and Paxos consensus.

8.2 Contractors contributing to the Deliverable

KTH(P2) has contributed to this deliverable.

KTH(P2) KTH has implemented and tested a prototype of the Kompics reactive component model, presented in D2.1b (see Chapter 7), as a Java library.

8.3 The Kompics component framework

The latest release of the Kompics component framework is publicly available at `http://kompics.sics.se`. The release includes technical documentation, source code, API documentation, the binary library and user guide.

THE ADVENTURES OF SELFMAN

KTH KOMPICS

Reactive Component Model for Distributed Computing

Context

Decentralized Dynamic Distributed Systems encompass core distributed protocols like **failure detectors**, **reliable group communication**, **agreement protocols**, etc. These are inherently **reactive**, **concurrent,** and present **complex interactions** which makes them challenging to program and compose in complex hierarchical architectures.

Kompics components

» are **reactive** / event-driven

» are **decoupled** by a flexible **publish-subscribe** system

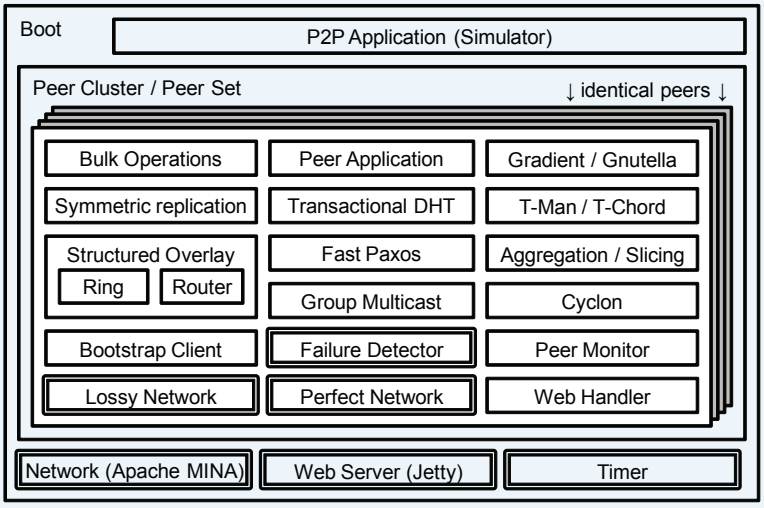
» are **concurrent**
- readily exploit **multi-core** architectures

» can be **composed** out of **encapsulated** subcomponents
- subcomponents can be **shared** between multiple composite components

» form **dynamically reconfigurable** architectures

» are **fault tolerant**
- can form flexible **fault supervision hierarchies**

Peer-to-peer system architecture



Goals

» to make programming of complex distributed systems an easy and painless job

» to enable the implementation of distributed systems in a way that reflects their nature
- concurrent activities
- reactive behavior
- complex interaction

Contribution

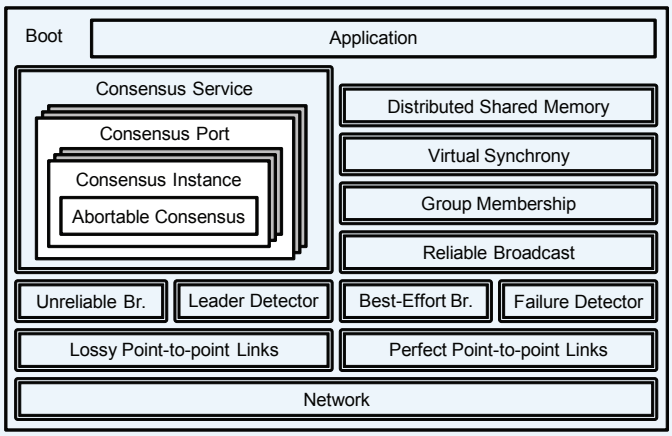
» Kompics component model

» Java implementation

» methodologies and patterns
- composition and sharing
- dynamic reconfiguration

» distributed abstractions component library
- Communication abstractions
- Failure detectors
- Overlay networks
- Reliable group communication
- Gossip based systems

Composition with sharing



Documentation and source at http://kompics.sics.se/

KTH VETENSKAP OCH KONST

Distributed Computer Systems Group
Electronic, Computer and Software Systems
Information and Communication Technology

Contact persons:
Cosmin Arad (cosmin@sics.se)
Seif Haridi (seif@sics.se)

Computer Systems Laboratory SICS

Chapter 9

D2.2b: Report on architectural framework tool support

9.1 Event-Condition-Action Rule-Based Service for Decision Making

Decision making mechanims provides the tools (models, languages and runtime) for implementing the reactive part of autonomic management policies defined by autonomics managers in their control loop. Each management policy is "distributed" across the different functions of the architecture:

- in the monitoring function for extracting the relevant information. A part of the filtering and aggregation task can be done in the monitoring feature.
- in the analysis function for providing the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations. The analyse function evaluates the different conditions that aims to update the global state of the system. This state and its changes are used as inputs in the condition part of the plan's rules.
- in the plan function for providing the mechanisms that construct the actions needed to achieve goals and objectives. The plan function applies the adaptation policy and fires the rules acting on the system. Depending of the complexity of the operation, the number of steps, actions can be organized in a workflow process. In this case, the plan rules throws an action part that creates an instance of a process. The different interactions at each task can be held by others rules instead of human.
- in the execute function by providing the mechanisms that control the execution of a plan.

An approach which has been investigated in Selfman for decision making and more globally for reactive capabilities in component-based systems is based on active rules (Event Condition Action or ECA rules).

Objective Reactive behaviour, the ability to act/react automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. This reactive behaviour is typically incorporated by active rules (Event-Condition-Action or ECA rules), a mechanism widely used in active database systems to provide a reactive behaviour (an elaborated form of triggers as found in most commercial DBMS). Active rules in ADBMS are used for the implementation of integrity constraints, derived data, update propagation, default values, versions and schema evalution management, authorisations, etc.

The approach followed here consists in defining a mechanism for the integration of active rules in component-based systems to augment them with autonomic properties. The fundamental idea being to "extract" the reactive functionality of active database systems, and to "adapt" and "inject" it for component-based systems so as to provide them with autonomic capabilities.

Rational Active rules in database systems have been extensively studied but cannot be directly applied to component-based systems. Three main points deserve a special attention in this respect:

- the definition of a active rule definition model suitable for component-based distributed systems. In active database systems, events, conditions are actions are essentially related to data manipulation through (SQL) query statements - while in component-based autonomic systems, events, conditions and actions are essentially related to interactions between components (operation invocations on component interfaces),

- the definition of a rule execution model suitable for component-based distributed systems. In active database systems, rules are triggered by events generated in the context of a transaction, conditions are evaluated and actions executed in the context of a transaction as well (the three steps in one unique transactions or in concurrent transactions). All dimensions/parameters of rule execution in active database systems are also based on the presence of transactions in ADBMS which represent a natural and convenient execution unit. Transactions are generally absent in autonomic component-based systems. Active rule execution models in ADMBS have to be re-visited for component-based systems.

- the architectural integration of rules in component-based distributed systems. In active database systems, rules are generally represented and manipulated as any other data: typically relations (tables) in relational DBMS

or objects in object-oriented DBMS. Their scope is global to a database schema (a set of relations in relational DBMS, a set of persistent classes in object DBMS). In component-based systems, the nature (e.g. implicit rules implemented as part of a component platform or rules as components) and the scope (a rule attached to one component or rules with broader scopes) of rules have to be stated.

Also important, the extensive study of active rules in database systems has shown that one semantics (specify by an execution model) does not match all applicative needs. On the contrary, what is needed are flexible execution models that allow programmers to adapt the rule execution semantics to their specific needs. A overall objective is then to come up with an adaptable architecture that would support flexible rule execution models.

Reference models and architecture We draw here the big picture of the ECA rules mechanim:

Definition Model The rule definition model specifies the form (format) of events, conditions and actions. Considered events are applicative events generated by operation invocations on components interfaces and access the components attributes, structural events related to changes in the topology of the considered target system (additions, removals, replacements of components and bindings between components) and system events typically generated by the underlying JVM and OS. Applicative and structural events will be typically detected and notified by interceptors. System events will typically come from monitoring system such as JMX, WildCat, Lewys/CLIF, Fractal JMX, etc in the Fractal context [1]. Conditions relate to the states of the considered system known typically by FPath queries on components attributes and system structure (and possibly behavior). Actions range from simple components attributes settings or external notifications (e-mail, SMS) to complex (possibly transactional) reconfigurations (typically expressed with FScript).

Execution Model The basis of the execution model for component-based systems is the "execution unit" delimited by the interval between the reception of an operation invocation on a server interface and the emission of a response onto a client interface. Applicative events (generated by operation invocations) and structural events (add, remove of components and bindings) are thus decomposed into two signals begin and end. Other forms of events (e.g. system events) can be integrated in the model by considering their begin and end signals are merged (i.e. they represent both the same

[1] cf. OW2 open source middleware consortium (http://www.ow2.org/view/Activities/Projects) for information about WildCat, CLIF, Lewys projects.

execution point or point in time). The execution model will also typically define event processing modes (instance-oriented or set-oriented triggering of rules) and coupling modes (execution in a immediate, delayed or differed mode in a same or separate thread of execution).

Architectural Integration The reference architecture of the ECA mechanism is based on the concept of management domain. A domain a set of entities on which is applied a common policy. A domain embodies a unit of composition and a unit of control. The reference architecture is hierarchy of nested domains implemented as components: a policy component encapsulates a set of rules components and provide them with a execution strategy in case of multiple or cascading rules, a rule component encapsulates a event component, a condition component and an action component and provide them with a local execution strategy (event processing mode, coupling modes). Event, condition and action components encapsulate sets of applicative components which embody the scope of event detections, conditions evaluations and actions executions.

Summary The rule service described here is proposed as part the Selfman architectural framework. It proposes an active rule model, i.e. a rule definition model and a rule execution model, that can be coherently integrated into a component model (Fractal in the context of the work); and a graceful architecture for the integration of active rules into component-based systems in which the rules as well as their semantics (execution model, behaviour) are represented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. These foundations form the basis of a framework/toolkit which can be seen as a library of components to construct events, conditions, actions, rules and policies (and their execution sub-components). The framework is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to specific applicative requirements.

9.2 Composite Probes: a Architectural Framework for Hierarchical Monitoring Data Aggregation

Autonomic control loops - as considered in Selfman and more generally in autonomic computing e.g. in the reference architecture by IBM - link autonomic elements and autonomic managers and need a monitoring service in charge of getting data from sensors associated to the Managed Elements, and to make these data available for the decision function (typically an Autonomic Manager).

These data typically describe the dynamic state of a managed element rather than its static constitution (e.g. for a computer, number of processors or memory size). However, changes may occur even to something that would look like a "static constitution". For instance, some advanced computers may have a varying number of processors and memory size. Such changes may be of interest for the autonomic management features, and shall be taken into account by the monitoring service. There are actually two kinds of data:

- plain measures of resource consumption (e.g. CPU time, free memory, database connection pool usage, request queue size in an arbitrary middleware...);
- alarms that notify the occurrence of an event that is not necessarily measurable (e.g. a garbage collector occurrence in a JVM, a node failure, etc.).

The monitoring service relies on components that observe a given resource, namely probes. In the following, we describe the architectural description of these probes in two steps:

- basic probes that provide the monitoring service;
- an extension of the basic probes to introduce probe sharing and aggregation through a composite probe architecture.

It shlould be noted here that the work on basic probes was well initiated prior the Selfman project. The work done in the context of Selfman concerns the development of composite probes. We discuss basic probes here for completeness reason (basic probes has to be known before introducing composite probes).

9.2.1 Probe components

Basically, a probe is a component with an autonomous activity for observing and getting measures from the resource it observes. This activity is controlled by a given lifecycle:

- A probe component is first instanciated in the 'deployed' state.
- It is then typically 'initialized and started, and then possibly 'suspended' and 'resumed'.
- The end of activity is depicted as a pseudo state that actually represents three states:
  - 'aborted' means something did wrong and the probe could not achieve what it was supposed to do (i.e. either its computation or a lifecycle transition request);
  - 'completed' means the probe normally terminated its activity;

– 'stopped' means that the blade did not reach the end of its activity, but simply conformed to the stop lifecycle transition request.

Once the end of activity has been reached, the blade activity may be rerun after an initialization step. Suspend and resume requests may be useful when some faults have been detected or some reconfiguration is under way, in order to avoid getting meaningless measures and possibly bursty generations of alarms. Suspending a probe is also a way to check the disturbance caused by its activity.

We now go into the details of the basic probe component architecture using the Fractal model. This architecture comes from the CLIF load testing framework's so-called blade architecture, hence the frequent use of "blade" in the terminology. The probe component type consists of three mandatory server interfaces (namely DataCollectoAdministration, StorageProxyAdministration and BladeControl) and one mandatory client interface (SupervisorInformation).

Interfaces BladeControl and SupervisorInformation are tightly coupled because most of probe activity control operations (init, start, stop, suspend...) are asynchronous: the call returns as soon as the operation processing starts. Once the operation is terminated, a call-back operation from interface SupervisorInformation is used to inform the supervisor component about the actual probe state. The reason for asynchronous operations in probe activity control is that we consider scalability issues. A typical usage of activity control operations is to simultaneously initialize, start, suspend, etc. a whole set of probes. We could implement asynchrony at the supervisor's side, simply by using parallel threads calling activity control operations and waiting for operation return. But, first, this could introduce a possible high overload on the supervisor if you consider large scale systems (hundreds of probes or more). Second, we still need a call-back operation to give feedback about the probe state at least for states aborted and completed. As a result, we'd rather introduce asynchronous operations and a unified way of providing the supervisor with feedback information about probes states. At last, interface SupervisorInformation provides an operation to notify arbitrary alarm events to the Supervisor. Interface BladeControl offers two extra operations, respectively to consult and modify specific properties. These properties include the activation or deactivation of the memory of the various events (measures, lifecycle, alarms) it generates.

Interface DataCollectorAdministration provides statistical data about the probe - typically about the measures obtained from the resource it observes. These data are represented as an array of integer values. It may look like an arbitrary limitation not to be able to deliver other data types, but it is actually a pragmatic choice that is directly inspired by the LeWYS project. This choice seems particularly relevant to monitor such things like CPU usage percentage, free memory, average throughputs and response times, etc. In a general way, we consider that for other needs than numerical monitoring resource usage, alarms are a good way of notifying probe events holding data of arbitrary type. For instance, a node failure would be typically notified through an alarm. Interface StorageProxyAdministration is

bound to the storage proxy role played by the probe, to enable possible buffering and final collection of probe events. This interface provides methods to possibly allocate a buffer for a new run, and to collect events.

9.2.2 Composite probes

The basic probes described above are primarily designed to be used in a single level, as a flat layer: each probe is managed one by one and monitors one resource. For both scalability and convenience reasons, it appears useful to be able to compose these basic probes into composite probes whose data do not come from a resource observation, but from a set of other probes, whatever they are basic or composite. Here, the idea is to take advantage on the Fractal model's support for component hierarchy and sharing to be able to:

- obtain as many measures as possible from a minimal set of basic probes, with an adaptable level of details and different aggregated values;
- transparently manage a whole hierarchy of probes through a single composite probe.

Let's take as an illustrative example the use case of monitoring the system load of a clustered computing system (details and figures in article in Appendix). For each cluster node, a basic probe is necessary to observe the CPU load and the memory usage. Other system resources could be added to this use case: network bandwidth, disk transfer rate, etc. Now, getting all the measures from all these basic probes, as well as managing all these probes, is quite cumbersome. Conversely, composite probes enable getting a global system load indicator for the cluster obtained through a single probe that transparently handles control operations for the underlying sub-probes. Then, the global system load probe may be based on individual system load probes that aggregate measures from basic probes (CPU, memory). Finally, component/probe sharing enables an arbitrary number of different aggregations, such as the global cluster CPU load indicator provided by the clusterCPU composite probe.

9.3 MyP2PWorld: The Case for Application-level Network Emulation of P2P Systems

9.3.1 Introduction

Reflecting on previous research in Peer-To-Peer systems, one can find that the majority of the research work passes through a number of typical stages where every stage has its associated tools for reasoning and evaluation. In the algorithm design stage, formal/semi-formal reasoning is used to prove aspects like liveness

and safety properties. In the second stage, where a system design is outlined, the goal is to understand the effect of different parameters on the performance of the system. Simulation is intensively used and probably is the most dominant tool, and coudl be found in probably every single paper suggesting a new system. At this stage, analytical modeling is also a frequently used tool. Examples include fluid models for BitTorrent[158] and Chord[89]. At the final stage, where the system is implemented, global scale testbeds like PlanetLab is used as well as emulators like ModelNet[140] or NCTUns[152].

Stage	Tools
1. Algorithm Design	Formal/Semi-Fromal Reasoning
2. Performance Analysis	Numerical Simulation, Analytical Modelling
3. Implementation	Testbeds (e.g. Planetlab), Emulation (ModelNet)

Table 9.1: Summary of the tools needed for reasoning, evalution or testing in the different stages of designing large-scale distributed systems.

In this work, we report our experience with P2P systems testing while developing a P2P solution for streaming live video events at Peerialism Inc.[2]. We mainly argue that at late development stages, i.e the implementation stage, testbeds and emulators are not sufficient for testing/evaluation needs. We describe a tool that we developed entitled "MyP2Pworld". In the next section, we elaborate on why we needed to come up with yet another tool for testing. Afterwards, we show the architecture of MyP2PWorld and how it was used in our projects and finally we discuss it current limitations and our future plans for it.

9.3.2 Motivation

We faced a number of problems: First, the discrepancy between the simulated protocols and their implementation in the production code. The case in an industrial context also is slightly complicated by the fact that the people who design and simulate the protocol (Researchers), are different from those who deliver the production-quality software (Developers). The main issue, while scientifically unprovable but anecdotally evident, is that when one designs a protocols and specifies it for others to implement it, some intuitive or based-on-trial/error design decisions are implicit. When given to another person the question of "Why do not we do it the other way?" always becomes an issue and there is no fast way to answer that, especially when it comes to non-obvious second order effects.

The second issue was debugging and reproducibility. We have used PlanetLab extensively and it has been very useful. Our main problem was debugging and reproducibility. We started by implementing an environment for collecting log files from all nodes which helped but we either faced the case where we could not reproduce it again or where the logs were not sufficiently verbose to show the problem.

The third issue wast the testing environment. With PlanetLab, we had the practical issue that the environment was not handy to everybody at all times and there was the coordination overhead of sharing slices. Modelnet and NetUNCS retain also this property because of the many customization on OS level that need to be done to start using them. In fact, on PlanetLab things are much simpler because of the central administration.

Therefore, the needed requirements for a testing environment were:

- Testing is done on the production code base.
- Easy to deploy on every development and testing machine.
- Provides total reproducibility.
- Can be used for automated integration testing not only unit testing. We mean by that automated testing of particular sections of the protocol implementations in contrast to non-reproducible complete-system runs in testbeds of emulators.
- Allows debugging using a debugger and not only by depending on log files.

We mainly tried to achieve all of the above sacrificing one main feature which is that we have to modify the real code to make it capable of interchangeably running in emulation and real modes. Nevertheless, we tried to realize that with maximal transparency whenever possible.

Desirable Property	Simulation	TestBed	Emulation	MyP2PWorld
Production Code Base	No	Yes	Yes	Yes
Ease of deployment	High	Medium	Low	High
Reproducibility	Yes	No	No	Yes
Automated testing	N/A	No	No	Yes
Mofidied App. Code	N/A	No	No	Yes

Table 9.2: Comparison of MyP2PWorld against other testing tools.

9.3.3 What MyP2PWorld is Not

We have to explicitly state that the point of MyP2PWorld is not to replace other tools, but rather to complement them by providing an additional tool in the toolbox of P2P systems testing. It is another point in the design space of the tools that aims to retaining the reproducibility property of the simulators while working on the production code like the testeds and the emulators.

9.3.4 Related Work

The work that is most similar to our work is EmuSockets[12] in the sense that it advocates application-level emulation of the network, and that it specifies a congestion model for TVP links. However we complement the above with local concurrency emulation to achieve exact reproducibility, a property not attainable by EmuSockets. [12] also references a number of application-level emulators, none of which shares with us the focus on reproducibility.

9.3.5 System Architecture Overview

MyP2PWorld is organized into four layers:

Discrete-Event Simulation (DES) Layer: Provides simulation time and network model and is not visible to the real application.

Emulation Layer: Provides to the real application and interface that looks like real network/OS services however that get routed to the DES instead of being routed the corresponding network/OS services.

Real Application Under Test: Multiple instances of the real application that got modified to use the glue layer.

Scenario Management Layer The main execution entry point. Responsible for taking as input a scenario fie and configures all layers such as forking and killing instances of the peers at specified times, configure network behavior etc.

9.3.6 Discrete-Event Simulation (DES) Layer

This layer could be (and in fact has been) used on its own as a traditional simulator. Every simulated node has a single identifier and has access to a timer abstraction where it can schedule things in the future, e.g. to timeout on an event or perform a periodic activity. For the network model, we provide random delays between nodes and we do not currently model a physical network topology. For bandwidth we have support for reliable, FIFO links where the transmission rate of the data is always the maximum possible at a given link. Our work has mainly been inspired by BitTorrent simulators such as [20] and [156], however we have worked on providing a well specified model with an efficient implementation. While we won't delve into the details of the DES we will describe our bandwidth model in more detail:

Bandwidth Model

We assume that, given a sender S and a receiver R, the sender sends blocks of data that are substantially bigger than an IP packet. Once the sender starts sending a block, the network should try to send the block at the maximum possible speed

between the two parties. While the piece is in transit we say that S and are R have an ongoing "transfer" Naturally, the transfer of a certain block is affected by other transfers taking place between S or R and any other third party. The main quantities needed for the description of the model are :
β_S sender's maximum bandwidth
β_R receiver's maximum bandwidth
α_S sender's available (free) bandwidth
α_R receivers's available (free) bandwidth
τ_S set of sender's ongoing transfers
τ_R set of receiver's ongoing transfers
We sometimes use the above symbols without specifying sender/receiver side to mean that the argument is interchangeably used for either side.

Bandwidth allocation Each time a block is sent, i.e. a transfer is started, the amount of bandwidth that is given to the new transfer is equal to:

$$t = min\left(max\left(\alpha_S, \frac{\beta_S}{|\tau_S|+1}\right), max\left(\alpha_R, \frac{\beta_R}{|\tau_R|+1}\right)\right) \tag{9.1}$$

If $\alpha > t$, then a bandwidth of α is reserved for the new connection and the algorithm halts. Otherwise $\pi = t - \alpha$ amount will be deducted collectively from the transfers in τ according to some rules. See next subsection.

Deduction algorithm

A transfer gets a deduction only if it is using more than its fair share (its bandwidth $> fs$), where fs is the fair-share $= \frac{\beta}{|\tau|+1}$. Note that $fs = t$ for at least one side, but might not be true for the other side ($t \leq fs$). Let τ' be the set of transfers with bandwidth $> fs$. We never cut a transfer more than its fair share. We cut from the all transfers in τ' by π in total for each according to its bandwidth.

Deduction distribution For transfer i let X_i be that transfers bandwidth, $\pi_i = X_i - fs$. Let X_i' is the new bandwidth after deduction.

$$X_i' = X_i - \pi p(i)$$

where $p(i) = \frac{\pi_i}{\Sigma_i \pi_i}$

Note that $\Sigma_i \pi_i \gneqq \pi$ when $t > \alpha$. If $t = \alpha$ then $\Sigma_i \pi_i = \pi$ but in that case we don't need to deduct.

After allocation, other nodes might have unused bandwidth due to the deduction process, in that case these connections are boosted. See bandwidth deallocation.

perfromDeduction (amountToReclaim) **begin**
  $\tau' \leftarrow$ t $\in \tau$ such that t.bandwidth $> \frac{\beta}{|\tau|+1}$;
  $\mu \leftarrow \Sigma_{t \in \tau'}$ t.bandwidth ;
  for t $\in \tau'$ **do**
    t.bandwidth -= t.bandwidth / μ * amountToReclaim ;
  end
end

Algorithm 1: Bandwidth: Allocation Deduction

Bandwidth deallocation

Like the bandwidth allocation, each connection gets additional bandwidth from the free bandwidth proportional to its bandwidth. We define 'loose' transfer as the transfer that both nodes at both ends has available bandwidth. And we define its loose value as the minimum of both.

boostTransfers () **begin**
  $\tau' \leftarrow$ t $\in \tau$ such that t.loose = true ;
  $\mu \leftarrow \Sigma_{t \in \tau'}$ t.bandwidth ;
  for t $\in \tau'$ **do**
    t.bandwidth += t.bandwidth / μ * t.getLooseValue() ;
  end
end

Algorithm 2: Bandwidth: Deallocation Boost

This process can take some iterations to converge totally where all bandwidth is utilized, however, frequently bandwidth fragementation occurs. However, accepting a threshold of 2% of untilized bandwitdh usually results in quick convergence.

9.3.7 Emulation Layer

This layer is actually the core layer of MyP2PWorld. We can say that it provides three core functionalities:

Network Services

The point of this part is to make all network communication code exactly the same whether the system is running in real mode or in emulation mode. In the real application, we have been using Apache MINA framework [52]. A modular framework on top of java non-blocking I/O libraries, that has many advantages

such as filter chains and decoupling of marshaling formats from communication logic among other things.

Listing 9.1: MINA TCP server with minimal changes that enable switching between real and emulated modes with minimal code changes

```
1  import org.apache.mina.common.IoAcceptor;
2  import ~~org.apache.mina.transport.socket.nio.SocketAcceptor~~ com.peerialism.simpipe.SimPipeAcceptor;
3  import java.net.SocketAddress;
4  ....
5  SocketAddress serverAddress = new SocketAddress(''localhost'', 1234);
6  IoAcceptor acceptor = ~~SocketAcceptor~~ SimPipeAcceptor();
7  acceptor.bind(serverAddress, new IoHandlerAdapter(){
8      public void messageReceived(IoSession session, Object message){...}
9      public void messageSent(IoSession session, Object message){...}
10     public void sessionClosed(IoSession session){...}
11     public void sessionCreated(IoSession session) {...}
12     public void sessionIdle(IoSession session, IdleStatus status){...}
13       ...
14     });
```

Time & Concurrency Services

Given that exact reproducibility is one of our main goals, we have to make sure that we have total control over how concurrent events get scheduled. For a certain experiment, we want to run it many times, while having exactly the same sequence of events happening exactly the same way every single time. Concurrency occurs on two levels. The first level is between different nodes that are running concurrently. In real mode, naturally, nodes are either on different machines or, if it is required, they can be run in separate OS processes on the same machine. In an emulation with the level or reproducibility that we target, having a separate process per node will violate reproducibility. Therefore, we run all nodes in one OS process. The consequences of that are discussed in the next section. The second level of concurrency is between different threads that are running inside one node. Typically nodes, in real mode, use multiple threads for network/local I/O, timeouts, periodic activities, etc. However, in emulated mode, we can not possibly keep these multiple threads that get schedule by the OS in a non-reproducible fashion. Therefore we also had to find a way of running all these activities in one thread which we are explaining in this section.

In general, the DES layer gives us concurrency by means of events being scheduled at discrete instances on the simulated time scale. Many events can happen in one such instance either on different/same application nodes. So to get rid of the threads, our solution was to make sure that the application architecture avoided blocking threads and instead depended on an event-based architecture to provide concurrency. Thus, event scheduling and handling is managed by a thread pool in real mode, and by the DES scheduler in emulated mode.

In our case, Apache MINA already provided non-blocking I/O and and an event-based model, so we wrote emulation hooks to let the events be scheduled by the DES layer but code changes at the application layer are minimal changes as we explained in the previous section. However, all periodic activities and time-outs were based on blocking threads. Therefore, we had to refactor the code to make it

event-based using Java scheduled future. The end result was that we found a way of writing code for scheduling concurrent activities where real mode and emulated mode code are minimally different.

It is also important to mention here, that in real mode, threads use real time for specifying timeouts and periodic events frequencies. In emulated mode, we had to also override the library calls for getting the system time. We configured the DES layer such that the smallest unit of simulated time models one millisecond.

```
1
2  import ~~java.util.concurrent.ScheduledFuture~~ com.peerialism.ScheduledFuture;
3  import ~~java.util.concurrent.ScheduledThreadPoolExecutor~~ com.peerialism.ScheduledExecutor;
4  ...
5  class SomePeriodicAction implements Runnable {
6      public void run() {
7          // Action
8      }
9  }
10 ....
11 SomePeriodicAction action;
12 SchedulingExecutor executor = new SchedulingExecutor();
13 ScheduledFuture future = executor.scheduleAtFixedRate(action, 2000, TimeUnit.MILLISECONDS);
```

Context Services

The third set of changes that made our approach feasible was the management of many nodes inside one OS process, namely one Java virtual machine. The main problem is that our application (like most other P2P applications) was not designed for many nodes to run in the same OS process. Global data structures like singletons and loggers are examples of major issues in this category. For that, we had to introduce to the DES layer the concept of a "context", i.e. when a node is created, it has to request from the DES layer the creation of a context labeled by a unique id of the node. When the time comes for an event to be fired the scheduler switches to the context of the executing node and we expose to the application the service of querying the emulation layer about the current context. Using the context services, singleton and loggers of all nodes were able to coexist in the same OS as described below.

A singleton, in real mode, stores one instance of an object. In emulated mode, singletons were made to store sets of objects indexed by context ids, every time a singleton is requested to return an instance, it calls the scheduler to know in which context it is running and returns the corresponding instance. This is one place where we really could not find a direct transparent way that would make the real and emulated mode code look identical. However, we have plans to improve that in the next major release using a component framework.

For logging, we were using the slf4j[3] package whose purpose is to make an application log using a simple interface and then the package is configured to bind all logging to a real logging package. We used this feature to implement a context-aware binding for slf4j. Therefore, that was a totally transparent change from an application point of view.

Other minor issues like port numbers, file locations, etc. were solved using configuration parameters.

Misc. Performance Issues

External Web Services A minor issue was that each node in real mode depended on an external web service (called "publisher") that acted as a library for content-specific meta-information. To avoid having any "real" networking while in emulated mode, we cached all the info we needed from the publisher transparently provided a fake publisher that loads meta-information from the local cache while in emulation mode.

Byte Buffers All communication between nodes is actually accomplished by cloning byte buffers that contain the messages in transit. Creating lots of byte buffers and garbage-collecting them was a slowing the performance and the peak memory in a negative way. Instead, we allocated pools of byte buffers that we reused.

Multiple CPU Cores To make use of multi-core processors without violating our reproducibility constraints, we made it possible for events that run in the same discrete time step on different machines to run in multiple OS threads. The change needed for that was to provide a deterministically-seeded random number generators per node instead of per simulation.

9.3.8 Scenario Management Layer

This is the top layer that binds everything together, it loads: *i*) Configuration files containing DES configuration parameters, application parameters, etc. *ii*) Scenario files containing a particular setup of joins failure *iii*) Binaries for the different types of nodes like in our case a tracker, a source and many clients.

9.3.9 Conclusion & Future Work

We presented a new tool for testing Peer-to-Peer systems at the implementation stage. We were mainly motivated by the lack of a testing tool that can test the production code base while providing exact reproducibility. The main difference between this tool and other application level emulators is that we do not only simulate the network layer, but we also handle local concurrency. i.e. despite running many nodes on the same machine, one single OS thread is used to exectue all nodes and their threads. The tool has been used in production environments to fix a substantial number of bugs, that were extremely hard to catch on a testbed like PlanetLab.

The current status of MyP2PWorld is that we have actually started to realize how to make application-level emulation work. However, there are a number of things that we need to do before we can have the tool totally generic to be used in other applictions. The first future task would be to reengineer the switching between real and emulation mode using a component framework to be able to

make the emulation hooks as transparent as possible. The second task would be to have a UDP bandwidth model like our TCP model.

Chapter 10

D2.2c: Architectural framework – Components & Navigation

10.1 Executive summary

This deliverable defines the API of the Oz library that forms the core of the component and navigation framework detailed in Deliverable D4.1a on self-confguration (Chapter 15 of this book), and gives some developer instructions for its use. The API takes the form of Oz functions and procedures for creating, navigating and querying Fractal-like structures. It enables the construction of distributed, self-deployable and self-monitoring components, at least for mid-size, or cluster-size systems.

10.2 Contractors contributing to the deliverable

INRIA (P3) contributed to this deliverable.

10.3 Components and navigation API

We list here most primitives related to the component model and to the deployment process introduced in the FructOz framework, presented in Chapter 15 of this book. Deployment primitives are an integral part of the Framework since they allow the construction of distributed (i.e. whose implementations span multiple machines), self-deployable and self-configurable components. The primitives are annotated with typing information conforming to the ML standard. The first letter of a primitive usually indicates the type of data it mainly applies to.

10.3.1 Notations

Typing information use the following notations:

$\mathcal{C}$: a component (i.e. a Membrane)

$\mathcal{I}$: an interface

$\mathcal{B}$: a binding

$\mathcal{S}$: a set

$\mathcal{S}\{X\}$: set of elements of type X

$\mathbb{N}$: a numeral (either an integer: $\mathbb{Z}$ or a float: $\mathbb{R}$)

$\mathbb{B}$: a boolean

10.3.2 Component model

The following primitives, sorted on the main entity they apply to, are related to component entities manipulations.

Components (i.e. Membranes)

CNew: $unit \rightarrow \mathcal{C}$
This operation creates a new empty component membrane.

CAddInterface, CRemoveInterface: $\mathcal{C} \times \mathcal{I} \rightarrow unit$
These operations add or remove, respectively, an interface to or from the target component.

CGetInterfaces: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$
This operation retrieves the set of interfaces of the target component;

Interfaces

INew: $(client|server) \rightarrow \mathcal{I}$
This operation creates a new interface, client or server, not associated to any component;

IImplements: $\mathcal{I} \times$ (native Oz object) $\rightarrow unit$
This operation defines the procedure or object to invoke to process messages addressed to an interface.

IResolveSync, IResolveAsync: $\mathcal{I} \rightarrow$ (message $\rightarrow unit$)
These operations resolve an interface into a synchronous or asynchronous proxy that may directly be invoked to send messages to the interface. An asynchronous proxy just fires off a message without waiting for a response. A synchronous proxy blocks the calling thread in wait of a response.

IGetComponent: $\mathcal{I} \rightarrow \mathcal{C}$
This operation retreives the component owning the target interface.

IGetBindingsFrom, IGetBindingsTo: $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$
These operation get the set of bindings connected from (resp. to) the target interface.

IIsClient, IIsServer: $\mathcal{I} \rightarrow \mathbb{B}$
These operation test whether the target interface is a client (resp. server) interface with respect to its owning component.

Bindings

BNew: $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{B}$
This operation creates a binding between a client and a server interface;

BNewLazy: $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{B}$
This operation creates a lazy binding between a client and a server interface: the client interface is required to be instantiated to establish the binding, while the server interface will remain lazy (unneeded), as long as no introspection occurs involving the interface and no functional usage of the binding happens.

BBreak: $\mathcal{B} \rightarrow unit$
This operation breaks the target binding.

BGetClientInterface, BGetServerInterface: $\mathcal{B} \rightarrow \mathcal{I}$
This operation retrieves the interface the given binding connects from (resp. to).

Generic browseable entities

Components, interfaces and bindings are browseable entities, and are thus associated with a set of tags. Tags allow the identification of any entity. Tags are manipulated with the following primitives:

Tag, Untag: entity $\times$ *tag* $\rightarrow$ *unit*
: This operation applies or removes a tag to or from the given entity.

HasTag: entity $\times$ *tag* $\rightarrow \mathbb{B}$
: This operation tests an entity (a component membrane, an interface or a binding) for the given tag.

Controllers

The FructOz framework comprises a set of functions and procedures providing additional controller capabilities. In particular, one finds the equivalent of Fractal attribute (for attaching arbitrary meta data to components) and content (for manipulating the subcomponents of a composite one).

Attribute control primitives are given below.

CListAttributes: $\mathcal{C} \rightarrow \mathcal{S}\{\text{Name}\}$
: This operation returns the set of attribute names of the target component (an atttribute is essentially a pair Name#Value, where Value can be an arbitrary Oz value).

CHasAttribute: $\mathcal{C} \times \text{Name} \rightarrow \mathbb{B}$
: This operation returns **true** if the target component has an attribute of the indicated name.

CSetAttribute: $\mathcal{C} \times \text{Name} \times \text{Value} \rightarrow \textit{unit}$
: This operation sets the attribute of the target component designated by its name to the indicated value.

CGetAttribute: $\mathcal{C} \times \text{Name} \rightarrow \text{Value}$
: This operation returns the value associated with an attribute of a component, designated by its name.

CRemoveAttribute: $\mathcal{C} \times \text{Name} \rightarrow \textit{unit}$
: This operation removes the attribute from the set of attributes of the target component .

Content control primitives are given below.

CListContentContexts:$\mathcal{C} \rightarrow \mathcal{S}\{\text{Context}\}$
: This operation retrieves the set of contexts associated with the target component. A context is essentially a (dynamic) set of sub-components of a given component.

CAddSubComponent : $\mathcal{C} \times \mathcal{C} \times \text{Context} \rightarrow unit$
This operation adds a subcomponent to the given context of the target component.

CRemoveSubComponent : $\mathcal{C} \times \mathcal{C} \times \text{Context} \rightarrow unit$
This operation removes a subcomponent from the given context of the target component.

CGetSubComponents: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
This operation retrives the set of subcomponents of the target component.

CGetSuperComponents : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
This operation rertrieves the set of parent components of the target component.

10.3.3 Deployment primitives

The following primitives are related to the deployment process and to the control of the distributed environnement.

Deploy: package $\rightarrow \mathcal{C}$
This operation deploys the target component package on the local host.

RemoteDeploy: package $\times$ ($\mathcal{C}$: Host) $\rightarrow \mathcal{C}$
This operation deploys the target component package on the specified host component.

NewCluster: $unit \rightarrow$ ($\mathcal{C}$: Cluster)
This operation creates an empty cluster component.

NewRemoteHost: ($\mathcal{C}$: Cluster) $\times$ (string : hostname) $\rightarrow$ ($\mathcal{C}$: Host)
This operation creates a new host component on the given remote host identified by its host name (string), and integrate the new host component to the given cluster component.

CloseHost: ($\mathcal{C}$: Host) $\rightarrow unit$
This operation removes the host component and shutdown the corresponding remote virtual machine.

CloseCluster: ($\mathcal{C}$: Cluster) $\rightarrow unit$
This operation shutdowns all hosts contained in the target cluster.

10.3.4 Introspection, navigation and query primitives

The dynamic computation model reimplements a set of standard primitives and collections data types, such as booleans, numericals and sets.

Dynamic standard data types

BNot: $\mathbb{B} \to \mathbb{B}$
This operation computes the boolean negation.

BAnd, BOr: $\mathbb{B} \times \cdots \times \mathbb{B} \to \mathbb{B}$ or $\mathcal{S}\{\mathbb{B}\} \to \mathbb{B}$
These operations compute the logical "and" and "or" operations.

BWait: $\mathbb{B} \to unit$
This operation waits until the given boolean becomes true.

NSum, NMultiply,

NMin, NMax, NAverage: $\mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ or $\mathcal{S}\{\mathbb{N}\} \to \mathbb{N}$
These operations apply numerical aggregation operators (sum, multiply, minimum, etc).

NSubtract: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
This operation computes the difference between two numerals.

NDivide: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
This operation computes the floating-point division of two numerals.

NIDivide: $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
This operation computes the integer division of two numerals.

SNew: $unit \to \mathcal{S}$
This operation creates a new empty dynamic set.

SSize: $\mathcal{S} \to \mathbb{Z}$
This operation retrieves the number of elements of the target dynamic set.

SIsEmpty: $\mathcal{S} \to \mathbb{B}$
This operation test the emptiness of the target dynamic set.

SUnion: $\mathcal{S} \times \cdots \times \mathcal{S} \to \mathcal{S}$ or $\mathcal{S}\{\mathcal{S}\} \to \mathcal{S}$
This operation computes the union of the given dynamic set of dynamic sets.

SFilter: $\mathcal{S} \times \mathcal{F} \to \mathcal{S}$
This operation extracts a subset from the target set given a predicate function, of the type $\mathcal{F} : \mathcal{V} \to \mathbb{B}$.

SMap: $\mathcal{S} \times \mathcal{F} \to \mathcal{S}$
This operation, where $\mathcal{F} : \mathcal{V} \to \mathcal{V}$ is assumed deterministic, computes a mapped set obtained when applying the given map function to all elements of the dynamic set.

SSubtract: $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$
This operation computes the difference between two dynamic sets.

SIntersect: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$
This operation computes the intersection of two dynamic sets.

This list above provides an overview of the most useful dynamic primitives. This set can easily extended to cover additional dynamic operations.

Navigation primitives

The LactOz library provides a set of common reusable navigation primitives that may be composed and extended to describe arbitrarily complex dynamic navigation expressions.

IIsBoundExternally, IIsBoundInternally: $\mathcal{I} \rightarrow \mathbb{B}$
These operations test whether the interface is externally (resp. internally) bound relatively to the implicit inside of its owning component.

CGetExternalBindingsFrom, CGetExternalBindingsTo,

CGetExternalBindings: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
These operations retrieve the external bindings to and/or from the target component.

CGetInternalBindingsFrom, CGetInternalBindingsTo,

CGetInternalBindings: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
These operations retrieve the internal bindings to and/or from the target component.

BGetClientComponent, BGetServerComponent: $\mathcal{B} \rightarrow \mathcal{C}$
These operations retrieve the component client (resp. server) of the target binding.

CGetExternalComponentsBoundTo, CGetExternalComponentsBoundFrom,

CGetExternalComponentsBoundWith: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
These operations retrieve the external components bound to and/or from the target component.

CGetInternalComponentsBoundTo, CGetInternalComponentsBoundFrom,

CGetInternalComponentsBoundWith: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
These operations retrieve the internal components bound to and/or from the target component.

As before, this list only provides an overview of the navigation primitives, and may be extended as necessary to cover more operations.

10.4 Starting with FructOz and LactOz

Here is a short description explaining how to start using the primitives listed in the previous sections.

We assume that the FructOz and LactOz compiled modules are copied to a well known and available location. For instance, copying the `*.ozf` module files into the ~/.oz/cache/x-ozlib/<username>/ user directory will make the modules available under the following URL: x-ozlib://<username>/*.ozf*.

Once the module files have been compiled and correctly set up, one may import and use them as follows:

```
functor
import
   Utils at 'x-ozlib://<username>/Utils.ozf' % dynamic computation toolset
   FructOz at 'x-ozlib://<username>/FructOz.ozf' % FructOz and LactOz primitives
   ClusterModule at 'x-ozlib://<username>/Cluster.ozf' % distributed cluster bootstrap

define
   %% Include FructOz and LactOz definitions
   \insert 'FructOzHeader.oz'

   %% FructOz and LactOz primitives are now directly usable
   HostA = {NewRemoteHost 'hostname'}
end
```

Furthermore, we provide an Oz code stub named FructOzHeader.oz which adds a few naming shortcuts to the current declaration scope. Including this code stub allows one to make direct use of most FructOz and LactOz primitives without the need for module prefixing.

Chapter 11

D2.3b: Report on Formal Operational Semantics - Formal Fractal Specification

11.1 Executive Summary

This report contains a formal specification of the Fractal model, written in the Alloy v4 specification language, and verified using the Alloy Analyzer model checking tool. The Fractal model is the programming language independent component model at the basis of the development in WP2 and WP4 of the Selfman project. This specification is intended as a first step towards the formal specification of the Kompics model, reported in Deliverable D2.2b (Chapter 8 of this book).

11.2 Contractors contributing to the deliverable

INRIA (P3) contributed to this deliverable.

11.3 Introduction

The Fractal component model [25] is a programming-language-independent component model, which has been introduced for the construction of highy configurable software systems. The Fractal model combines ideas from three main sources: software architecture, distributed configurable systems, and reflective systems. From software architecture, Fractal inherits basic concepts for the modular construction of software systems, encapsulated components and explicit connections between them. From reflective systems, Fractal inherits the idea that components can exhibit meta-level activities and reify through controller interfaces part of their internal structure. From configurable distributed systems, Fractal inherits explicit component connections across multiple address spaces, and the ability to define meta-level activites for run-time reconfiguration. The Fractal model has been used as a basis for the development of several kinds of configurable middleware, and has been used successfully for building automated, architecture-based, distributed systems management capabilities, including deployment and (re)configuration management capabilities [6, 28, 49], self-repair capabilities [22, 134], overload management capabilities [23], and self-protection capabilities [33].

The Fractal model is currently defined by an informal specification. The specification only briefly mentions the general foundations that constitute the Fractal model per se, and focuses mostly on default meta-level capabilities (or controllers, in Fractal parlance). The specification has been successfully implemented in different languages and environments, notably in Java and C, without giving rise to serious issues, which is a testimony to its consistency. However, there are aspects of the specification that remain decidedly insufficiently detailed or ambiguous. The present report attempts to correct these deficiencies by developing a formal specification of the Fractal component which makes explicit the underlying general component model which constitutes the foundation of Fractal; which clarifies a number of ambiguities in the informal Fractal specification; and which identifies places where the informal Fractal specification may be overconstraining.

Beyond ensuring the consistency of the Fractal model, a formal specification for the Fractal model can serve several purposes: to provide a more abstract, truly programming language independent specification of the Fractal model; to allow a formal verification of Fractal designs; to allow a formal specification and verification of Fractal tools; to allow a rigorous comparison with other component models, and in particular to assess whether a component model constitutes a proper refinement or specialization of the Fractal model. The latter is important because the Fractal specification aims to define a very general component model (e.g. meta-level capabilities in Fractal are not fixed, nor is the semantics of composition realized by composite components), from which more specialized component models can be derived and combined.

The specification in this report is written in Alloy 4 [1, 75, 76], a lightweight specification language based on first-order relational logic. Alloy is interesting be-

cause of its simplicity and because of the straightforward usage of its analyzer, which acts essentially as a model checker and counter-example generator, and which enables rapid iterations between modelling and analysis when writing a specification (very much akin to debugging a specification). For a detailed introduction and motivation of Alloy, we refer the interested reader to the book [76]. An online tutorial for Alloy is also available on the Alloy Analyzer Web site [1].

The report is written in a litterate programming style: the specification is presented in its entirety, the (informal) commentary on the formal specification being interspersed with excerpts of the Alloy code. All assertions (Alloy facts) and theorems (Alloy assertions) have been checked with the Alloy analyzer, checking for the existence of finite models in the first case, and for the absence of counter-examples in models below a certain size in the second case. We do not introduce Alloy nor the Fractal model. Hopefully, the commentary running along the Alloy code excerpts will suffice.

The report is organized as follows. Section 11.4 discusses related work. Section 11.5 details the Alloy specification of the core Fractal concepts. Section 11.6 details the Alloy specification of the naming and (distributed) binding framework associated with Fractal. The following sections, Section 11.7 to Section 11.10, detail the Alloy specification of the different optionnal Fractal controllers mentioned in the informal Fractal specification. These different controllers are key primitive effectors for management operations in an architecture-based approach of self-management: this is shown e.g. by [134] in the context of self-repair.

11.4 Related work

There have been several approaches to the formalization of component-based software and component models. Representative samples are provided by the two books [93, 97]. The two bodies of work closest to ours are: the co-algebraic approach developed by Barbosa, Meng et al. [92, 13, 102, 103], and the formal specification in Alloy of Microsoft COM component model developed by Jackson and Sullivan [77], following work by Sullivan et al. on the formal specification of the COM model in Z [137]. Although the presentation we give is relational, the notion of component or kell we develop in this report is essentially coalgebraic in nature since a kell can be understood primarily as a set of transitions. Whereas Barbosa et al. develop a categorical framework, we prefer to adopt a simpler set-based approach: while we lose the benefit of dealing in the same way with multiple forms of behavior (e.g. probabilistic, time-based, etc) as in [92], the intuition is in our view better aided by a set-based presentation, and it is easier to understand for it directly generalizes the well-known notion of transition system. The COM specification presented in [77] focuses on the structural aspects of the COM model, and notably on the definition of its query interface and aggregation mechanism. While the Component controller in the Fractal specification provides much the same functionality as the query interface in COM, the Fractal model does not exhibit

the COM-specific difficulty arising in the interplay between query interface and aggregation highlighted in [137], and possesses several forms of meta-level behavior (so-called controllers and controller interfaces). Thus, our work focuses more on the specification of meta-level behavior, and in particular on the interplay between base level behavior and meta-level behavior in a component.

11.5 Foundations

This first part of the specification captures the underlying core of the Fractal model: a very general notion of component, called kell (a remote reference to the biological cell). At this level of abstraction, the notion of kell first emphasizes two facts:

- A kell has entry points, called gates. The notion of gate is an abstract form of the notion of interface in the Fractal specification. A gate constitutes a named point of interaction between a kell and its environment. The set of gates of a kell constitutes its sole means of interaction with its environment, i.e. a kell is a unit of encapsulation.
- A kell may have subcomponents, called subkells. All transitions in a kell may act on the set of subcomponents, and modify it in arbitrary ways. This flexibility is key to allow different semantics for composition, and to support different kinds of meta-level operations (i.e. operations operating on the internal structure and behavior of components).

The first primitive sets in the Alloy specification of the core Fractal model are given below. [1]

```
module fractal/foundations

sig Id {}
sig Val {}
sig Op extends Id {}
```

The above declarations introduce three primitive sets: Id, Val, and Op, respectively. They correspond, respectively, to the set of identifiers, base values, and operation names. Identifiers are just primitive forms of names or references. Base values represent values of some (unspecified) data types, such as integers, booleans, strings,

[1] In Alloy, primitive sets are just sets of atoms, i.e. elements which have no internal structure (and are not sets – atoms are sometimes called urelements in the logic litterature, e.g. as in [14]). Primitive sets are called signatures in Alloy, hence the key word **sig** for introducing them. Note also the module declaration: in ALloy, specifications can be broken down into modules, which can then be imported for use in other modules using a declaration of the form: **open** moduleX **as** X, where X is some local name used, in the current module, as an abbreviation for the imported module.

etc. At this level of abstraction, the exact forms base values can take is of no import, hence their specification as just atoms. [2]

The general notions of interface and component in the core Fractal model are given below. They are called, respectively, gate and kell.

```
sig Gate {
      gid: Id
}

sig Kell {
      gates: set Gate,
      sc: set Kell,
      kid: Id
}

fact GatesInIKellHaveUniqueIds {
     all c:Kell | all i,j:c.gates | i.gid = j.gid implies i = j
}
```

A gate, i.e. an element of the set Gate, is an entry point to communicate with a kell. The declaration above stipulates that a gate has an identifier (in Alloy, a declaration of the form gid:Id can be read as declaring a feature, or instance variable, of the class Gate; formally, it declares a binary relation $\mathsf{gid} : \mathsf{Gate} \rightarrow \mathsf{Id}$ between the set of gates, Gate, and the set of identfiers, Id). A kell, i.e. an element of the set Kell, is defined as having an identifier, given by the feature kid, a set of gates, given by the feature gates, and a set of subcomponents, given by the feature sc. The fact that a kell has an identifier is necessary (e.g. for management purposes) to manifest a notion of identity that persists throughout state changes. The Alloy fact named GatesInIKellHaveUniqueIds expresses an invariant on kells, namely that gates that belong to a kell have distinct identifiers. [3]

These elements provide the basic structure of a kell but do not explain how it behaves. This is captured by the definition of the set TKell below, which endows kells with transitions. Transitions are defined below as 4-tuples that comprise a set of initial kells (feature tsc), a set of input signals (feature sin), a set of output signals (feature sout), and a set of residual kells (feature res). Intuitively, the initial set of kells of a transition corresponds to subkells of the kell to which the transition

[2] Keyword **extends** in Alloy indicate that a primitive set is declared as a subset of another one (and that it will form, with other subsets similarly declared, a partition of the set it extends).

[3] This invariant takes the form of a simple first-order logical formula, where the keyword **all** denotes the universal quantifier $\forall$, where a declaration such as c:Kell denotes an arbitrary element c of the set Kell (likewise, i:c.gates denotes an arbitrary element i of the set of gates of the kell c – the dot notation c.gates is the standard notation for accessing a feature, or attribute, of an instance of a class). In a more classical logical notation, the GatesInIKellHaveUniqueIds formula would read:

$$\forall \mathtt{c} \in \mathtt{Kell},\ \forall \mathtt{i}, \mathtt{j} \in \mathtt{gates}(\mathtt{c}),\ \mathtt{gid}(\mathtt{i}) = \mathtt{gid}(\mathtt{j}) \Rightarrow \mathtt{i} = \mathtt{j}$$

belongs (the set of subkells on which the transition acts). The set of residual kells are the kells produced by the transition. The kell to which the transition belongs may or may not belong to the residual of the transition. This allows us to model component factories, as in the Fractal specification, i.e. component that can create other components, or operations that delete or transform the target component. Effectively, this means that a kell can be seen as some sort of generalized Mealy machine (a labelled transition system, whose labels denote input and output signals handled during a transition).

```
sig TKell in Kell {
    transitions: set Transition
}

sig Transition {
    tsc: set Kell,
    sin: set Signal,
    sout: set Signal,
    res: set Kell
}

fact TransMayNotHaveDifferentSubComps { all c:TKell | all t:c.transitions | t.tsc = c.sc }
```

The invariant TransMayNotHaveDifferentSubComps ensures that the initial kells associated with each transition of a given kell care indeed the subkells of c.

Signals are defined below as records of arguments (feature args), with a target gate (feature target), i.e. the gate at which a signal is received (if it is an input signal) or emitted (if it is an output signal), and an operation name (feature op). In object-oriented terms, a signal looks very much like a reified method invocation.

```
sig Signal {
    target: Gate,
    operation: Op,
    args: Id -> set Arg
}

sig Arg in Id + Val + Gate + Kell {}

fact SignalsTargetInterfaces { all c: TKell | c.transitions.(sin + sout).target in c.gates }
```

Signal arguments belong to the set Arg defined above as the union[4] of four sets: identifiers, values, gates and kells. This means in particular that signal may carry gates (much as in the π-calculus messages may carry channel names [5], and kells. The latter capability is not explicitly reflected in the Fractal specification, but

[4]In Alloy, in denotes the subset relation, or the set membership relation, + denotes set union, & denotes set intersection, – denotes set difference, # denotes set cardinality.

[5]Note that in the π-calculus, channels, i.e. communication capabilities, are just names. Strictly speaking we could have avoided to include gates as posible arguments to signal, by just relying on identifiers. However, we have been careful in the specification to ensure that gates remain immutable, in contrast to kells (i.e. no operation will transform a gate into another with the same identifier). Having a gate identifier or a gate itself as an argument are thus strictly equivalent, but allowing gates as signal arguments simplifies the specification.

is required to model mobile agents and strong mobility, as well as deployment, checkpointing and reconfiguration capabilities[6]. The fact SignalsTargetInterfaces expresses the invariant that all target gates in signals appearing in transitions of a kell c are gates of c.

Discussion

This completes the specification of the core Fractal model. As can be seen the core is very small, and it merely asserts that components are higher-order Mealy machines, that can be hierarchically organized. This core model allows a number of seemingly unusual, or unexpected, features:

- We allow kells to have no gates, and thus only internal behavior. As a result, the following assertion [7] is not valid:

  ```
  assert AllKellsHaveGates { all c:TKell | some c.gates }
  ```

  In contrast, the following assertion is valid:

  ```
  assert NoInterfaceImpliesInternalActions {
     all c:TKell | (no c.gates) implies (no c.transitions.(sin + sout) )
  }
  ```

  It asserts that if a kell has no gate, then its transitions are merely internal as they involve no exchange of signals with the environment.

- We allow component structures with sharing, i.e. a kell may be a subkell of two different kells. Thus, the following assertion to the contrary is invalid:

  ```
  assert SharingIsImpossible {
     all c1,c2:Kell | no cs:Kell {
        cs in c1.sc & c2.sc and (not c1 = c2) and (not c1 in c2.sc) and (not c2 in c1.sc)
     }
  }
  ```

  Component sharing is an original feature of the Fractal model, which as been found useful to model situations with resource sharing, i.e. where components at different places ina component hierarchy need access to same resource such as a software library, or an operating system service.

[6] Alernatively, one could model this through marshalling and unmarshalling operations, allowing to transform a kell into a value, and vice versa. The above specification of arguments makes this higher-order character of signals and of the kell model explicit.

[7] An assertion in Alloy is written exactly like a fact, except for the use of the **assert** keyword to declare it. An invalid assertion is detected by the Alloy Analyzer when it generates a finite model that contradicts it (a counterexample). The Alloy keyword **some** denotes the existential qualifier. Thus **some** c:Kell | P where P is some predicate, asserts the existence of some kell c verifying P. By extension, **some** s, where s is a set, asserts that s is not empty (i.e. that there is some element in s).

- We allow component structures which are not well-founded, i.e. where a kell may appear as a subkell of itself, or as a subkell of some of its subkells, etc. Thus, the following assertion to the contrary is invalid[8]:

```
assert ContainmentIsWellFounded {
    all c:Kell | (not c in c.*sc)
}
```

  This may seem counterintuitive, however this is not really different from allowing recursive procedure calls, and we therefore do not enforce it at the level of abstraction of the core model. The Fractal specification explicitly disallows this (ContainmentIsWellFounded would be witten as an invariant – an Alloy fact), but this feature could be interesting to model recursive component structures. This is one occurrence where the present formal specification relaxes the constraints from the informal Fractal specification.

- We allow components to have a varying number of interfaces during their lifetime, i.e. kells to have a varying number of gates in the course of their execution. Thus, the following assertion to the contrary is invalid:

```
assert NumberOfGatesInKellDoesNotVary {
    all c:TKell | all t:c.transitions | all cr:TKell {
        (cr in t.res and cr.kid = c.kid) implies (cr.gates = c.gates)
    }
}
```

- In contrast to most other component models (see e.g. [91] for a discussion of recent ones), we do not need to introduce a notion of connector or binding to mediate the communication, or reify the communication paths, between components. Sharing, containment (i.e. the kell-subkells relationship) and the fact that each component or kell institutes its own composition semantics suffice to make explicit communication channels in a component structure, and to define their semantics.

The specification of kells as Mealy machines has however one major drawback as an Alloy specification. Because transitions make explicit the state changes that a kell may go through, specifying state changes in the present specification amounts to require that certain facts hold, which would take the form of closure properties such as "kells of this kind – and the kells that appear in the residues of their transitions – must have transitions of this sort". For instance, we would require all kells that support the Component gate to have certain transitions implementing the Component operations, and all the kells in the residues of their transitions to be kells of a similar kind. Closure properties of this kind are unfortunately instances of so-called generator axioms that may lead to a state explosion in models of the specification, which makes them impossible to analyze using the model checking

[8] In Alloy, *r of some relation r denotes the reflexive and transitive closure of r, while ^r denotes its transitive closure.

approach of the Alloy Analyzer. This problem is an instance of the unbounded universal quantifiers problem, discussed in Section 5.3 of [76], and needs to be avoided if we want to exploit the Alloy Analyzer in assessing the consistency of the specification. Our approach in this specification is to not describe explicitly the set of transitions logically associated with a kell. Instead, we will define Alloy predicates that describe state changes on certain kells, but refrain from imposing that these state changes appear as explicit transitions in the supporting kells. In effect, for the purposes of this specification, we will deal only with elements of the Kell set, and will not consider elements of the TKell set. In the following sections, we adopt this approach: all properties and predicates considered will deal apply to elements of Kell.

Before moving to the specification of the (optional) default meta-level capabilities of the Fractal model, we gather here a number of declarations used in the rest of the specification. The distinction between Client and Server is here merely a primitive type distinction, which governs bindings between gates: to bind two gates together, one must be a dual of the other. The denominations Client and Server merely reflect that duality. The predicate isoKell can be interpreted as a strong identity predicate, whereby two kells are strongly identical if they have the same identifier, the same gates and the same subkells (they may differ in their internal state).

```
sig Client extends Gate {}
sig Server extends Gate {}
one sig NoSuchInterfaceException extends Val {}
one sig IllegalBindingException extends Val {}
one sig IllegalLifeCycleException extends Val {}
one sig IllegalContentException extends Val {}
one sig Ok extends Val {}
one sig Null extends Val {}
pred isoKell[c:Kell, c1:Kell] {
    c.kid = c1.kid
    c.sc = c1.sc
    c.gates = c1.gates
}
```

11.6 Naming and binding

The naming and binding part of the specification captures the notions necessary for the construction of distributed configurations. We follow here the informal Fractal specification, [26], Section 2.2.

The first concept is that of name. A name is merely an entity that is used to refer to another one. A name comes equipped with a reference to its naming context (feature context).

```
module fractal/naming
open util/relation as RR
open fractal/foundations as FF

sig Name {
```

```
        context: Id,
        pack: NamePickle
    }
```

A name can also be pickled (or marshalled) to make it persistent or to communicate it between different machines. This is obtained through the combination of the pack feature and of operations encode and decode [9], defined as follows:

```
fact PackUnpackIdempotent { all n:Name | n.pack.unpack = n }

pred encode(n:Name, p:NamePickle) { p = n.pack }

pred decode(nc:NamingContext, p:NamePickle, n:Name) {
    p.context = nc.nid and p.unpack.context = nc.nid implies n = p.unpack
}

assert DecodingYieldsSameNameThanEncoded {
    all p:NamePickle, n:Name, nc:NamingContext {
        encode[n,p] and nc.nid = n.context implies decode[nc,p,n]
    }
}
```

Names exist only within contexts. Contexts ar primarily associations between names and referents (i.e. entities which are refered to by names). Making contexts explicit allows us to define different systems of names and to have them coexist and cooperate without the need to rely on a global naming authority to disambiguate independently created names. Contexts are defined as follows:

```
sig NamingContext {
    nid: Id,
    exported: Name -> lone Referent
}

fact NameRefersToContext { all nc:NamingContext | all n:dom[nc.exported] | n.context = nc.nid }
```

The feature exported in a naming context identifies the association between names in the context and their referents. [10]

Two key invariants, given below, apply to names and contexts. The first one merely asserts that names appearing in a context correctly refer to this context. The second one clarifies the fact that referents cannot be names that belong to the context (they can be names that belong to other contexts, though, thus allowing referral chains to be constructed across multiple contexts).

```
fact NameRefersToContext {
```

[9] Alloy allows the definition of first-order predicates, whose declarations start with keyword **pred** and are optionally followed by the list of the predicate arguments. Operations, that perform state changes in a system can be defined as predicates with some arguments corresponding to the initial state, i.e. the state prior to the execution of the operation, and other arguments corresponding to the final state, i.e. the state resulting from the execution of the operation. For a discussion on how to model state changes in Alloy.

[10] In Alloy a declaration of the form exported: Name -> **lone** Referent denotes an injective binary relation between the set Name and the set Referent. The keyword **lone** is an example of a relation multiplicity. In our case, it signifies that a given name is to be asociated with one, and only one, referent. Of course, two distinct names can have the same referent.

```
        all nc:NamingContext | all n:dom[nc.exported] | n.context = nc.nid
}

fact InContextNamesNotExported {
    all nc:NamingContext | all n:ran[nc.exported] | n.context != nc.nid
}
```

The main operation supported by a naming context is the export operation. Operation export returns a new name n for referent r in context nc. The name n is a valid name for referent r in the context nc. The naming context nc can for instance be a network context where remotely accessible interfaces are given names of a special form (eg URLs for a Web service context). Note that a name can be exported as well. This is necessary to handle names across different naming contexts. A referent that is already a name in the target context cannot be exported. Operation export is specified below. [11]

```
one sig NamingException extends Val {}

pred export (nc1, nc2: NamingContext, r: Referent, n:Name + NamingException){
    let A = not (some s:Referent − r | n−>s in nc1.exported),
        B = (not r.context = nc1.nid) {
         (A and B) implies nc2.exported = nc1.exported + n −> r and nc1.nid = nc2.nid
         else n in NamingException and nc1 = nc2
    }
}
```

One may verify a number of properties in relation to operation export. Here are a few self-explanatory ones:

```
assert ExportReturnsNewNameOrOldMap {
    all nc,ncc:NamingContext, r:Referent, n:Name |
        export[nc,ncc,r,n] implies
            let A = (not n in dom[nc.exported]),
                B = (n.(nc.exported) = r),
                C = (not r.context = nc.nid) {
              (A or B) and C
            }
}

assert ExportNameBelongsToContext {
    all nc,ncc:NamingContext, r:Referent, n:Name |
        export[nc,ncc,r,n] implies n.context = nc.nid
}

assert ExportExceptionLeavesContextUnchanged {
    all nc,ncc: NamingContext, r:Referent, n:NamingException |
        export[nc,ncc,r,n] implies nc = ncc
}
```

The following assertions highlight the fact that name resolution within a single naming context can be partial. To be complete, name resolution must typically

[11] Note the use of the Alloy construct let A = ... { S }, which just declares a variable A to stand as a denotation for some value, denotation which is then used inside the statement S. Note also the use of a nested implication of the form C1 implies F1 else F2, which is equivalent to (C1 and F1) or ((not C1) and F2). Note, finally, the keyword one which precedes the declaration of the NamingException value: it just signifies that the set NamingException is a singleton. In Alloy, set elements are essentially identified with singletons.

take place across several naming contexts. However, we also allow partial name resolution across several naming contexts. This takes care of situations were name resolution cannot be carried out in full (eg in disconnected situations) or need not be carried out in full (e.g. when no access to a referenced interface or component is attempted).

```
assert ExportClosureIsJustExport {
    all nc:NamingContext | nc.exported = ^(nc.exported)
}

assert ExportClosureEndsInInterfaceOrNotInContextName {
     all nc:NamingContext, n:Name, r:Referent |
            r in n.(nc.exported) implies (r in Gate) or (r in Name and r.context != nc.nid)
}
```

A binder is naming context that can resolve names and establish connections (bindings) towards entities refered to by resolved names. A binding is created typically by a bind operation. The creation of a binding results in the creation of a component that provides a (local) interface which corresponds to (e.g. is a proxy to) the resolved name. A binder records the assocation (bindings) between resolved names and the (local) interfaces they refer to. Binders are specified below.

```
sig Binder extends NamingContext {
     bindings: Name -> lone Gate,
}

fact BindingNamesBelongToContext {
     all b:Binder | all n: dom[b.bindings] | n.context = b.nid
}

fact BindingsAndExportedDomainsDisjoint {
     all b:Binder | no (dom[b.bindings] & dom[b.exported])
}
```

The bind operation is specified below, together with some self-explanatory properties.

```
pred bind(b,b1:Binder, n:Name, i:Gate + NamingException) {
     b.nid = b1.nid
     n -> i in b.exported implies b = b1
     else i in Gate implies b1.bindings = b.bindings + n -> i
     else i in NamingException and b = b1
 }

assert BindExceptionLeavesBinderUnchanged {
      all b,b1:Binder, n:Name, i: NamingException | bind[b,b1,n,i] implies b = b1
}

assert BindReturnsNewInterfaceOrFromExported {
     all b,b1:Binder, n:Name, i:Gate | bind[b,b1,n,i] implies n -> i in b.exported + b1.bindings
}
```

Finaly, one can prove a correct interplay between export and bind, namely that bind returns a local (in-context) gate referred to by a previously exported name.

```
assert BindReturnsPreviouslyExportedReferent {
    all b,b1:Binder, n:Name, i:Gate |
              export[b,b1,i,n] implies bind[b1,b1,n,i]
}
```

11.7 Component controller

The component controller in Fractal supports basic introspection capabilities: discovering all the interfaces associated with a component and their type. We follow here the informal Fractal specification, [26], Section 3.

Our formal specification of the component controller begins with the declaration of the Type signature, with its subtype relation, noted sstypes. At this level of abstraction, the only property recorded of the subtype relation is that it constitutes a partial order.[12]

```
module fractal/component
open util/relation as RR
open fractal/foundations as FF
open util/graph[Type] as GG

sig Type extends Val {
    sstypes: set Type
}

fact SubTypingIsPartialOrder { GG/dag[sstypes] }

sig CompType extends Type {}
sig InterfaceType extends Type {}
```

The next signatures declare the Component gates and the Interface gates. A Component gate is a server gate which also records the type of the component it belongs to. An Interface gate records its type, as well as the Component gate of the component it belongs to. As noted in the informal Fractal specification, this setting is similar to that adopted by the Microsoft COM model, with Component corresponding to the COM IUnknown interface.

```
sig Component extends Server {
    ctype: CompType
}

sig Interface in Gate {
    owner: Component,
    itype: InterfaceType,
}
```

A ckell is now defined as a kell with one gate which is an instance of Component (its other gates can be arbitrary gates). The fact CKellsHaveComponent constraints ckells to have only one Component gate.

```
sig CKell in Kell {
    comp: Component
}

fact CKellsHaveComponent { all c:CKell | c.comp = c.gates & Component }
```

Likewise, we define an ikell as a kell whose gates are all instances of Interface.

```
sig IKell in Kell {}

fact IKellsHaveInterfaces { all c: IKell | c.gates in Interface }
```

[12] Note the use of the Alloy utility module graph, and the predicate dag from this module.

We now define compkells as ckells which are also ikells, thus, as kells which have a Component gate, and whose gates are all instances of Interface.

```
sig CompKell in Kell {}

fact CompKellsAreCKellsAndIKells { CompKell = CKell & IKell }

fact InterfacesInCompKellsHaveOwner { all c: CompKell | all i:c.gates | i.owner = c.comp }
```

The basic properties of compkells are corroborated by the following simple, self-explanatory assertions.

```
assert OneComponentPerCompKell {
    all c:CKell | one c.gates & Component
}

assert ComponentInCompKellsIsInterface {
    all c:CompKell | c.comp in Interface
}

assert CompKellsHaveOnlyInterfaces {
     no c:CompKell { some c.gates & (Gate - Interface) }
}

assert OwnersInCompKellsAreComponent {
    all c:CompKell | all i:c.gates | i.owner in Component & Interface
}
```

Before specifying the different operations that are attached to Component and Interface, we first define an equivalence predicate on compkells. Roughly, isoCKell indicates that two compkells have the same identifier, the same subcomponents, and the same gates, i.e. their internal and external structures (but not necessarily their exact states) are the same. By virtue of the above invariants, two equivalent compkells have the same Component gate.

```
pred isoCKell[c:CompKell, c1:CompKell] {
    isoKell[c,c1]
}

assert IsoCompKellsHaveSameComponent {
    all c,c1:CompKell | isoCKell[c,c1] implies c.comp = c1.comp
}
```

We give below the different operations attached to a Component gate. Operation getInterfaces returns the set of gates is of a compkell c, given its Component interface o. In the process, compkell c becomes compkell c1, which is required to be equivalent to c, i.e. have the same gates, the same identifier, and the same subkells. Operation getInterface returns the gate i whose identifier iid is passed as argument to the operation. In the process, compkell c becomes compkell c1. Operation getCType returns the component type ct of compkell c.

```
// Operations from the Component interface

pred getInterfaces[c:CompKell, o:Component, is: set Interface, c1:CompKell] {
   o = c.comp
   is = c.gates
   isoCKell[c,c1]
}
```

```
pred getInterface[c:CompKell, o:Component, iid:Id, i:Interface + NoSuchInterfaceException, c1:CompKell] {
    o = c.comp
    isoCKell[c,c1]
    i in c.gates implies iid = i.gid
    else i = NoSuchInterfaceException
}

pred getCType[c:CompKell, o:Component, ct: CompType, c1:CompKell] {
    o = c.comp
    ct = o.ctype
    isoCKell[c,c1]
}
```

The specification of the above operations provide examples of ambiguities that arise in the informal Fractal specification (in fact, in any informal specification), and which are difficult to weed out without a formal model. In fact, [26] Fractal specifications leaves unspecified the exact postconditions of operations. Here we strike a middleground between a strong form which would require that the target compkell be left untouched, i.e. that would specify c = c1 in place of our isoCKell[c,c1], and a very weak form which would only require c.kid = c1.kid. The strong form would forbid any kind of side-effect to such meta-level operations (such as, e.g. setting a counter or updating a log of such operations), whereas the very weak form would make these introspection operations essentially useless (since the obtained information would be obsolete as soon as it is obtained).

We specify below operations associated with an Interface gate. Operation getOwner returns the Component gate associated with the compkell c that hosts the target Interface gate i. Operation getName returns the identifier of the target Interface gate i. Operation getType returns the interface type it of the target Interface gate i.

```
// Operations from the Interface interface

pred getOwner(c:CompKell, i:Interface, o:Component, c1:CompKell) {
    i in c.gates
    o = i.owner
    isoCKell[c,c1]
}

pred getName[c:CompKell, i:Interface, iid:Id, c1:CompKell] {
    i in c.gates
    iid = i.gid
    isoCKell[c,c1]
}

pred getType[c:CompKell, i:Interface, it: InterfaceType, c1:CompKell] {
    i in c.gates
    it = i.itype
    isoCKell[c,c1]
}
```

We give below two simple properties, which assert the consistency of the Component and Interface operations.

```
assert ComponentToInterfacesAndBack {
    all c,c1:CompKell | all i: Interface | all is: set Interface {
        getInterfaces[c,c.comp,is,c1] and i in is implies getOwner[c,i,c.comp,c1]
```

```
    }
}

assert InterfaceToComponentAndBack {
    all c,c1:CompKell | all i: Interface | all is: set Interface {
        getOwner[c,i,c.comp,c1] and getInterfaces[c,c.comp,is,c1] implies i in is
    }
}
```

11.8 Binding controller

The binding controller in Fractal supports the binding of client interfaces of a component to server interfaces. The effect of this binding is to allow the components that are connected via these bound interfaces to communicate. We follow here the informal Fractal specification, [26] Section 4.3.

In our case, we do not specify the exact effect of binding a client and a server gate, since the semantics of this binding typically depends on the enclosing component where it takes place. However kells providing a BindingController gate record which client interfaces are bound (feature bindings in an instance of BCKell). Notice the different constraints that apply:

- a client gate is bound at most to a single server gate (**lone** multiplicity in bindings feature declaration);
- client gates must be gates of the hosting kell (fact ClientsInBindingCntrlAreBCKellGates);
- the bindings relation record the binding of client gates (fact BindingsBindClientGates).

```
module fractal/binding
open util/relation as RR
open fractal/foundations as FF

sig BindingController extends Server {}

sig BCKell in Kell {
    bctrl: BindingController,
    clients: set Client,
    bindings: Client -> lone Server
}

fact BindingsBindClientGates {
    all c:BCKell | dom[c.bindings] in c.clients
}

fact ClientsInBCHaveUniqueIds {
    all c:BCKell | all ci,cj:c.clients | ci.gid = cj.gid implies ci = cj
}

fact ClientsInBindingCntrlAreBCKellGates {
    all c:BCKell | c.clients in c.gates
}

fact BindingControllerAreBCKellGates {
```

```
    all c:BCKell | c.bctrl in c.gates
}
```

Before specifying the operations attached to BindingController gates, we define an equivalence predicate between kells with a BindingController gate. Two such kells are equivalent if they have the same identifier, the same subkells, the same client gates and the same BindingController gate.

```
pred isoBCKell(c:BCKell, c1:BCKell) {
    isoKell[c,c1]
    c1.clients = c.clients
    c1.bctrl = c.bctrl
}
```

We specify below the different operations associated with BindingController gates. Operation list returns the set of client gates of the kell hosting the target BindingController gate; in the process, the hosting kell c evolves into kell c1. Operation lookup returns the server gate i that is bound to the client gate whose identifier iid is passed as argument. Operation bind binds the client interface whose identifier cid is passed as argument to the server gate si passed as argument. Finally, operation unbind unbinds the client gate whose identifier cid is passed as argument.

```
// Operations from the BindingController interface

sig BindingReturn in Ok + NoSuchInterfaceException + IllegalBindingException + IllegalLifeCycleException {}

pred list(c:BCKell, bc:BindingController, cs: set Client, c1:Kell) {
    c.bctrl = bc
    cs = c.clients
    isoBCKell[c,c1]
}

pred lookup(c:BCKell, bc:BindingController, iid: Id, i: Server + NoSuchInterfaceException, c1:Kell) {
    c.bctrl = bc
    isoBCKell[c,c1]
    iid in (c.clients).gid implies (some if: Client { if.gid = iid and if in c.clients and i in if.(c.bindings) })
    else i = NoSuchInterfaceException
}

pred bind(c:BCKell, bc:BindingController, cid: Id, si:Server, r: BindingReturn, c1:Kell) {
    bc = c.bctrl
    some ci:Client {
        r = IllegalLifeCycleException implies c1 = c
        else no cid & (c.clients).gid implies r = NoSuchInterfaceException and c1 = c
        else some ci.(c.bindings) implies r = IllegalBindingException and c1 = c
        else ( cid in (c.clients).gid and no ci.(c.bindings) and
                ci.gid = cid and ci in c.clients and
                r = Ok and c1.bindings = c.bindings + ci -> si and
                isoBCKell[c,c1] )
    }
}

pred unbind(c:BCKell, bc:BindingController, cid: Id, r: BindingReturn, c1:Kell) {
    some ci:Client, si: Server {
        c.bctrl = bc
        r = IllegalLifeCycleException implies c1 = c
        else no cid & (c.clients).gid implies r = NoSuchInterfaceException and c1 = c
        else no ci.(c.bindings) implies r = IllegalBindingException and c1 = c
        else ( cid in (c.clients).gid and ci -> si in (c.bindings) and
```

```
                ci.gid = cid and r = Ok and
                c1.bindings = c.bindings - ci -> si and
                isoBCKell[c,c1] )
    }
}
```

We give below a number of properties that assess the mutual consistency of the different BindingController operations. Predicates getClient, getBoundClient, and getBoundServer are just abbreviations for some simple conditions. The last two properties UnbindAfterBindPossible and BindAfterUnbindPossible are commutation conditions on the bind and unbind operations.

```
assert LookupAfterBindYieldsCorrectServer {
   all c:BCKell, cid: Id, si:Server, r: Ok, c1:BCKell |
      bind[c,c.bctrl,cid,si,r,c1] implies lookup[c1,c1.bctrl,cid,si,c1]
}

pred getClient[c:BCKell, cid:Id, ci:Client] {
     cid = ci.gid
     ci in c.clients
}

pred getBoundClient[c:BCKell, cid:Id, ci:Client] {
    getClient[c,cid,ci]
    ci in dom[c.bindings]
}

pred getBoundServer[c:BCKell, cid:Id, si:Server] {
   some ci:Client | getBoundClient[c,cid,ci] and ci -> si in c.bindings
}

assert UnbindPossibleMeansBindingExists {
    all c:BCKell, cid:Id, c1:BCKell {
       unbind[c,c.bctrl,cid,Ok,c1] implies some s:Server { s in Client.(c.bindings) }
    }
}

assert UnbindAfterBindPossible {
   all c:BCKell, cid: Id, si:Server, c1:Kell |
      bind[c,c.bctrl,cid,si,Ok,c1] implies unbind[c1,c1.bctrl,cid,Ok,c]
}

assert BindAfterUnbindPossible {
   all c:BCKell, cid: Id, c1:BCKell, si:Server {
      unbind[c,c.bctrl,cid,Ok,c1] and getBoundServer[c,cid,si] implies bind[c1,c1.bctrl,cid,si,Ok,c]
   }
}
```

11.9 Content controller

The ContentController interface in Fractal allows to introspect the internal structure of a component in the form of its so-called internal interfaces and of its subcomponents. We follow here the informal Fractal specification, [26], Section 4.4.

We specify below kells with ContentController gates, i.e. elements of CCKell. Internal gates appear only as a set of gates, which are not gates for interaction with the

environment, i.e. the exterior, of a kell. There are no further semantics associated with this notion of internal gate, since it typically varies with each kell (internal gates typically allow to explicitly connect subkells to some inner functionality of their parent kell). Making explicit internal gates allows to control, through the ContentController gate, the internal connections between a parent kell and its subkells. Instances of CCKell also provide access to (in general, a subset of) their subkells (feature subcomps in the CCKell signature). Notice that CCKell is defined as a subset of CKell, i.e. each instance of CCKell has both a ContentController gate and a Component gate. All subkells of a kell in CCKell, or cckell, are ckells, i.e. they all have a Component gate.

```
module fractal/content

open util/relation as RR
open fractal/foundations as FF
open fractal/component as FC

sig ContentController extends Server {}

sig CCKell in CKell {
      cctrl: ContentController,
      internals: set Gate,
      subcomps: set CKell
}

fact ContentControllerInCCKellIsExternalGate { all c:CCKell | c.cctrl in c.gates }

fact InternalsAreNotExternalsInCCKells { all c: CCKell | no (c.internals & c.gates) }

fact SubcompsAreSubComponentsInCCKells { all c: CCKell | c.subcomps in c.sc }

fact InternalsIdsAreDistinct { all c:CCKell | all g,g1:c.internals | g.gid = g1.gid implies g = g1 }

fact CCKellsHaveDistinctComponentsInSubComps {
      all c:CCKell | all c1,c2:c.subcomps {
                  c1.comp = c2.comp implies c1 = c2
                  c1.kid = c2.kid implies c1 = c2
      }
}

assert CCKellsHaveCompAsIdsInSubComps {
     all c:CCKell | all c1,c2:c.subcomps {
          c1.kid = c2.kid <=> c1.comp = c2.comp
     }
}
```

We now define an equivalence predicate on cckells.

```
pred isoCCKell(c:CCKell, c1:CCKell) {
   isoCKell[c,c1]
   c.cctrl = c1.cctrl
   c.internals = c1.internals
   c.subcomps = c1.subcomps
}
```

We specify below the different operations attached to a ContentController gate. Operation getInternalInterfaces returns the set of internal interfaces of the cckell which hosts the target ContentController gate. Operation getInternalInterface returns the internal gate

whose identifier iid is passed as argument. Operation getSubComponents returns the set scc of subkells accessible via the ContentController gate. Operation addSubComponent adds a ckell designated by its Component gate icc to the set of subcomps of the host cckell. Operation removeSubComponent does the reverse.

```
// Operations from the ContentController interface

pred getInternalInterfaces(c:CCKell, cc:ContentController, sg: set Gate, c1:Kell) {
    cc = c.cctrl
    sg = c.internals
    isoCCKell[c,c1]
}

pred getInternalInterface(c:CCKell, cc:ContentController, iid: Id, ig: Gate, c1:Kell) {
    cc = c.cctrl
    ig.gid = iid
    ig in c.internals
    isoCCKell[c,c1]
}

pred getSubComponents(c:CCKell, cc:ContentController, scc: set Component, c1:Kell) {
    cc = c.cctrl
    scc = (c.subcomps).comp
    isoCCKell[c,c1]
}

pred addSubComponent(c:CCKell, cc:ContentController, icc: Component, r: ContentReturn, c1:Kell) {
  some scc:CKell {
    cc = c.cctrl
    icc = scc.comp
    r = IllegalLifeCycleException implies c1 = c
    else icc in c.subcomps.comp implies r = IllegalContentException and c1 = c
    else r = IllegalContentException implies c1 = c
    else r = Ok and c1.kid = c.kid and c1.comp = c.comp and c1.cctrl = c.cctrl and scc in c1.subcomps
  }
}

pred removeSubComponent(c:CCKell, cc:ContentController, icc:Component, r:ContentReturn, c1:CCKell) {
   cc = c.cctrl
   r = IllegalLifeCycleException implies c1 = c
   else no icc & c.subcomps.comp implies r = IllegalContentException and c1 = c
   else r = IllegalContentException implies c1 = c
   else some scc: c.subcomps {
            icc = scc.comp and
            r = Ok and
            c1.cctrl = c.cctrl and c1.comp = c.comp and c1.kid = c.kid and
            no scc & c1.subcomps
        }
}

one sig ContentReturn in Ok + IllegalContentException + IllegalLifeCycleException {}
```

Finally we give some consistency properties on operations.

```
assert RemoveAfterAddIsPossible {
   all c,c1: CCKell, icc:Component {
       (addSubComponent[c,c.cctrl,icc,Ok,c1] and
        c1.gates = c.gates and
        some scc:CKell {scc.comp = icc and c1.subcomps = c.subcomps + scc} ) implies
           removeSubComponent[c1,c1.cctrl,icc,Ok,c]
   }
```

```
}


assert AddAfterRemoveIsPossible {
    all c,c1: CCKell, icc:Component {
        (removeSubComponent[c,c.cctrl,icc,Ok,c1] and
         c1.gates = c.gates and
         some scc:CKell {scc.comp = icc and c1.subcomps = c.subcomps - scc} ) implies
            addSubComponent[c1,c1.cctrl,icc,Ok,c]
    }
}

assert GetSubCompSucceedsAfterAdd {
    all c,c1: CCKell, icc:Component {
        addSubComponent[c,c.cctrl,icc,Ok,c1] implies
            (getSubComponents[c1,c1.cctrl,c1.subcomps.comp, c1] and icc in c1.subcomps.comp)
    }
}

//
// The following property does not hold.
// Because of the weak conditions on removeSubComponent,
// it may well be that a component with the same Component gate cc
// exists as a subcomponent of a component from which a component
// with Component gate cc has just been removed.
//

assert GetSubCompFailsAfterRemove {
    all c,c1: CCKell, icc:Component {
        removeSubComponent[c,c.cctrl,icc,Ok,c1] implies no icc & c1.subcomps.comp
    }
}
```

11.10 Lifecycle controller

The LifecycleController interface in the Fractal model provides basic capabilities to control the execution of a component. The execution of a component from the point of view of this LifecycleController is abstracted as evolving between two macro-states, Started and Stopped. We follow here the informal Fractal specification, [26] Section 4.5.

We specify first these two macro-states.

```
module fractal/content

open util/relation as RR
open fractal/foundations as FF

sig LFState extends Val {}
one sig Started extends LFState {}
one sig Stopped extends LFState {}
```

We then define the set LFKell that offer a LifecycleController gate. The feature ctrls identifies the set of "control" gates, i.e. those gates whose operations are not inhibitetd when in the Stopped state.

```
sig LifeCycleController extends Server {}
```

```
sig LFKell in Kell {
      lfctrl: LifeCycleController,
      state: LFState,
      ctrls: set Gate
}

fact LFCtrlIsACntrlGate { all c:LFKell | c.lfctrl in c.ctrls }

fact CtrlGatesAreInGates { all c:LFKell | c.ctrls in c.gates }

fact LFStateIsStoppedOrStarted { all c:LFKell | c.state in Started + Stopped }
```

We define first an equivalence predicate between kells with LifecycleController gates.

```
pred isoLFKell(c:LFKell, c1:LFKell) {
   isoKell[c,c1]
   c.lfctrl = c1.lfctrl
   c.state = c1.state
   c.ctrls = c1.ctrls
}
```

We specify below the operations attached to LifecycleController gates. Operation getState returns the macro-state s of the kell hosting the target LifecycleController gate lfc. Operation start places the kell c hosting the target LifecycleController gate lfc into the Started macro-state. This may imply all sorts of changes in c, hence the weak constraint on the resulting kell c1: it has the same identifier than c, and the same LifecycleController interface, and it is in the Started macro-state.

```
// Operations from the LifecycleController interface

pred getState(c:LFKell, lfc:LifeCycleController, s:LFState, c1:LFKell) {
     c.lfctrl = lfc
     s= c.state
     isoLFKell[c,c1]
}

pred start(c:LFKell, lfc:LifeCycleController, r:LFReturn, c1:LFKell) {
      c.lfctrl = lfc
      r = IllegalLifeCycleException implies c1 = c
      else c.state = Started implies r = Ok and c1 = c
      else c.state = Stopped and r = Ok and c1.state = Started and c1.kid = c.kid and c1.lfctrl = c.lfcrtrl
}

pred stop(c:LFTKell, lfc:LifeCycleController, r:LFReturn, c1:LFTKell) {
      c.lfctrl = lfc
         r = IllegalLifeCycleException implies c1 = c
         else c.state = Stopped implies r = Ok and c1 = c
         else c.state = Started and r = Ok and c1.state = Stopped and c1.kid = c.kid and c1.lfctrl = c.lfcrtrl
}
```

Unfortunately, in this instance, the exact semantics of the Started and Stopped states, and hence of the start and stop operations, can only be given by reference to the behavior of the hosting kell. We specify below this semantics, exploiting the notion of transition. Essentially, the Stopped state is defined as one where no transition involving signals targetting non control gates is possible.

```
sig LFTKell in LFKell {}
```

```
fact LFTKellsAreTKells { all c: LFTKell | c in TKell }

fact LFTKellStoppedHasNoFunctionalTransitions {
    all c:LFTKell | all t:c.transitions | c.state = Stopped implies t.(sin+sout).target in c.ctrls
}
```

11.11 Future work

This report formalizes the programming language independent programming model that is at the basis of much the work in WP2 and WP4 of the Selfman project. We have started expanding this specification to cover the Kompics model reported in Chapter 8 of this book, so as to formally describe the event-based execution model that Kompics embodies. This work will find its place in a revised version of this deliverable.

Chapter 12

D3.1b: Second report on formal models for transactions over structured overlay networks

12.1 Executive Summary

Application developers using a storage system such as a relational database or a file-system requires well-defined semantics for reading and writing data. In a database storage layer this also include transaction functionality where read and write over multiple data entries are Atomic, Consistent, Isolated and Durable (ACID). We aim to develop a self-managing and scalable storage layer supporting transactions based on Structured Overlay Networks and DHTs. A system with these properties will enable applications with higher requirements on data consistency and transaction support. An example is the Wikipedia demonstrator presented in D5.2a.

In this deliverable we present a transaction algorithm suitable for DHTs based on the Paxos transaction commit protocol [64, 107]. Transaction algorithms for DHTs are particularly challenging due to the issue of lookup inconsistencies [132]. Empirical studies showed [131] that by using quorum-based algorithms the effects of lookup inconsistencies are negligible.

12.2 Partners Contributing to the Deliverable

ZIB (P5) and KTH (P2) have contributed to this deliverable.

ZIB (P5) ZIB has contributed on the transaction model and the DHT consistency model. The largest focus during the report period was to finalize the transaction model A.12 and to apply the developed techniques for the wiki demonstrator.

KTH (P2) KTH contributed on the transaction model as well as leading the work on the consistency model. The deliverable summarizes the results from the paper [131] and [132], included as Appendix A.14 and A.13.

12.3 Results

A SON-based DHT is a self-managing storage layer providing basic item manipulation primitives such as *put(key, value)*, for inserting a new (key, value)-pair and *get(key)*, for retrieving a value associated with a given key [128]. Traditionally, DHTs have been used by applications with immutable state or weak consistency guarantees. Applications with higher requirements on data consistency and interface flexibility are increasingly demanding easier to manage and more scalable storage layers than what current systems can provide [41, 110].

Self-management implies that the storage layer must deal with SON node failures. The consequence of a node failure is two-fold. First, the items stored as part of the nodes responsibility range are not available. Second, as discussed in 12.3.1, there may become an inconsistency of responsibilities. The first problem is solved by replicating items to more than one node, using e.g. symmetric replication [58]. The second issue was extensively studied in [132, 131], showing the frequency of occurrence of this problem and methods to remedy it.

The transaction processing framework initially presented in D3.1a enables updates and/or reads over multiple data items stored in a DHT. In this deliverable we expand this model and present an overview of the transaction algorithm. A detailed description of the algorithm is available in the Appendix A.12.

To demonstrate the use of transactions in DHTs, a wikipedia application was implemented on top of a DHT. In a wiki, users can concurrently make changes to the same entry. In order to avoid that any data is overwritten, an update to a wiki-page is wrapped in a transaction. When the user saves the changes, the transaction will detect any concurrent saves from other users. The user can then incorporate these changes in the text and try to save the wiki-entry again without any lost changes. The wiki-application demonstrator is further described in D5.2a.

We solve the problem of consistency at two levels, 1) routing-level (section 12.3.1), working mainly with the overlay routing pointers, and 2) data-level (section 12.3.2), working mainly on the DHT level with data.

12.3.1 Consistency on the Routing-Level

To achieve data-consisteny in DHTs with high probability, algorithms are required to achieve consistency on the routing level as well. In this section, we discuss the importance of having routing-level consistency and its affect on data consistency.

It is easy to see that even without concurrent operations, data consistency can be violated in DHTs due to lookup inconsistencies. Informally, a lookup inconsistency is a case where in an overlay configuration, multiple lookups for the same key return different results. Figure 12.1 illustrates such a configuration where lookups for key k can return inconsistent results. This configuration arises when, due to inaccuracy of the failure detector, $N1$ falsely suspects $N2$ and $N3$ as failed. Thus, $N1$ believes that the next (clockwise) alive node on the ring in $N4$, so it

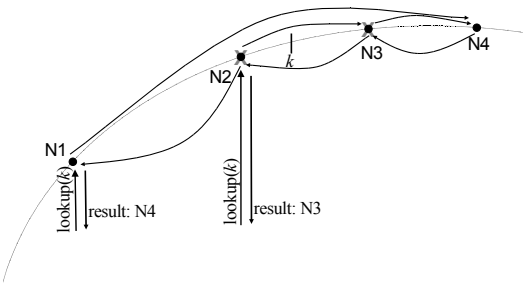


Figure 12.1: An inconsistent configuration. Due to imperfect failure detection, $N1$ suspects $N2$ and $N3$, thus pointing to $N4$ as successor.

points its successor pointer to $N4$. Subsequently, a lookup for key k ending at $N1$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N2$ would return $N3$.

In the scenario depicted in figure 12.1, an update for the data stored under key k will be stored at either $N3$ or $N4$. A read for data at k will return inconsistent/old results if it reaches the node that didn't receive the update.

The afore-mentioned scenario shows that an inconsistency on the routing-level leads to inconsistency on the data-level i.e. data inconsistency. Thus, as a first step to achieve data consistency in DHTs, we aim at achieving routing-level consistency.

In our work, first, we ran simulations to see the frequency of occurrence of lookup inconsistencies. We showed that even if there is no churn in the system, there will be lookup inconsistencies due to imperfect failure detectors. This is an important result as previous research mainly focuses on churn. Next, we devised two techniques to reduce lookup inconsistencies, 1) local responsibilities and 2) quorum-techniques.

The basic idea of using local responsibilities is to modify the lookup operation such that a lookup always returns from the locally responsible node. A node n is said to be locally responsible for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. Thus, before returning the result of a lookup, the node checks if it is locally responsible for the key being looked up. Using this technique reduces inconsistencies significantly, but has a side-effect of keys being unavailable. This can be seen as a trade-off between availability and consistency.

Since DHTs replicate data on different nodes to increase availability and prevent loss of data, we employ these replicas to increase consistency by using majority based quorum-techniques. Thus, a read/write operation has operate on a majority of the replicas. Due to lookup inconsistencies, a single replica might appear as two

replicas, thus changing the number of replicas in the system. Consequently, an operation can still be inconsistent, yet with lesser probability than without using quorum techniques. The reason being that previously, if there was a lookup inconsistency, it would generate inconsistent data. Using the afore-mentioned technique, even with lookup inconsistency, multiple quorums exist that intersect which eventually leads to data consistency.

The details of the work done on routing-level consistency as part of this deliverable was published as [131] and [132], both included in the appendix.

12.3.2 Transactional DHTs

With replication, concurrent *get/put*-operation to a single item could block the system if all replicas must be available. In order to make progress for each operation, this requirement is relaxed by using majority-based read and writes. Read or writes are then successful even if a minority of the replicas have failed. The replication factor decides how many replicas are supposed to be available in the system. This is a critical parameter for the system to function and must be set appropriately depending on system churn.

A transaction involves a read- and write-set over one or more items. Each transaction is executed optimistically. This means that if a transaction fails due to for example another concurrent transaction with an overlapping write/read-set, the full transaction must be re-executed by the initiator. With optimistic transactions, items are only locked during the commit phase of the transaction algorithm.

The node types involved in the execution of a transaction is the client, a set of TMs corresponding to the replication factor, a set of TPs consisting of the nodes responsible for the items in the read/write-set as well as the item replica nodes. The first node contacted by the client is part of the TMs and is called the *leader*.

Transaction algorithm The transaction processing consists of two phases: read and commit as shown in figure 12.2. During the read phase, the client initiates a leader. The leader maintains state of the read and write operations from the client and contact the TPs to find out the item values and versions. The client can either decide to commit a transaction if all intermediary conditions was satisfied, or cancel the transaction. If a client commits the transaction, the leader starts the Commit Phase. The client API for transactions is described in Deliverable D3.3a.

The commit phase contains three internal phases: Initialization, Validation and Consensus. During the Initialization phase, the leader finds all TMs which corresponds to the replica nodes responsible for the Transaction ID. The leader waits for a majority of TMs, which are necessary for the protocol to make progress.

If a leader fails, a new leader needs to be elected among the TMs. Thus, in first step of the Validation phase, the leader tells all TMs of all other TMs. The second step is to send a prepare message to all TPs, including the replicas, responsible

for an item. When a TP receives a prepare request it compares the item versions and if it is valid it sends a commit vote to all TMs, otherwise abort. A TP also locks the item in the transaction at this point. The involved items are locked until the TP finds out the results of the transaction. The sending of the vote message starts the Consensus Phase which follows the atomic commit protocol [64]. The TPs send their decisions to the TMs which then forwards the result to the leader. The leader collects all results and decides if the transaction succeeded or not. This result is then shared with the client, the TMs and the TPs. A detailed description of the algorithm is available in [107], which can also be found in the appendix.

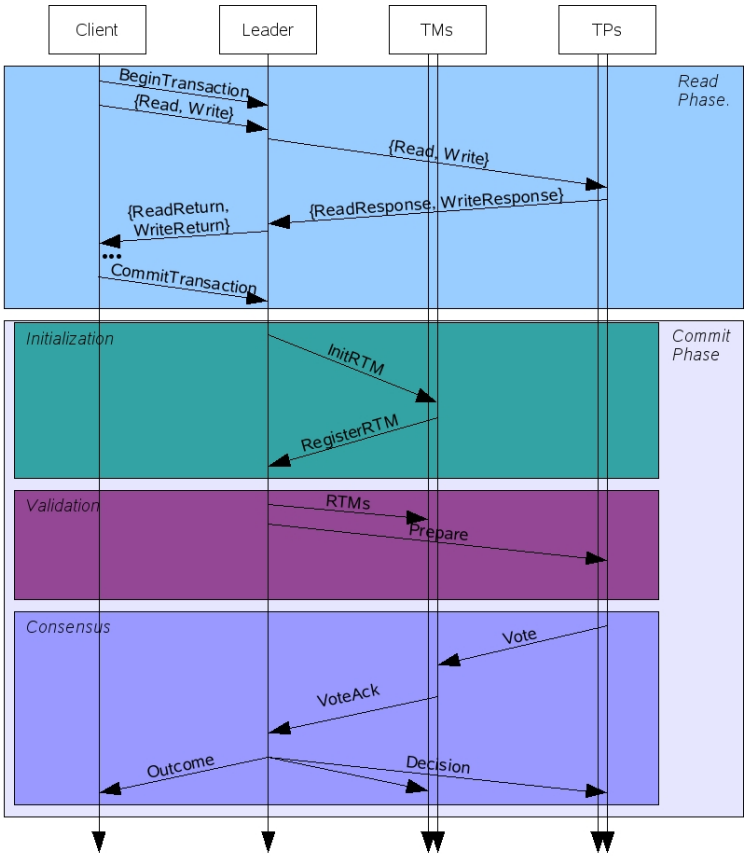


Figure 12.2: The different phases and message exchanges in a single instance of the transaction protocol.

12.4 Conclusion

Due to the dynamics and decentralization of DHTs and the asynchronous nature of the Internet on which DHTs are deployed, it is difficult to build abstractions with stronger consistency guarantees on top of DHTs. We propose using techniques on both the routing-level and the data-level to decrease data inconsistencies. Although it is impossible to guarantee consistency, availability and partition-tolerance in Internet-based systems [59], our results show that it is reasonable (consistency is maintained with high probability) to build reliable services on top of a DHT. As an application of a transactional storage service on top of a DHT, a distributed wikipedia was implemented (D5.2a).

While studying the factors contributing to inconsistencies in DHTs, we found that other than the obvious factor of churn, imperfect failure detectors contribute a lot to generating inconsistencies. Thus, choice of a failure detection algorithm, implementation of the failure detector and the trade-off between the accurary and completeness of a failure detector are of crucial importance in DHTs.

Chapter 13

D3.2a: Report on replicated storage service over a structured overlay network

13.1 Executive summary

The replicated storage service is based on Chord$^{\#}$and the transaction framework presented in D3.1b (see Chap. 12). It is completely developed in Erlang. The storage service can be accessed using a command line interface or from Java. The Java Interface is described in more detail in D3.3a (see Chap. 14).

13.2 Contractors contributing to the Deliverable

ZIB (P5) has contributed to this deliverable.

ZIB (P5) ZIB has contributed on the design and implementation of the replicated storage service.

13.3 Introduction

The replicated storage service is based on Chord$^{\#}$and the transaction framework presented in D3.1b (see Chap. 12). It is completely developed in Erlang and the architecture is based on three layers:

DHT layer At the bottom is a DHT, Chord$^{\#}$, which is described in great detail in Sec. A.3. For load-balancing, we use [81]. In the future we will investigate how to include other algorithms described in D4.3a (see Chap. 17).

It provides a simple key-value store with range queries and load-balancing.

Replication Layer The middle layer implements a simple replication scheme based on symmetric replication [58]. For Chord$^{\#}$, we use different prefixes to identify the different replicas and the load-balancing scheme will distribute the data in a way which is similar to symmetric replication. The replication degree as well as the prefixes can be specified before startup in the configuration file.

Transaction Layer On top of the replication layer, we implemented the transaction algorithms described in D3.1b (see Chap. 12). Further details and evaluations can be found in [107, 131, 132].

The storage service can be accessed using a command line interface or from Java. The Java Interface is described in more detail in D3.3a (see Chap. 14).

13.4 Installation and Configuration

Source Code is available at `http://www.zib.de/schuett/chordsharp-selfman.tgz`. Note, that this is an internal SelfMan relase.

Chord# Directory Structure. The directory tree under chordsharp is structured as follows:

`src`	contains the Chord# source code
`bin`	contains shell scripts needed to work with Chord# (e.g. start the boot services, start a node, ...)
`docs`	contains Chord# documentation files
`java-api`	contains the Java API

13.4.1 Requirements

For building and running Chord#, some third-party modules are required which are not included in the Chord# release:

- Erlang R12
- GNU Make
- rrdtool

Note, the Version 12 of Erlang is required. Chord# will not work with older versions.

To build the Java API the following modules are required additionally:

- Java Development Kit 1.6
- Apache Ant

Before building the Java API, make sure that `JAVA_HOME` and `ANT_HOME` are set. `JAVA_HOME` has to point to a JDK 1.6 installation, and `ANT_HOME` has to point to an Ant installation.

13.4.2 Building Chord#

Go into the chordsharp directory and execute:

```
%> ./configure
%> make
%> make rrd-init
%> make docs
```

The `configure` script will probably note, that you are missing common_test. You can ignore this message, as the module is not required for the correct execution of Chord#.

13.4.3 Installation

Note: there is no `make install` at the moment! The nodes have to be started from the bin directory.

13.4.4 Configuration

Chord# is configured by two configuration files (`bin/chordsharp.cfg` and `bin/chordsharp.local.cfg`). It will read the former for default values and then the latter which can override the defaults. After going through the build process there will be no `chordsharp.local.cfg`. It has to be created by the user, because there are two configuration parameters which have no default value: `boot_host` and `log_host`, and Chord# won't start if the file is missing.

%IP Address, Port, and label of the boot server

```
{boot_host, {{130,73,72,80},14195,boot}}.
```

%IP Address, Port, and label of the log server

```
{log_host, {{130,73,72,80},14195,boot_logger}}.
```

`boot_host` defines the node where the boot server is running.

%possible values: 14195, [14195, 14196, 14197](list of ports), or 14195, 15000 range of ports

```
{listen_port, 14195}.
```

%undefined or an ip tuple, e.g. 130.73.108.1

```
{listen_ip, undefined}.
```

13.5 User Guide

13.5.1 Starting Chord#

In Chord# there are two kinds of processes:

- boot servers
- regular servers

In every Chord#, at least one boot server is required. It will maintain a list of nodes taken part in the system and allows other nodes to join the ring. For redundancy, it is also possible to have several boot servers.

Open at least two shells. In the first, go into the bin directory:

```
%> cd bin
%> ./boot.sh
```

This will start the boot server. On success http://localhost:8000 should point to the statistics page of the boot server. The main page will show you the number of nodes currently in the system. After a couple of seconds a first Chord# should have started in the boot server and the number should increase to one. The main page will also allow you to store and retrieve key-value pairs.

The boot server should show output similar to the following, when starting the first Chord# nodes. The first line is printed when the Chord# is spawned. Afterwards he will try to connect the boot server. When the third line is printed, he managed to contact the boot server and joined the ring. In this case, he was the first node in the ring.

```
[ I | Node   | <0.97.0> ] joining "23947834870"
[ I | Node   | <0.97.0> ] join as first [50,51,57,52,55,56,51,52,56,55,48]
[ I | Node   | <0.97.0> ] joined
```

In a second shell, you can now start a second Chord# node. This will be a "regular server". Go in the bin directory:

```
%> cd bin
%> ./cs_local.sh
```

The second node will read the configuration file and use this information to contact the boot server and will join the ring. The number of nodes on the web page should have increased to two by now.

Optionally, a third and fourth node can be started on the same machine. In a third shell:

```
%> cd bin
%> ./cs_local2.sh
```

In a fourth shell:

```
%> cd bin
%> ./cs_local3.sh
```

This will add 3 nodes to the network. The web pages should show the additional nodes.

Chord# can be installed on other machines in the same way as described in Sect. 13.4. Please make sure, that the chordsharp.local.cfg is the same on all nodes. Otherwise the other nodes will not find the boot server. On the remote nodes, you only need to call `./cs_local.sh` they will automatically contact the configured boot server.

13.5.2 Java-API

The following commands will build the Java API for Chord#:

```
%> cd java-api
%> ant
```

This will build chordsharp4j.jar, which is the library for accessing the overlay network. Optionally, the documentation can be build:

```
%> ant doc
```

The jar file additionally contains a small cli client.

```
%> java -jar chordsharp4j.jar -help
usage: chordsharp
 -getsubscribers <topic>   get subscribers of a topic
 -help                     print this message
 -publish <params>         publish a new message for a topic: <topic>
                           <message>
 -read <key>               read an item
 -subscribe <params>       subscribe to a topic: <topic> <url>
 -write <params>           write an item: <key> <value>
```

Read and write can be used to read resp. write from/to the overlay. getsubscribers, publish, and subscribe are the PubSub functions.

```
%> java -jar chordsharp4j.jar -write foo bar
write(foo, bar)
%> java -jar chordsharp4j.jar -read foo
read(foo) == bar
```

The chordsharp4j library requires that you are running a “regular server” on the same node. Having a boot server running on the same node is not sufficient.

Chapter 14

D3.3a: Simple database query layer for replicated storage service

14.1 Executive summary

The "Simple database query layer" is a small Java library for accessing the replicated storage service described in D3.2a (see Chap. 13). It provides functions for reading and writing key-value pairs. Several read resp. write requests can be executed within a transaction.

14.2 Contractors contributing to the Deliverable

ZIB (P5) has contributed to this deliverable.

ZIB (P5) ZIB has contributed on the API design, implemented the Java API and developed the Java to Erlang interface.

14.3 Introduction

The "Simple data query layer for replicated storage service" is a Java API for accessing data stored in the replicated storage service presented in D3.2a (see Chap. 13). The storage service has a native interface for accessing the storage, which allows to specify transactions in Erlang. This interface is very similar to the API for mnesia, an Erlang database. For Java users, we developed a more traditional interface.

Simple API The simple API allows to read and write key-value pairs. The respective read and write operations are executed within a transaction and the replicas are accessed with strong consistency, however each transaction will contain exactly one operation – read or write. The functions are provided by the `ChordSharp` class.

Transactions The `Transaction` class provides a more powerful interface, as several operations can be executed within one transaction.

For both interfaces, we use JInterface, which is a Java library which can send message to Erlang VMs. The communication between the Java VM and the Erlang VM is using the native Erlang protocol and the `Transaction` resp. `ChordSharp` class provide a wrapper around JInterface to make it more usable for Java programmers.

14.4 API

14.4.1 de.zib.chordsharp.ChordSharp

public class ChordSharp

Public ChordSharp Interface.

Version: 1.1

Author: Nico Kruber, kruber@zib.de

Method Summary	
static Vector<String>	**getSubscribers**(String topic) Gets a list of subscribers of a topic.
static void	**publish**(String topic, String content) Publishes an event under a given topic.
static String	**read**(String key) Gets the value stored with the given key.
static void	**subscribe**(String topic, String url) Subscribes a url for a topic.
static void	**write**(String key, String value) Stores the given key/value pair.

read

public static String read(String key)
throws ConnectionException,
TimeoutException,
UnknownException,
NotFoundException

Gets the value stored with the given key.

Parameters: key - the key to look up

Returns: the value stored under the given key

Throws:

ConnectionException	if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie
TimeoutException	if a timeout occurred while trying to fetch the value
NotFoundException	if the requested key does not exist
UnknownException	if any other error occurs

write

```
public static void write(String key,
                         String value)
                  throws ConnectionException,
                         TimeoutException,
                         UnknownException
```

Stores the given key/value pair.
Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

publish

```
public static void publish(String topic,
                           String content)
                    throws ConnectionException
```

Publishes an event under a given topic.
Parameters: topic - the topic to publish the content under content - the content to publish

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

subscribe

```
public static void subscribe(String topic,
                             String url)
                      throws ConnectionException
```

Subscribes a url for a topic.
Parameters: topic - the topic to subscribe the url for url - the url of the subscriber (this is where the events are send to)

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

getSubscribers

```
public static Vector<String> getSubscribers(String topic)
                                    throws ConnectionException,
                                           UnknownException
```

Gets a list of subscribers of a topic.
Parameters: topic - the topic to get the subscribers for

Returns: the subscriber URLs

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie UnknownException - is thrown if the return type of the erlang method does not match the expected one

14.4.2 de.zib.chordsharp.Transaction

```
public class Transaction
```

Provides means to realise a transaction with the chordsharp ring using Java.

It reads the connection parameters from a file called ChordSharpConnection.properties or uses default properties defined in ChordSharpConnection.defaultProperties.

```
OtpErlangString otpKey;
OtpErlangString otpValue;
OtpErlangString otpResult;
String key;
String value;
String result;

// Transaction()
Transaction transaction = new Transaction();
// start()
transaction.start();
// write(OtpErlangString, OtpErlangString)
transaction.write(otpKey, otpValue);
// write(String, String)
transaction.write(key, value);
//read(OtpErlangString)
otpResult = transaction.read(otpKey);
//read(String)
result = transaction.read(key);
// commit()
transaction.commit();
```

For more examples, have a look at TransactionReadExample, TransactionParallelReadsExample, TransactionWriteExample and TransactionReadWriteExample.

Attention:

If a read or write operation fails within a transaction all subsequent operations on that key will fail as well. This behaviour may particularly be undesirable if a

read operation just checks whether a value already exists or not. To overcome this situation call revertLastOp() immediately after the failed operation which restores the state as it was before that operation.

The TransactionReadWriteExample example shows such a use case.

Version: 1.0

Author: Nico Kruber, kruber@zib.de

Constructor Summary
Transaction() Creates the object's connection to the chordsharp node specified in the "ChordSharpConnection.properties" file.

Method Summary	
void	**abort**() Cancels the current transaction.
void	**commit**() Commits the current transaction.
OtpErlangString	**read**(OtpErlangString key) Gets the value stored under the given key.
String	**read**(String key) Gets the value stored under the given key.
void	**revertLastOp**() Reverts the last (read or write) operation by restoring the last state.
void	**start**() Starts a new transaction by generating a new transaction log.
void	**write**(OtpErlangString key, OtpErlangString value) Stores the given key/value pair.
void	**write**(String key, String value) Stores the given key/value pair.

Transaction

```
public Transaction()
            throws ConnectionException
```

Creates the object's connection to the chordsharp node specified in the "ChordSharpConnection.properties" file.

Throws: ConnectionException - if the connection fails

start

```
public void start()
           throws ConnectionException,
                  TransactionNotFinishedException,
                  UnknownException
```

Starts a new transaction by generating a new transaction log.

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TransactionNotFinishedException - if an old transaction is not finished (via commit() or abort()) yet UnknownException - if the returned value from erlang does not have the expected type-/structure

commit

```
public void commit()
        throws UnknownException,
               ConnectionException
```

Commits the current transaction. The transaction's log is reset if the commit was successful, otherwise it still retains in the transaction which must be successfully committed or aborted in order to be restarted.

Throws: UnknownException - If the commit fails or the returned value from erlang is of an unknown type/structure, this exception is thrown. Neither the transaction log nor the local operations buffer is emptied, so that the commit can be tried again. ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

See Also: abort()

abort

```
public void abort()
```

Cancels the current transaction.

For a transaction to be cancelled, only the transLog needs to be reset. Nothing else needs to be done since the data was not modified until the transaction was committed.

read

```
public OtpErlangString read(OtpErlangString key)
                     throws ConnectionException,
                            TimeoutException,
                            UnknownException,
                            NotFoundException
```

Gets the value stored under the given key.

Parameters: key - the key to look up

Returns: the value stored under the given key

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to fetch the value NotFoundException - if the requested key does not exist UnknownException - if any other error occurs

read

```
public String read(String key)
            throws ConnectionException,
                   TimeoutException,
                   UnknownException,
                   NotFoundException
```

Gets the value stored under the given key.
Parameters: key - the key to look up

Returns: the value stored under the given key

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to fetch the value NotFoundException - if the requested key does not exist UnknownException - if any other error occurs

write

```
public void write(OtpErlangString key,
                  OtpErlangString value)
           throws ConnectionException,
                  TimeoutException,
                  UnknownException
```

Stores the given key/value pair.
Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

write

```
public void write(String key,
                  String value)
           throws ConnectionException,
                  TimeoutException,
                  UnknownException
```

Stores the given key/value pair.
Parameters: key - the key to store the value for value - the value to store

Throws: ConnectionException - if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie TimeoutException - if a timeout occurred while trying to write the value UnknownException - if any other error occurs

revertLastOp

public void revertLastOp()

Reverts the last (read, parallelRead or write) operation by restoring the last state. If no operation was initiated yet, this method does nothing.

This method is especially useful if after an unsuccessful read a value with the same key should be written which is not possible if the failed read is still in the transaction's log.

Chapter 15

D4.1a: First report on self-configuration support

15.1 Executive summary

The work on self-configuration support in the second year covered mostly the development of an Oz-based framework (called FructOz) for the construction of self-deployable and self-configurable components. This work builds on three earlier developments by the Selfman partners:

- The Oz/Mozart distributed programming environment, which is used for supporting distributed deployment and configuration processes, and for integrating self-deployment and self-configuration capabilities within component packages.
- The Fractal component model, which provides the basic concepts and structures for defining self-configurable components, including basic introspection and navigation capabilities.
- The FPath language for navigating and querying Fractal architectures (an analog of the XPath language for querying XML documents).

The development of FructOz was motivated primarily by two objectives:

1. To provide basic support for complex deployment and configuration processes, including the definition of potentially complex workflows as exhibited e.g. by large enterprise-wide software depployments.
2. To provide basic support for embedding deployment and configuration capabilities within component themselves, as a first step towards the construction of distributed and self-configurable components.

The deliverable presents in more details the motivations and objectives for the work, focusing in particular on shortcomings of existing technology to support complex deployment and configuration processes. After a presentation of related work, the deliverable introduces briefly the language-independent reference model which we use as our baseline. The reference model introduces the notion of component package as a unit of executable software installation, and defines a minimal structure required for performing local, i.e. node level, component deployment. The deliverable then presents the FructOz framework, which exploits the Oz environment as a partial implementation of the reference model, and which consists of two parts:

- A lightweight implementation of the Fractal model in Oz, and an implementation of component packages as Oz functors.
- An implementation of a dynamic variant of the FPath query language, which allows to automate the generation of non-trivial monitoring predicates on a distributed component implementation (reified as a Fractal architecture).

To illustrate how our technical requirements for supporting complex deployment processes are met with the FructOz framework, we then present a set of examples, including the deployment of a cluster-size system, and the construction of a self-configurable component, which continuousy monitors itself and can react to changes in its internal configuration.

The deliverable also presents some preliminary evaluation of the FructOz framework, in the form of comparative micro-benchmarks for local deployment, and of a comparison of the performance of different deployment processes built with FructOz.

The deliverable ends with a discussion of the limitations of the FructOz framework and of future work. To support the claim that FructOz provides a way to handle complex deployment and configuration processes, we have added as a supplement material documenting how well-known control-flow workflow patterns can be supported in Oz.

15.2 Contractors contributing to the deliverable

INRIA contributed to this deliverable, with the help of UCL for developing the FructOz framework and for carrying out the preliminary experiments reported in this deliverable.

15.3 Motivations

Deploying and configuring a distributed software system can be a complex process, involving multiple distributed activities. This complexity is well illustrated by papers and specifications which document deployment and configuration activities. For instance, Hall et al. [69], characterize software deployment as a collection of interrelated activities (such as release, install, adapt, reconfigure, update, activate, deactivate, remove and retire); Coupaye et al. [36], extend this analysis to activities involved in the enterprise-wide deployment of large software applications (including activities such as assemble, install, and activate, which subsume or encompass those identified in [69]); standards such as the OMG specification for the deployment and configuration of component-based applications [65], which identifies a number of activities in a deployment process (including installation, configuration, planning, preparation, and launch).

A general approach for dealing with this complexity has been proposed by van der Hoek [143], under the term architecture-based deployment and configuration. Roughly, the main thrust of the approach is to exploit software architecture descriptions (possibly extended with domain-specific annotations) to drive deployment and configuration activities. As already noted in [143], an architecture-based approach to deployment and configuration management has a number of benefits:

- A rise in abstraction level, which allows to encapsulate and deal uniformly with idiosyncrasies of system configuration and deployment in legacy and heterogeneous systems.
- A seamless integration between software configuration management activities and software deployment activities, limiting architectural erosion and enabling a more rapid development / deployment cycle.

Actually, these benefits carry over more generally in an architecture-based approach to distributed systems management, as demonstrated by works such as Rainbow [54], Automate [109], and our own Jade [22]. In this broader context, software architecture descriptions can serve as pivot information for multiple management functions, including fault management [134], performance management [23], and security management [33], and an architecture description language (ADL) can serve as a pivot language for supporting tools.

An architecture-based approach to configuration management and deployment has been pursued in a number of different works, including e.g. [6, 7, 11, 28, 30, 37, 49, 72, 73, 87, 90, 105, 104, 112].

In these works, the deployment process is either an ad-hoc algorithm or task framework operating on descriptions of mostly static software architectures, as in [6, 28, 49, 83, 87, 90, 112], or generated by a constraint solving algorithm or an automated AI planner, that interprets software descriptions with deployment constraints as a goal, as in [11, 37, 73, 104]. The level of support for the deployment process that is provided in these different works is unsatisfactory, however.

Deployment in situations such as the enterprise environments envisaged by [36], must take into account multiple constraints, policies, synchronization and error conditions that require more sophisticated distributed coordination facilities than is provided by these different approaches, or than can currently be effectively handled by an AI planner approach. In particular, we expect support for deployment and (re)configuration processes to satisfy the following requirements, which are not adequately covered in the existing litterature:

1. Ability to define complex deployment workflows, including support for well-known control flow and exception patterns.

2. Ability to define parameterized and higher-order workflows for the concise specification of complex distributed architectures.

3. Ability to finely control activities involved in the deployment and (re)configuration of a distributed software architecture, and the moment they are triggered.

4. Ability to internalize deployment and (re)configuration activities in the description of a self-configurable software architecture.

Requirement 1 concerns the ability to directly support the main patterns of synchronization and exception handling that have been identified by the workflow community [142]. These patterns are useful yardsticks for the specification and programming of deployment and configuration processes for they embody higher-level abstractions and operators for process synchronisation and task composition which have proved useful in the design of enterprise-wide formal processes and on-line services.

Requirement 2 concerns the ability to define parametric deployment and configuration processes, and is key to support scalable and compositional definitions of deployment and configuration processes. Parameters of a deployment and configuration process may include, for instance, the size and structure of the target environment, or binding and interconnection schemas between deployed components as in multi-tier systems.

Requirement 3 concerns in particular the ability to delay the deployment and configuration of individual components up to the point where they are finally needed. This lazy deployment capability in turn supports different performance and adaptation tradeoffs.

Requirement 4 is a necessary step towards autonomic components and systems.

In this paper, we present an approach for the development of architecture-based deployment processes that meets the above requirements. It is based on the Fractal component model [25], and the Oz programming language and its Mozart distributed environment [144]. Specifically, we present an Oz framework, called FructOz, that can be used for the development of complex distributed deployment and (re)configuration workflows, and the development of self-configurable distributed components. FructOz can be understood as providing the basis for the

description of dynamic distributed software architectures, i.e. software architectures whose descriptions embody provisions for change and evolution, in response to events from their environment.

The paper makes the following contributions:

- We introduce a language-independent reference model for distributed deployment and configuration.
- We introduce the FructOz framework, written in the Oz programming language, that can be used for building complex distributed software architectures, including complex deployment and reconfiguration workflows.
- We illustrate via several examples how the above identified requirements are met by the use of the FructOz framework and the Oz programming language.
- We report preliminary performance results that demonstrate the viability and the potential benefits of the FructOz approach to distributed deployment and configuration.

15.4 Related work

The reference model presented in Section 15.5 builds on several works, including the consideration of general component dependencies as in [87], a notion of local installation store inspired from the Nix system [46], and a notion of component package analogous to that of the Edos project [99]. Our notions of component and component packages are analogous to, but generalize the wiring notions of Assemblages [96]; in particular, plugging and mixing notions in Assemblages are examples of our notion of binding between components and component packages. Our reference model does not attempt to identify formally different activities or phases involved in a deployment process e.g. as in [69, 65]. Not only are these different phases potentially environment and application dependent, but our examples on lazy deployment and on self-configurable components shows that such distinctions can be elusive.

Previous approaches for supporting complex distributed deployment processes fall roughly into four categories: (i) approaches that rely on a fixed deployment algorithm or more extensive task framework, driven by (static) software architecture descriptions, including [6, 7, 28, 30, 42, 49, 83, 85, 87, 90, 105, 112, 153]; (ii) approaches that rely on a workflow language for describing deployment processes, including [8, 63, 82, 122, 141]; (iii) approaches that rely on a constraint solving algorithm for determining a deployment target, including [16, 104, 98, 139]; (iv) approaches that rely on an AI planner for generating deployment processes, including [11, 73, 84].

Approaches that rely on fixed deployment algorithm or a supporting framework, do not meet the requirements we have identified in the introduction. How-

ever, the work reported in this paper could be exploited in a number of the frameworks referenced above. In particular, the work in this paper complements our previous work on component deployment and configuration in Jade [6], and would be directly usable in the FDF [49] framework.

Approaches that rely on constraint solving and AI planning are interesting for their high degree of declarativity, but they do not provide sufficient support for dealing with complex deployment workflow, and to support effectively our four requirements above. Approaches based on constraint-solving tend to focus on specific deployent problems and objectives (e.g. such as ensuring a certain degree of availability [98]). They must be complemented with additional mechanisms to deal correctly with exceptional conditions or to ensure synchronization conditions which are not enforceable within their scheduling framework. Approaches based on AI planning are interesting because they have the potential to automate the generation of complex deployment workflows, but it is not clear at this point that they can be employed successfully beyond simple system configurations. There is certainly promising work for integrating AI planning techniques with workflow management systems [118] but issues with respect to expressivity and scalability are still open.

Approaches that rely on a workflow language are closer to ours. They include the SmartFrog system [8, 63], the Workflakes system [141], the Andrea system [122], and the use BPWS4J workflow engine with the IBM Tivoli deployment engine for the provisioning of application services [82]. Compared to our approach, which relies on the coordination and distributed programming capabilities of the Oz programming language, these systems do not provide direct support for parameterized, higher-order workflows (requirement 2 – see our case study 15.7.1), and they do not provide direct support for lazy execution (requirement 3 – see our case study 15.7.3). Also the use of a workflow or process management engine remains essentially external to the components used or the system being managed, and no provision has been made for the construction of self-deployable components and self-configurable hierarchical components (requirement 4 – see our case study 15.7.5). This is also the case with the SmartFrog system, even though its workflow constructs take the form of SmartFrog components, because these are used for the structuring of the initial deployment process. Note that, in our approach, even if components are not programmed in Oz, internalizing a complex deployment and configuration behavior is made simpler by the reflective character of the Fractal model. We could, for instance, make use of the aspect-oriented capabilities of the reference implementation of Fractal in Java, to program advices interfacing directly to the (meta-level) deployment and configuration behavior written in Oz.

Our work is also related to architecture description languages for dynamic architectures, i.e. ADLs that can describe dynamically evolving architectures. A recent survey of such ADLs can be found in [24]. The survey shows that the subject of ADLs for dynamic architectures is well researched but that none of the surveyed approaches provided support for unconstrained evolution. Interest-

ingly, this can be traced to the fact that none of the surveyed approaches (such as Dynamic Wright, Darwin, etc) are higher-order, e.g. none can specify the receipt of a new component from the environment, which is not already specified in the original architecture description. Although some of the works surveyed such as Darwin [56] provide a supporting distributed infrastructure, none deal directly with deployment issues. Another ADL for dynamic architectures is Plastik [78], which benefits from a supporting infrastructure that provides a causal connection between architecture descriptions and the supporting OpenCOM component model [35]. The Plastik infrastructure provides basic support for local deployment, through its loader component, and supports reconfiguration scripts coded with the Lua programming language. Plastik provides good support for local reconfiguration but does not provide direct support for our requirements above.

15.5 Reference model

The architectural background for our work takes the form of a programming-language-independent reference model. It consists of two main elements: a component model, and local node structure. The component model is a refinement of the Fractal component model to make explicit notions of software component packages and their dependencies. The local node structure identifies a set of abstractions and functions for local installation and deployment.

The component model we adopt as our basis is the Fractal component model [25]. Fractal is a general component model which builds on classical software architecture concepts as captured e.g. in ACME [55]: components as encapsulated data and behavior, which clearly expose their depencencies and connections to their environment via interfaces (entry points to components, which generalizes the port notion), and bindings (interrelations between components, which generalizes the connector notion). To this classical foundation, Fractal adds the following elements:

- Bindings can be reified as components, in particular to build them by composition.
- Components can be endowed with meta-level behavior, made explicit via so-called controller interfaces.
- Components can be shared between multiple composite components, i.e. the containment relation is in general a directed acyclic graph of components and not necessarily a tree.
- Fractal does not impose a predefined semantics for meta-level behavior, component containment and bindings.

The general form of a component is that of a composite, with several subcomponents, and a membrane, that encompasses all the meta-level activity of the com-

ponent, as well as base level (or functional) behavior that is characteristic of the component (i.e. not delegated to its subcomponents). A component membrane can thus contain different controllers such as a content controller, which provides access to, and allows manipulation of subcomponents, a lifecycle controller, which provides control over the execution status of the component, an attribute controller, which provides access to data associated with the component, etc.

A Fractal component is a run-time entity, i.e. whose presence is manifest during execution. Deploying and configuring a Fractal application informally implies: installing the appropriately configured executables (e.g. source code, binaries) at chosen nodes (i.e. physical or virtual machines), creating and activating the necessary components (including the necessary bindings constituting interaction pathways with other components).

In our reference model, executables are made explicit and manipulated by means of component packages. A component package (or package for brevity) is a bundle that contains executables necessary for creating run-time components, as well as data needed for their correct functioning, and metadata describing their properties and requirements with respect to the target environment. A package is itself a Fractal component, which means it can contain other packages, be shared between multiple containing packages, provide certain interfaces, require other interfaces from its environment. A containment relation between packages corresponds to a requirement dependency between packages: the composite package requires the presence of the sub-package in the target environment. As with general Fractal components, different qualifications (such as mandatory, optional, or lazy) may apply to sub-packages and to required interfaces. As for component attributes in general, our model does not enforce a specific set of metadata to associate with a package. As a minimum, a package metadata must identify mandatory qualifications for package dependencies. A package metadata may contain additional requirements, such as version information, resource requirements, or conflicts (identifying that certain packages, or components in general, must not be present in the target environment). Note that there are two aspects of configuration involved in our model: the first one is related to package resolution, and the second one corresponds to setting appropriate activation parameters and attributes, and establishing necessary bindings between run-time components.

A distributed deployment and configuration process for Fractal systems, can be understood as a distributed application executing on a set of nodes. Nodes that form the targets of the deployment process are called managed nodes. To effect deployment and configuration, managed nodes must be equipped with a minimal structure. This comprises, for each managed node:

- One or more binding factories for establishing bindings supporting remote communication with components residing on the managed node.
- An installation store component, which is a repository of packages.

- One or more loader components, which are responsible for loading executables from packages in the installation store.

At least one binding factory per managed node is required to enable communication with component interfaces located on managed nodes, and to transfer packages to managed nodes (either in a pull or a push mode). The installation store constitutes a local target environment for deployment, and provides the context for deciding whether a package is installable or not, i.e. whether all the mandatory dependencies of a package can transitively be resolved (see [99] for a formalization of – a form of – installability). The installation store component need not execute on the managed node it belongs to, but may be located on a different node (e.g. because of resource constraints). Loader components (loosely analogous to Java class loaders) can take many forms, performing strictly binary loads for execution, or engaging in run-time resolution of package dependencies, with installation of required packages in the local installation store. By definition, loader components execute on the managed node they belong to.

15.6 The FructOz framework

15.6.1 Overview

The FructOz framework allows Fractal developers to leverage the Oz programming language for the deployment and configuration of Fractal systems (be they programmed in Oz or in some other language). FructOz also allows the development of self-configurable and self-deployable distributed components, i.e. components whose implementation may span several nodes, and which can reconfigure themselves during execution, possibly proceeding to changes in subcomponents and in supporting component packages. FructOz currently comprises:

- A lightweight implementation of the Fractal model in Oz.
- A "dynamic FPath" library providing support for navigating, querying, and monitoring Fractal component structures.

The Fractal implementation which FructOz provides is "lightweight" in the sense that it does not provide all the features and capabilities of the reference Java implementation of Fractal. Also, we have not aimed at this point for a highly optimized implementation. The "dynamic FPath" library has been inspired by the FPath language [40], and allows navigating and querying a Fractal component structure, much like XPath allows to navigate and query XML documents. In our case, navigating and querying can take place in a component structure that spans multiple nodes.

FructOz can typically be complemented by a library implementing distributed control flow and exception patterns, such as e.g. those documented in the workflow

litterature [142, 125]. We present elements of such a library in Section 15.10. They take the form of operators that formalize the main control flow patterns identified in [124, 142]. We also provide some examples showing how to support the different exception patterns presented in [125].

FructOz provides a simple implementation of the reference model presented in Section 15.5. The construction of Fractal components is explained below. Component packages are constructed as Oz functors (a general form of module, analogous to functors in ML), with package dependencies corresponding to import clauses in functors. Loaders in FructOz correspond to Oz module managers, which support the resolution of import clauses in functors and return components (in the form of membrane constructs – see below). Loaders in FructOz are thus also component factories. The installation store is not reified in FructOz: it just corresponds to the Oz memory store since FructOz packages are plain Oz functors, and thus language values. Binding factories are not reified either: they just correspond to the communication capabilities of the Oz infrastructure.

We lack the space in this paper to provide a detailed introduction to the Oz language. A comprehensive reference is [144]. Tutorial material is available on the Oz/Mozart Web site [71]. To facilitate the understanding of program fragments below, here are a few indications:

- Variables in Oz are logic variables: they can be bound or unbound. An unbound variable holds no value. Once bound, i.e. once a value has been assigned to it, a variable is immutable. Variables in Oz programs are denoted by tokens that begin with an upper case letter. Assignment is denoted by an = sign. Thus: `X = V` denotes the assignment to variable `X` of some value `V`.

- Values in Oz can be integer, strings, atoms (denoted by tokens that begin with a lower case letter), records, cells, ports, or procedures. A record takes the following form `r(f1:X1 ... fn:Xn)` where `r` (the label of the record) and `f1 ... fn` (the fields of the record) are typically atoms, and `X1 ... Xn` are (bound or unbound) variables. Access to a record field is noted with a . sign (thus: `r(f1:X1 f2:X2).f1` returns `X1`). Special cases of records are pairs, noted with an infix # sign (thus: `X#Y` corresponds to the pair `(X,Y)`), and lists, noted with a | sign (thus: `H | T` denotes a list whose head is `H` and whose tail is `T`).

- A cell corresponds to a mutable reference. A cell content is a variable. Thus `@C` denotes the content of a cell `C`, and `C := X` (assignment) updates the content of cell `C` with variable `X`.

- A procedure call takes the form `{P A1 ... An}` where `P` is a procedure name, `A1 ... An` are variables denoting the arguments of the call. A procedure can update several of its (unbound) arguments, and thus return several results. A procedure declaration takes the form `proc {P X1 ... Xn} E end` where `P` denotes the name of the procedure, `X1 ... Xn` denote the formal arguments of the procedure, statement `E` corresponds to the body of the procedure. A function,

declared with a statement of the form **fun** {F X1 ... Xn} E end is a special case of procedure which returns a single result. Anonymous procedures and functions can be declared with the anonymous marker $.

- Variable declarations typically take the form X **in** ... or X = E **in** ..., where E is some statement.
- Sequences of two statements S1 and S2 is just written by the juxtaposition of the two statements, horizontally as S1 S2, or vertically.
- A new concurrent thread is spawned by a statement of the form thread S end, where S is the statement to be executed in parallel with the current thread.

The rest of this section describes in more details the main elements of the FructOz framework.

15.6.2 Interfaces and components

FructOz implements interfaces as Oz ports: "An Oz port is an asynchronous channel that supports many-to-one communication. A port P encapsulates a stream S. A stream is a list with unbound tail. The operation {Send P M} adds M to the end of S. Successive sends from the same thread appear in the order they were sent." (quotation from the Mozart/Oz tutorial). Considering a port more in details, the port itself constitutes the input of the interface, i.e. where clients address their messages, while the stream encapsulated in the port constitutes its output, i.e. where the implementation reads the messages it processes.

Actually, a FructOz interface combines a port with a cell (a mutable reference), so as to allow changing the implementation which processes clients' messages at runtime, thus featuring dynamic reconfigurations.

A FructOz component is represented by its membrane, a structure holding a mutable set of interfaces. Some interfaces are server interfaces and export a functionality outside the component, while others are client interfaces that import functionalities inside the component.

There is no distinction between primitive, composite or compound components in FructOz as in SmartFrog or Fractal/Julia (the reference implementation of Fractal in Java): a component may well implement and make direct use of some of its interfaces, and at the same time, have subcomponents bound to other interfaces.

```
%% Create a new empty component membrane
Comp = {CNew}

%% Create and add an interface to the component membrane
Itf = {INew server [/*tags list*/]}
{CAddInterface Comp Itf}
```

Note that there is no explicit representation for subcomponents or composition at this level. The functional content of a component is here implicitly defined as the set of components directly or indirecty bound to any internal interface side of the component.

15.6.3 Bindings

Interfaces can be bound with each other and, eventually, to some native code. Technically, a binding between two interfaces is an active pump implemented with an Oz thread waiting for messages on the output stream of one interface and forwarding these messages to the input port of the other interface. Though this representation of a binding by an Oz thread is probably not the most efficient, it is simple and affordable for our case studies since Oz threads are designed to be very light. Establishing a binding between two interfaces is realized through the BNew primitive.

```
IClient = {INew client [/*tags list*/]} % declare a client interface
IServer = {INew server [/*tags list*/]} % declare a server interface
Binding = {BNew IClient IServer} % bind the client interface to the server interface
```

Bindings may be chained to connect two interfaces through multiple reconfiguration points.

On the endings of a binding chain, the interfaces are connected to native Oz code. From an interface provider perspective, this boils down to define what to do with the messages in the output stream of an interface. The IImplements primitive allows one to define the procedure to invoke an every message. The primitive may be invoked multiple times on the same interface to replace the previous procedure with another one.

```
%% How to implement a server interface of a component
Itf = {INew server [/*tags list*/]} % declare the server interface
{IImplements Itf % bind the interface to a procedure
   proc {$ Message}
      /* process the Message here */
   end}
```

A convenient way to implement an interface is to associate it with an Oz object, because the object methods determine the different types of message expected on the interface:

```
Itf = {INew server [/*tags list*/]} % declare the server interface
{IImplements Itf % bind the interface to an object
   {New class $
      meth message1(Parameter1 ...)
         /* process Messages of type 'message1' */
      end
      ...
   end} init}
```

From an interface user perspective, this involves resolving the Interface into a proxy for the implementation procedure of the interface. The primitive IResolveSync (resp. IResolveAsync) performs the resolution of an interface into a synchronous (resp. asynchronous) proxy.

```
%% How to use a client interface inside a component
Itf = {INew client [/*tags list*/]} % declare the client interface
ItfProxy = {IResolveSync Itf} % resolve the interface into a proxy
{ItfProxy message1(Parameter1 ...)} % invoke the implementation procedure through the proxy
```

15.6.4 FructOz entities

FructOz represents the interfaces, components and bindings presented before as objects inheriting an abstract Entity base class, that provides generic navigation facilities through tags filtering: an Entity instance may be tagged with a set of keywords. Set of entities can then be filtered based on their tags. FructOz thus exports the following classes:

Interface : the representation of an interface is oriented, being either client or server; an interface has a static reference to the membrane it belongs to; the interface also references the bindings starting from and pointing to it.

Membrane : the representation of a component membrane, containing a dynamic set of interfaces.

Binding : the representation of a binding between a client interface and a server interface. A binding is unmutable and keeps two static references to the client and the server interfaces.

Here are the notations that we use in the following:

$\mathcal{C}$: a component (i.e. a Membrane)

$\mathcal{I}$: an interface

$\mathcal{B}$: a binding

$\mathcal{S}$: a set

$\mathcal{S}\{X\}$: set of elements of type X

$\mathbb{B}$: a boolean

Entities programming interface.

The three entities presented above may be created and manipulated by the following set of primitives:

CNew: $unit \rightarrow \mathcal{C}$, create a new empty component membrane;

INew: $(client|server) \rightarrow \mathcal{I}$, create a new interface, client or server, not associated to any component;

BNew: $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{B}$, create a binding between a client and a server interface;

BNewLazy: $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{B}$, create a lazy binding between a client and a server interface: the client interface is required to be instantiated to establish the binding, while the server interface will remain lazy (unneeded), as long as no introspection occurs involving the interface and no functional usage of the binding happens.

Components accept the following operations:

CAddInterface, CRemoveInterface: $\mathcal{C} \times \mathcal{I} \rightarrow unit$, add or remove an interface to or from a component.

Interfaces also accept the following:

IImplements: $\mathcal{I} \times$(native Oz object) $\rightarrow$ *unit*, define the procedure or object to invoke to process messages addressed to an interface.

IResolveSync, IResolveAsync: $\mathcal{I} \rightarrow$ (message $\rightarrow$ *unit*), resolve an interface into a synchronous or asynchronous proxy that may directly be invoked to send messages to the interface.

Bindings are immutable. Bindings can only be broken (or introspected), in which case they become garbage:

BBreak: $\mathcal{B} \rightarrow$ *unit*, break a binding.

As explained previously, components, interfaces and bindings are entities, and are thus associated with a set of tags. Tags allow the identification of any entity. Tags are manipulated with the following primitives:

Tag, Untag: entity $\times$ *tag* $\rightarrow$ *unit*, apply or remove a tag to or from the given entity.

HasTag: entity $\times$ *tag* $\rightarrow \mathbb{B}$, test an entity (a component membrane, an interface or a binding) for the given tag.

Immediate introspection primitives.

CGetInterfaces: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$, get the set of interfaces of the specified component.

IGetComponent: $\mathcal{I} \rightarrow \mathcal{C}$, get the component owning the given interface.

IIsClient, IIsServer: $\mathcal{I} \rightarrow \mathbb{B}$, test whether the given interface is a client (resp. server) interface with respect to its owning component.

IGetBindingsFrom, IGetBindingsTo: $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$, get the set of bindings connected from (resp. to) the given interface.

BGetClientInterface, BGetServerInterface: $\mathcal{B} \rightarrow \mathcal{I}$, get the interface the given binding connects from (resp. to).

15.6.5 Components as packaging and deployment entities

A functor is a function which creates (i.e. instantiates and exports) a new module (essentially, a record of Oz values), as a result of linking a module definition to its module dependencies (imports):
functor : {modules} $\rightarrow$ module.
The choice of the modules to use as imports, i.e. the resolution of imports, is done by a module manager.

A component is implemented as an Oz module exporting the component membrane. The functor which initializes the module thus corresponds to the deployment procedure of the component. Deploying the component means instantiating the module with a module manager. Note that a module (i.e. a component) may be instantiated multiple times by the same module manager. A module manager is thus a component factory.

```
functor ComponentPackage
export membrane: Membrane
define
    %% Create a new empty component membrane
    C = {CNew}
    ...
    %% Create the component content (interfaces, implementation, subcomponents, bindings, etc)
    ...
    %% Now that the component is deployed and ready to be
    %% used, make the membrane available
    Membrane = C
end
```

Note that the module imports in the functor declaration representing the component do not represent the client interfaces of the component (in the example `functor ComponentPackage` above, there is no `import` section).

The deployment of the component depicted above into a running entity can be achieved in the local running virtual machine with the following deployment primitive example:

```
fun {Deploy ComponentPackage}
    %% Create a new module manager
    ModuleManager = {New Module.manager init}
    %% Ask the module manager to apply the functor procedure,
    %% this instantiates the component and exports a reference to its membrane
    C = {ModuleManager apply(PackedComponent $)}
in
    %% Return the membrane exported by the instantiated module
    C.membrane
end
```

15.6.6 Distributed environments

The Mozart/Oz platform integrates a distributed programming environment. The platform aggregates several nodes into a unique global store in which objects (values) can be freely shared and accessed. The control of the distribution over the nodes is explicitly handled through the joint use of functors and module managers (i.e. functor executors). An Oz module manager is tied to the machine it has been created on, and thus always deploys modules locally on that machine.

In a centralized, non-distributed environment, components may be deployed on the same machine and in the same virtual machine, using the `{Deploy PackedComponent}` primitive presented in section 15.6.5. To represent a non-distributed environment consisting of a single machine, FructOz provides a Host component acting as a component factory: the Host component exports a `factory` interface, with a `deploy`

operation relying on the Deploy primitive. A Host component represents a managed node in the reference model of Section 15.5.

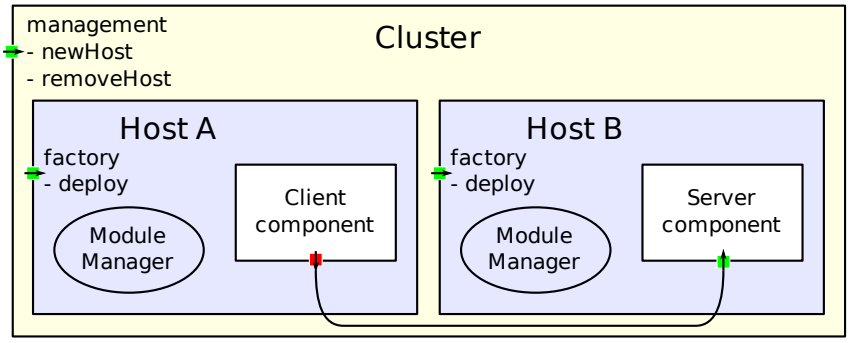


Figure 15.1: Distributed environments representation

As an example showing how to extend this to a distributed environment, a Cluster component has been implemented to represent a logical set of fully interconnected hosts. The Cluster component provides operations to add and remove machines from the set of nodes, hence acting as a Host factory. Introspection of the Cluster component thus allows to discover the distributed infrastructure. Distributed deployments may then be parameterized with a cluster parameter identifying the scope of the deployment. Deploying a component on this infrastructure now requires to make explicit the host a component should be deployed on. For this purpose, we provide a new deployment primitive: {RemoteDeploy Host PackedComponent}, relying on the factory interface of the Host component to dispatch the deployment on the remote hosts. The Deploy primitive remains available to deploy components locally.

```
fun {RemoteDeploy Host PackedComponent}
    %% Get and resolve the 'factory' interface of the Host component
    FactoryItf = {CResolveSyncInterface Host factory}
in
    {FactoryItf newComponent(PackedComponent $)}
end
```

As an implementation detail, the cluster component creates and integrates new hosts in the distributed environment as follows: it remotely starts an Oz virtual machine via an SSH shell command; the new remote Oz virtual machine opens a connection with the source virtual machine, creates a module manager and transmits it (more exactly, a proxy for it) back to the source machine, thus making it available to the cluster component. Finally the cluster component instantiates a host component remotely on the new virtual machine using the proxy for the remote module manager.

```
fun {NewRemoteHost Hostname}
    %% First start a new Oz virtual machine on the remote Host and obtain a proxy
    %% to a module manager in this virtual machine
    RemoteModuleManager = {New Remote.manager init(host:Hostname fork:ssh detach:false)}
```

```
    %% Remotely deploy a Host component on the new virtual machine
    HostModule = {RemoteModuleManager apply(PackedComponent $)}
in
    %% Returns the membrane of the new Host component
    HostModule.membrane
end
```

15.6.7 LactOz: a dynamic FPath library

FPath [39] is a query language for Fractal architectures. It is to Fractal architectures what XPath is to XML documents: a compact notation inspired by XPath for navigating through Fractal architectures and for matching elements according to some predicate. However, contrary to XML documents, Fractal architectures are dynamic and their structure and content may evolve over time. The evaluation of a standard FPath expression, as proposed in [39], produces a static result, only meaningful with respect to the original architecture it has been applied on, and which might not reflect the architecture as it may have evolved since the evaluation of the FPath expression. LactOz provides Dynamic FPath expressions that capture the dynamicity of architectures within FPath expressions. LactOz endows FPath expressions with the ability to be dynamically updated following to architecture evolutions.

Updates to dynamic FPath expressions may happen synchronously or asynchronously. Thus FPath variables and expressions that may be static, dynamic with synchronous updates or dynamic with asynchronous updates.

- Static variables are defined once and forever.
- Dynamic variables can be updated. Dynamic variables can be:
  - explicit: such variables are "manually" updated,
  - implicit: those variables are defined relatively to a set of (dynamic) source variables and are automatically updated when their sources change. A source propagate its changes to an implicit variable either synchronously or asynchronously.

Predicates

FPath predicates are built using Oz expressions on top of FructOz introspection primitives. We demonstrate here how to build the dynamic FPath expression contained in the dynamic FPath library:
CGetExternalComponentsBoundFrom: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$. Given a component C, we want to select all the components C' where there exists a binding from a client interface of C to a server interface of C'. The exploration process to achieve this computation starts from C, then gets C client interfaces, then C client bindings, then the server

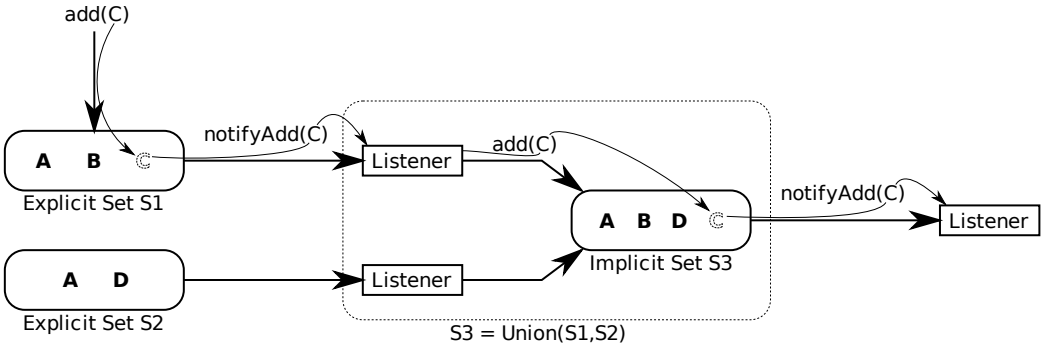


Figure 15.2: Anatomy of a simple union dynamic set: handling of an update.

interfaces of C client bindings, and finally the components pointed to by these bindings.

```
%% Auxilliary introspection functions
fun {CGetClientInterfaces C} % get the dynamic set of client interfaces of a component
   %% The kind of an interface cannot change, so we optimize
   %% using a dynamic set filtering with a static filter function
   {SStaticFilter AllItfs (fun {$ I} I.kind == client end)}
end

fun {BGetServerComponent B} % get the component owning the interface the binding points to
   {IGetComponent {BGetServerInterface B}}
end

fun {CGetExternalComponentsBoundFrom C}
   ClientItfs = {CGetClientInterfaces C}
   %% SMap maps the Set of (client) Interfaces into a Set of Set of Bindings,
   %% that we flatten thanks to SUnion into a Set of Bindings
   ClientBindings = {SUnion {SMap ClientItfs (fun {$ I} {IGetBindingsFrom I} end)}}
   %% and finally, map the Set of Bindings into a Set of Components
   {SMap ClientBindings (fun {$ B} {BGetServerComponent B} end)}
end
```

Now suppose you want to filter the set of components obtained with some predicate. For example, we want only those having a sub-component tagged with the tag "interesting", you might build and use the following predicate:

```
%% filter the dynamic set of components generated with the previous function
{SFilter {CGetExternalComponentsBoundFrom C}
   fun {$ C}
      %% get the sub-components of C
      Children = {CGetSubComponents C Context}
   in
      %% only keep components for which the sub-component set contains at least
      %% one child component tagged with 'interesting', i.e. for which the sub-component
      %% set filtered with respect to the 'interesting' tag is not empty
      {BNot {SIsEmpty {SFilterHasTag Children 'interesting'}}}
   end}
```

The predicate here is dynamic as its prototype is: $\mathcal{C} \rightarrow \mathbb{B}$ where the boolean might change if the component is added or removed some children or if some children are tagged or untagged with "interesting".

LactOz comes with a general set of primitives such as, for instance, BNot which creates a dynamic boolean computing a logical not of another dynamic boolean, SIsEmpty which tests the emptiness of a set, SMap, SUnion, SFilter etc. All these primitives can be tuned to be static or dynamic, synchronous or asynchronous. This requires some extra naming (such as Sync.sFilter or Async.sFilter) not shown here for brevity.

Basic entities primitives

BNot: $\mathbb{B} \rightarrow \mathbb{B}$

BAnd, BOr: $\mathbb{B} \times \cdots \times \mathbb{B} \rightarrow \mathbb{B}$ or $\mathcal{S}\{\mathbb{B}\} \rightarrow \mathbb{B}$

BWait: $\mathbb{B} \rightarrow unit$

NSum, NMultiply,

NMin, NMax, NAverage: $\mathbb{N} \times \cdots \times \mathbb{N} \rightarrow \mathbb{N}$ or $\mathcal{S}\{\mathbb{N}\} \rightarrow \mathbb{N}$

NSubtract: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

NDivide: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

NIDivide: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

SSize: $\mathcal{S} \rightarrow \mathbb{Z}$, get the number of elements of the set

SIsEmpty: $\mathcal{S} \rightarrow \mathbb{B}$, test the emptiness of the set

SUnion: $\mathcal{S} \times \cdots \times \mathcal{S} \rightarrow \mathcal{S}$ or $\mathcal{S}\{\mathcal{S}\} \rightarrow \mathcal{S}$

SFilter: $\mathcal{S} \times \mathcal{F} \rightarrow \mathcal{S}$, where $\mathcal{F} : \mathcal{V} \rightarrow \mathbb{B}$

SMap: $\mathcal{S} \times \mathcal{F} \rightarrow \mathcal{S}$, where $\mathcal{F} : \mathcal{V} \rightarrow \mathcal{V}$ is assumed deterministic

SSubtract: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$

SIntersect: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$

This set can be easily extended.

Basic component introspection primitives

Based on these primitives, we build higher level navigation operations. Note: the terms “external” and “internal” refer to the implicit inside and outside of a component.

IIsBoundExternally, IIsBoundInternally: $\mathcal{I} \rightarrow \mathbb{B}$
: test whether the interface is externally (resp. internally) bound relatively to the implicit inside of its owning component

CGetExternalBindingsFrom, CGetExternalBindingsTo,

CGetExternalBindings: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
get the external bindings to and/or from the component

CGetInternalBindingsFrom, CGetInternalBindingsTo,

CGetInternalBindings: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
get the internal bindings to and/or from the component

BGetClientComponent, BGetServerComponent: $\mathcal{B} \rightarrow \mathcal{C}$
get the component client (resp. server) of the binding

CGetExternalComponentsBoundTo, CGetExternalComponentsBoundFrom,

CGetExternalComponentsBoundWith: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
get the external components bound to and/or from the component

CGetInternalComponentsBoundTo, CGetInternalComponentsBoundFrom,

CGetInternalComponentsBoundWith: $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
get the internal components bound to and/or from the component

15.7 Case studies

In this section, we present examples of dynamic architectures illustrating how our framework supports the requirements we identified in the introduction.

15.7.1 Parameterized architectures

This example shows how to describe highly parameterized architectures. For this purpose we present an architecture composed of two sets of components and where a parameter defines the interconnection scheme between these two sets (see Figure 15.3).

The parameters for this architecture are:

- the size of the first set of components
- the size of the second set of components
- an interconnection scheme function: $f : \mathcal{S}\{\mathcal{C}\} \times \mathcal{S}\{\mathcal{C}\} \rightarrow \mathcal{S}\{\text{Desc}(\mathcal{B})\}$ where $\text{Desc}(\mathcal{B})$ represents a descriptor for a binding (in our scenario, a descriptor is a pair of interfaces IFrom#ITo).

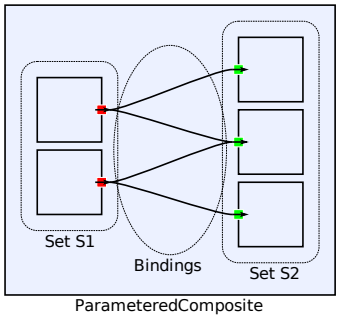


Figure 15.3: Parameterized architectures: interconnection scheme

```
fun {ParameterizedComposite N1 N2 BindingScheme}
   functor $
   export Membrane
   define
      C = {CNew nil}
      %% Create the first set S1 with N1 instances of component C1.
      S1 = {SNew} % create a new empty set S1
      for I in 1..N1 do
         SubComp = {Deploy C1} % deploy a component C1 locally
      in
         {S1 add(SubComp)} % add the new component into the set S1
      end
      %% Create S2 similarly to S1
      ...
      %% Invoke the interconnection scheme function: generate a list of binding descriptors.
      LDescs = {BindingScheme S1 S2}
      %% Translate these descriptors into real bindings.
      LBindings = {List.map LDescs
         fun {$ IFrom#ITo}
            {BNew IFrom ITo}
         end}
      Membrane = C
   end
end
```

A trivial example of interconnection scheme is the full interconnection scheme which realizes the complete two-party graphs.

```
fun {FullInterconnect S1 S2}
   L = {NewCell nil} % create a new empty descriptor list
in
   %% for all pairs (CFrom, CTo) in (S1 x S2)
   {S1 forAll(
      proc {$ CFrom}
         {S2 forAll(
            proc {$ CTo}
               IFrom = {CGetInterface CFrom [/*tags list*/]} % locate the client interface
               ITo = {CGetInterface CTo [/*tags list*/]} % locate the server interface
               Desc = IFrom#ITo % make the descriptor
            in
               L := Desc | @L % prepend the new descriptor to the list
            end)}
      end)}
```

```
    @L % return the list of descriptor
end
```

15.7.2 Synchronization and workflows

In the example depicted in section 15.7.1, the sub-components are deployed sequentially, one after the other. We describe here how to describe and integrate distributed synchronizations. For this purpose, we define a Barrier synchronization pattern in the Barrier procedure, analogous to a parallel AND workflow pattern. The procedure is used to parallelize and synchronize on the completion of the deployment of N_2 instances of component C_2. Expressing the synchronization pattern as a procedure allows to consider the synchronization pattern as a parameter of an architecture (such as the BindingScheme parameter of the example shown in section 15.7.1).

```
%% Barrier synchronization pattern
proc {Barrier P N}
    Bar = {Tuple.make barrier N}
    for I in 1..N do
        thread
            {P} % invoke the procedure
            Bar.I = true % ready signal once the procedure is over
        end
    end
    {Record.forAll Bar Wait} % wait for all N ready signals
end

%% Apply the barrier pattern to deploy N2 instances of C2 components
{Barrier
    proc {$}
        SubComp = {Deploy C2}
    in
        {S2 add(SubComp)}
    end
    N2}
```

15.7.3 Lazy deployments

In this example we show how to implement lazily deployed components. We consider 3 levels of deployment: (i) level 0: the component is represented by a lazy variable and is not deployed at all; (ii) level 1: the component membrane is deployed, which allows to introspect its interfaces and immediate sub-components; the implementation of the component is not deployed yet. (iii) level 2: the component is fully deployed.

Contrary to the transition from level 0 to level 1, which is atomic, the transition from level 1 to level 2 may contains several intermediary states with partial deployments of the component. For instance, a composite component with many sub-components may be somewhere between layer 1 and layer 2 with only a few of its sub-components deployed.

This relies on lazy bindings instantiated via the BNewLazy primitive. Lazy bindings require that the client interface is determined, but allow the server interface to be lazily determined. The BNewLazy primitive differs from BNew in that the server interface will only be made needed, thus deployed, on usage of the binding (for either functional or control purpose such as introspection).

```
%% Lazily deployed implementation
Itf = {INew server [tags ...]}
{IImplements Itf
   {ByNeed fun {$} % Implementation will be lazily instantiated
      {New class $ ... end}}}

%% Lazily deployed component
Client = {ByNeed fun {$} {Deploy LazyClient} end}

%% Lazily deployed binding between a Client component and a Server component
B = thread
      %% Wait until someone needs and thus triggers the deployment of the Client component
      %% Once this has happened, get the client interface
      IFrom = {CGetInterface {WaitQuietValue Client} [client]}
      %% Create a lazy reference to the server interface
      ITo = {ByNeed fun {$} {CGetInterface Server [service]} end}
   in
      %% Create a lazy binding between Client (now determined) and Server (might still be lazy)
      {BNewLazy IFrom ITo}
   end
```

15.7.4 Error handling

In the following, we sketch how to take deployment failures into account. First we define a (trivial) error handling pattern, which we use afterwards in a concrete scenario. The scenario consists in a simple compensation behavior to the instantiation of a logging component, which by default tries to deploy a remote logger, and fall back on a file logger if the remote logger cannot be instantiated. The handling pattern may thus be used as a parameter of an architecture, much as the synchronization pattern in a previous example.

```
%% Basic try/catch error handling pattern
proc {HandleError P E}
   try {P} % execute procedure P
   catch AnyException then
      {E} % if any error happens during P, then execute E
   end
end

%% Example of deployment with error handling
{HandleError
   proc {$}
      % try to deploy a component RemoteLogger
      Log = {Deploy RemoteLogger}
   end
   proc {$}
      % if RemoteLogger cannot be instantiated, fall back on component FileLogger
      Log = {Deploy FileLogger}
   end}
```

```
%% Within the deployment procedure of RemoteLogger
if (/*remote resource is not available*/) then
    raise instantiationFailure(cause: unavailableResource) end
end
```

15.7.5 Self-configurable architecture

We present now how to extend the example of parameterized architecture presented in section 15.7.1 with dynamic parameters. The original example is a parameterized architecture {ParameterizedComposite N1 N2 BindingScheme} where N1 and N2 are two constant integers and where BindingScheme is a function which generates a set of binding descriptors given two sets of components.

The architecture dynamicity with respect to N_1 means: (i) to dynamically deploy or undeploy instances of C_1, and (ii) to dynamically add and remove bindings between C_1 instances and C_2 instances. For this purpose, the interconnexion scheme takes the dynamicity into account in generating a dynamic set of binding descriptors. The composite component listens to this dynamic set of descriptors and translates new (resp. removed) set entries into binding creations (resp. removal).

The dynamic set of descriptors is implemented as an object integrated into an event-based system. The object is parametered with two dynamic sets S1 and S2 and is thus listening and reacting to these sets' updates. For example, in response to an update event related to S1 such as a removal of elements, method removeS1(SElements) is invoked, which adjusts the state of the descriptor set. Adjustements to the descriptor set made in removeS1 generate new events in cascade propagated to listeners of this set, such as a dynamic architecture.

```
%% How to construct a dynamic set
class DynamicBindingDescSet from ImplicitSet
    meth init
        %% Listen to S1 updates
        self.s1_listener = {New SetListenerForwarder init(self S1 sync addS1 removeS1)}
        {S1 listen(self.s1_listener)}
        ...
    end
    ...
    meth removeS1(SElements) % On removal of elements from S1:
        lock % mutual exclusion to prevent race conditions
            %% Update our local view of S1
            s1 := {FSet.removeAll @s1 SElements}
            %% Remove all descriptors referencing components that are removed from S1
            {FSet.forAll SElements
                proc {$ CFrom}
                    Set,filterInPlace(
                        fun {$ Desc}
                            ({IGetComponent Desc.iFrom} == CFrom)
                        end)
                end}
        end
    end
    ...
```

```
end

%% How to listen and react to events generated by the dynamic descriptor set
class MyDescSetListener from SetListener
   ...
   meth remove(SElements) % On removal of descriptors:
      {FSet.forAll SElements
         proc {$ Desc} % destroy the bindings corresponding to the removed descriptors
            {BBreak Desc.binding}
         end}
   end
end
%% Create and activate our listener on DescSet
{DescSet listen({New MyDescSetListener init(C DescSet sync)})}
```

15.7.6 Deployment scenarios

We present here how to describe different deployment strategies. We consider the deployment of a set of identical components on a cluster of nodes and organized as subcomponents of the composite component described as follows. The size of the set of components drives the complexity of the overall deployment process.

```
fun {CompositePackage Cluster N DeployProc}
      functor $
      export Membrane
      define
            Comp = {CNew}
            {DeployProc SubcomponentPackage Cluster N Comp}
            Membrane = Comp
      end
end
```

The composite component description is parameterized by a deployment procedure which is responsible of the deployment of the N identical subcomponent of the composite.

Sequential deployment. Our first deployment strategy consists in deploying all subcomponents sequentially from a single centralized controller as follows:

```
proc {DeploySeq CompPackage Cluster N Parent}
   NextRoundRobin = {MakeRoundRobin Cluster}
   for I = 1..N do
      Host = {NextRoundRobin}
      NewComp = {RemoteDeploy Host CompPackage}
      {CAddSubComponent Parent NewComp}
   end
end
```

Centralized asynchronous parallel deployment. Adding uncontrolled parallelism to the previous strategy is trivial thanks to Oz:

```
proc {DeployPar CompPackage Cluster N Parent}
   NextRoundRobin = {MakeRoundRobin Cluster}
   for I = 1..N do
      thread
```

```
            Host = {NextRoundRobin}
            NewComp = {RemoteDeploy Host CompPackage}
            {CAddSubComponent Parent NewComp}
        end
    end
end
```

Centralized synchronous parallel deployment Relying on the barrier synchronization described in section 15.7.2, we build a centralized deployment strategy that spawns and synchronizes on concurrent deployments as follows:

```
proc {DeployParallel CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    %% Deployment script
    proc {DeployProc}
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host CompPackage}
        {CAddSubComponent Parent NewComp}
    end
in
    %% Execute and synchronize on the scripts' execution
    {Barrier DeployProc N}
end
```

Tree distributed deployment All deployment strategies presented up to now are executed on a single centralized controller node. Here is how to build a distributed deployment process based on a tree distribution strategy. The deployment process is initiated on the root node of the tree. Each node of the tree locally deploys one instance of the component; each node also initiates the deployment process on all their branches and synchronizes on them.

```
NextRoundRobin = {MakeRoundRobin Cluster}
proc {DeployTree CompPackage Arity Depth Parent}
    functor DistributedDeployProc
    export Membrane
    define
        if (Depth > 0) then
            {DeployTree CompPackage Arity (Depth - 1) Parent}
        end
        Membrane = {Deploy CompPackage} % deploy component locally
    end
    proc {DeployProc}
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host DistributedDeployProc}
        {CAddSubComponent Parent NewComp}
    end
in
    {Barrier {MakeCopyList DeployProc Arity}}
end
```

	Component deployment	Remote invocation
SmartFrog/Java RMI	295.5 ms	0.097 ms
Julia/Fractal RMI	159.5 ms	0.099 ms
FructOz	146.3 ms	0.0048 ms
FructOz/Julia Bridge	262.3 ms	N/A

Table 15.1: Deployment and remote invocation costs comparison

15.8 Evaluation

15.8.1 Microbenchmarks

We show here a performance evaluation to compare the deployment process and the cost of remote method invocations on SmartFrog/Java RMI, Julia/Fractal RMI and FructOz/Mozart. Additionally we evaluate the deployment process of a "FructOz/Julia bridge" (see the details at the end of this section). For this purpose, we measure the latency of the deployment of the distributed composite component depicted in Figure 15.4 and we also evaluate the cost of a synthetic remote method invocation between the client and the server subcomponents.

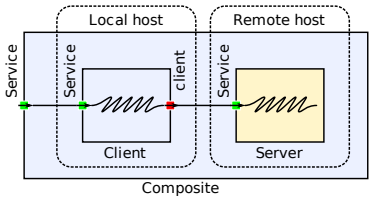


Figure 15.4: Simple distributed component

The experiments have taken place on the Grid'5000 infrastructure, and the measures are realized on dual-opteron 252 (2.6GHz) machines with 4Gb of memory and all interconnected with 1Gb network interfaces. We used SmartFrog 3.12.000 and Fractal 2.0.1, running on the Sun Java virtual machine v1.6.03, and the Mozart/Oz virtual machine v1.3.2. To evaluate the remote method invocation cost, we evaluate the time required to complete 10000 synthetic method calls. The garbage collector is manually triggered between measures sequences so as to minimize its impact on the performance. The experiments have been repeated 30 times to finally report averaged measures. Table 15.1 summarizes the results of this evaluation.

The deployment of the distributed component is more efficient on FructOz, which may be explained because FructOz does not need any descriptor parsing as this is the case for SmartFrog and Fractal ADL. In fact, SmartFrog reported during the experiments an average descriptor parsing time of 150 ms, which is about half the time of our SmartFrog deployment process. Remote method in-

vocations is more efficient on the Mozart/Oz platform. This may be explained because the Mozart/Oz marshaler is directly implemented and optimized in C++ while Java and Fractal RMI marshalers are implemented in Java and use the Java reflection. Additionally, Mozart/Oz heavily relies on lazy marshalling, and serializes complex structures when this is required by a remote process only. Overall, these microbenchmarks show that the basic operations in our platform compare favorably with other environments.

FructOz/Julia bridge: We extended the FructOz framework with an Oz/-Java bridge that endows FructOz with the ability to drive Java tasks. Moreover, the bridge exports specific hooks to manipulate the Fractal/Julia component model in the Oz world. This way, FructOz constitutes a deployment engine for Fractal/Julia components. The Oz/Java bridge is built as a set of XML-RPC handlers, and we used the Apache XML-RPC v3.1 Java implementation and the Oz XML-RPC client modules for the experimentation.

The deployment of the distributed component in this configuration thus adds the cost of XML-RPC invocations and Fractal/Julia primitive executions to the latency measured on the regular FructOz environment, which explains the higher deployment cost (262.3 ms). Finally, remote method invocations could happen either as Oz invocations or as Fractal RMI invocations, thus mixing the performance of both configurations.

This demonstrates the applicability of the FructOz framework on heterogeneous non-Oz environments. More concretely, driving the deployment of legacy applications such as J2EE servers would require the design of component wrappers for these legacy applications (see e.g. [6] for details).

15.8.2 Local deployments

In the following we evaluate the scalability of local single-machine deployments on the Mozart/Oz platform. As an evaluation base, we deploy a composite component containing a single client subcomponent bound to a number of server subcomponents. The number of server subcomponents thus constitutes the size of the deployed component. All measures are repeated 3 times and averaged.

Results are presented in Figure 15.5 and show that the deployment cost increases with the number of components deployed. This behavior may to be correlated with the garbage collector of the Oz virtual machine. Indeed we reported an issue preventing the correct collection of generated values, leading to an increase in the heap size, thus slowing down the collection mechanism. Moreover, the FructOz framework has not been designed with performance in mind, and circumvents a number of limitations of the current implementation of the Oz virtual machine with some design and performance overhead. Thus there is room for significant optimizations.

Figure 15.5: Local deployment evaluation

15.8.3 Distributed deployments

The following experiments consist in deploying increasing numbers of components on a distributed environment made of a cluster of 16 machines of the Grid'5000 infrastructure (described more in details in section 15.8.1). The machine on which a new component is deployed is determined with a simple round-robin policy over the machines available in the cluster. In Figures 15.6(a) and 15.6(b) we demonstrate the gains that can be obtained by defining the appropriate deployment workflow. Figure 15.6(a) compares two simple workflows, executing on a single machine: the first one is just the sequential deployment of a number of components on different machines; the second one is the parallel deployment of the same number of components on the same number of machines. Just increasing the parallelism provides an interesting improvement. A more drastic speed-up is obtained when changing the workflow to a distributed one. In Figure 15.6(b), we show the result of distributing the deployment workflow on a tree of machines. We experimented with different forms of trees, as reported in the Figure. One can notice a huge improvement (5× speedup) over a centralized sequential workflow. Interestingly, as can be seen in the code for our centralized and distributed deployment scenarios in Section 15.7.6, for the same target component configuration, changing the workflow process with our framework, is only a matter of changing a parameter in a higher-order procedure.

Note that in Figure 15.6(b) we have not been able to obtain results beyond 600 components, because of instabilities of the current Oz virtual machine. These are due, we believe to some interplay between the garbage collector and distribution, but we have not been able at this point to ascertain the exact source of the problem.

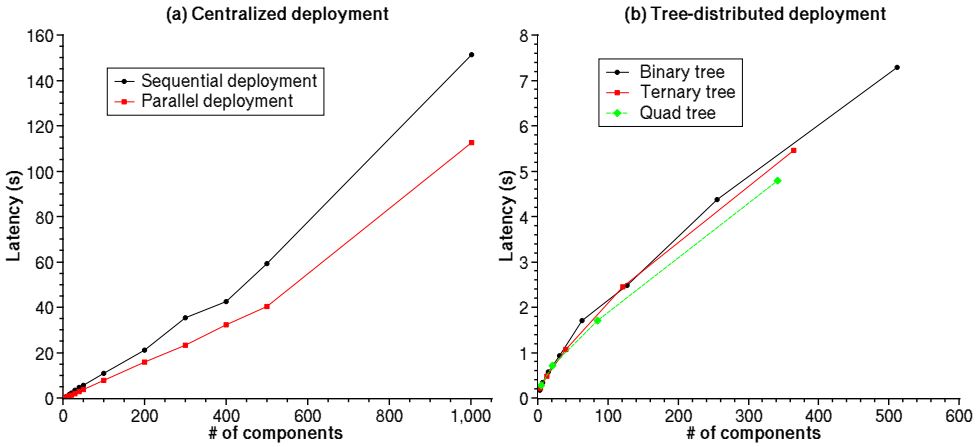


Figure 15.6: Distributed deployments evaluation

15.9 Discussion and future work

The work reported in this deliverable constitutes a first step towards meeting our two main objectives:

1. To provide basic support for complex deployment and configuration processes.
2. To provide basic support for the construction of distributed and self-configurable components.

Compared to the state of the art in software deployment in general and architecture-based deployment in particular, we have in place a framework that allows highly parameterized descriptions of complex distributed deployment and configuration processes, and that allows the construction of distributed components embedding their own monitoring and control loop capabilities. However, our work still faces a number of limitations, that call for additional study:

- The FructOz framework was developed using the 1.3.2 version of the Mozart environment. Hence, we could not benefit from the failure handling facilities introduced in the 1.4.0 version, and documented in R. Collet's PhD thesis (see appendix to this book, and Year 1 Selfman deliverable D2.3a). We plan to exploit these facilities in the next version of our FructOz framework, to support more comprehensive and systematic patterns for handling distributed failures.
- Handling distributed failures in deployment and configuration processes calls for transactional support. Work introducing transactional support for reconfiguration processes (exploiting the Fpath/Fscript technology for the definition of reconfiguration programs) is reported in Deliverable D4.2a (Chapter 16 of this book). We plan to exploit this work and to extend it to fit with

the FructOz Dynamic FPath and distributed deployment and configuration capabilities.

- The FructOz framework has been experimented successfully with cluster size environments (although we experienced some scalability issues which we revisit with the new Mozart 1.4.0 distributed infrastructure). However, it is not clear that we can readily make use of FructOz in the larger scale and more dynamic peer-to-peer environments targeted by Selfman. We identify three areas of work to consider in Year 3 of the Selfman project: extending our dynamic FPath monitoring capabilities to a P2P context, adding aggregation capabilities and coupling it with dynamic slicing capabilities [48]; studying the question of high-level programming abstractions for reconfiguration effectors in large P2P systems (for instance, to efficiently support large-scale push-style deployments); exploiting the work on DHT-based transactions reported in Deliverable D3.1b (Chapter 12 in this book) to support transactional deployment and configuration processes in a P2P environment.

15.10 Supplement: Workflow patterns in Oz

We present in this section an interpretation of a collection of workflow control-flow patterns in Oz. For simplicity, we present only an interpretation of the first twenty control-flow patterns taken from [142], which have then been refined and extended in [124]. Other patterns in [124] can be similarly captured in Oz, but we leave that as an item for future work. In the description of workflow patterns below, we keep the names and descriptions of patterns given by [124].

Before presenting our formalization of workflow patterns in Oz, a few words about task modelling. The interpretation presented in this appendix is related to the π-calculus formalization of the same control-flow workflow patterns presented in [111]. The π-calculus can be seen as a direct subcalculus of the Oz kernel language, so it is worthwhile to review the formalization proposed in [111], and to contrast it with that of this appendix. In [111], a basic task, i.e. one which is not built using workflow pattern operators, is modelled as a simple sequential π-calculus process, of the form

$$x_1 \ldots x_n.[a_1 = b_1] \ldots [a_p = b_p].\tau.\overline{y}_1 \ldots \overline{y}_m.\mathbf{0}$$

where x_i are input actions that trigger the basic task, the name equality checks $[a_i = b_i]$ correspond to the checking of some optional conditions (e.g. checking a cancellation flag), τ is the π-calculus silent action, which models the execution of the functional part of the basic task, and $\overline{y}_i$ are an output actions, that can trigger some other process, and also denote the termination of the basic task. Basic tasks that can be triggered more than once are modelled using the π-calculus replication operator, thus:

$$!x_1 \ldots x_n.[a_1 = b_1] \ldots [a_p = b_p].\tau.\overline{y}_1 \ldots \overline{y}_m.\mathbf{0}$$

The workflow patterns that are described then apply only to basic tasks, or to slight variants of basic tasks as modelled above. This modelling is, in our view, overly simplistic:

- It does not consider data dependencies between tasks, and in particular it does not allow the functional part of a basic task to depend on input parameters.
- It does not allow the functional part of a basic task to interact with its environment.
- It only allows task cancellation before the actual execution of its functional part.
- It does not allow a compositional definition of workflows.

The last point is crucial: because control flow operators operate only on basic tasks, one cannot build higher-order workflow operators, i.e. workflow operators acting on workfow processes. In our modelling, we lift all the above limitations, and provide a set of composable programming abstractions for building workflow processes in Oz.

Specifically, in our approach, each workflow pattern is captured as a particular operator (an Oz higher-order procedure) that acts on task procedures. Task procedures are Oz procedures, whose invocation corresponds to the launch of the modelled task. A task procedure can be arbitrary (e.g. it can launch multiple concurrent thread as part of its execution), except for certain constraints (such as having certain arguments and behaving in a certain way), which are necessary for the proper functioning of the operator. Task procedures, provided they meet the constraints required for their composition using a given workflow operator, can encapsulate arbitrary workflow processes, including ones which have been built using other workflow operators.

When describing each workflow operator, we clarify the constraints that the procedures which are passed as arguments to the operator must meet. We also give a brief informal description for each pattern, directly taken from [124], to clarify the intent of the pattern. Where it is necessary, we add clarifications to the pattern intended semantics in the form of so-called context conditions, also taken from [124], i.e. conditions on how the pattern is to used, and how it is expected to behave in its target environment.

15.10.1 Basic control flow patterns

Sequence

> Pattern description: An activity in a workflow process is enabled after the completion of a preceding activity in the same process.

The sequence pattern can be modelled by the Seq operator below. Seq acts on a pair of tasks. Each task is modelled by a unary procedure whose unique argument is a logic variable that will be bound by to some value by the procedure to signal its termination. Note that the procedure execution can give rise to a concurrent process: the only requirement is that the execution termination be signalled by binding the parameter variable.

```
proc{Seq P Q}
   ZP ZQ in
   {P ZP} {Wait ZP} {Q ZQ} {Wait ZQ}
end
```

This operator can be generalized to act on a list of tasks and to allow for the result of a task to be fed to the following task in the list. Each task in the task list is modelled by a unary procedure whose argument correspond to a pair: (input argument, termination variable). Note that procedure SeqL returns only after the last task in the list has terminated.

```
proc{SeqL L I}
   case L of
      H | T then Z in {H I#Z} {Wait Z} {SeqL T Z}
   [] nil then skip
   end
end
```

Parallel split

> Pattern description: The divergence of a branch into two or more parallel branches each of which execute concurrently.

The parallel split pattern can be modelled by the ParSplit operator below. ParSplit acts on a pair of tasks, each sipmply modelled as a nullary procedure.

```
proc{ParSplit P Q}
  thread {P} end
  thread {Q} end
end
```

This operator can be generalized to act on a list of tasks. Each task in the task list is modelled as a unary procedure.

```
proc{ParSplitL L}
   case L of
      H | T then thread {H} end {ParSplitL T}
   [] nil then skip
   end
end
```

Synchronization

> Pattern description: The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

The synchronization pattern can be modelled by the ParSync operator below. ParSync implements a synchronization barrier that is passed only when the two argument tasks P and Q have both terminated. When both tasks have terminated, the third task R is triggered. Tasks P Q are modelled by unary procedures, which bind their unique parameter variable to some value when they have terminated. Task R is modelled by a unary procedure that takes as argument a list, corresponding to results from the two synchronized tasks.

```
proc{ParSync P Q R}
   ZP ZQ in
   thread {P ZP} end
   thread {Q ZQ} end
   {Wait ZP} {Wait ZQ} {R [ZP ZQ]}
end
```

The ParSync operator can be generalized to act on a list of tasks. Each task in the list is modelled by a unary procedure, as above. As above, the results from the synchronized tasks are gathered in a list that is passed as argument to the triggered task R.

```
proc{ParSyncL L R}
   fun{ParSyncLF L Z Lr}
      case L of
         H | T then
            ZH ZT in
            thread {H ZH} {Wait ZH} ZT = Z end
            {ParSyncLF T ZT {List.append Lr [ZH]}}
      [] nil then Z#Lr
      end
   end
   Z#Lr = {ParSyncLF L unit nil}
in
   {Wait Z} {R Lr}
end
```

Exclusive choice

> Pattern description: The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch.

The exclusive choice pattern between two alternative tasks, can be modelled by the ExChoice operator below. ExChoice takes as argument a nullary function BF which evaluates to a boolean, and which corresponds to the logical expression associated with the pattern. It takes also two nullary procedures which corresponds to the two alternative tasks in the pattern.

```
proc{ExChoice BF P Q}
   if {BF} == true then {P} else {Q} end
end
```

The ExChoice can be generalized to act on a list of tasks, and a corresponding list of boolean conditions. The ExChoiceL operator takes as argument a list of pairs

of the form BF#P, where BF is a nullary function that evaluates to a boolean, and P is a nullary procedure, corresponding to the task triggered if the condition BF evaluates to true.

```
proc{ExChoiceL L}
   case L of
      BF#P | T then if {BF} == true then {P} else {ExChoiceL T} end
   [] nil then skip
   end
end
```

Simple merge

> Pattern description: The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

The simple merge pattern between two alternative tasks can be modelled by the SimpleMerge operator. SimpleMerge takes three unary procedures, P, Q, R as arguments. Procedures P and Q model tasks whose termination is indicated by binding their unique argument to some value. Procedure R corresponds to the task that is triggered as soon as one of P or Q terminates. The value recorded in the termination variable of P or Q is passed as an argument to the task R.

```
proc{SimpleMerge P Q R}
   ZP ZQ in
   thread {P ZP} end
   thread {Q ZQ} end
   if {Record.waitOr ZP#ZQ} == 1 then {R 1#ZP} else {R 2#ZQ} end
end
```

The code of the SimpleMerge operator uses the function waitOr from the Record module of the base Mozart environment [144]. The statement {Record.waitOr ZP#ZQ} blocks until at least one field of the pair ZP#ZQ is determined (i.e. until at least one of ZP or ZQ is bound to some value), and it returns the feature (here '1' or '2') of a determined field.

The SimpleMerge operator can be generalized to act on a list of tasks.

```
proc{SimpleMergeL L R}
   fun{Smlf L Zs}
      case L of
     H | T then ZH in thread {H ZH} end {Smlf T {List.append Zs [ZH]}}
      [] nil then Zs
      end
   end
   Zs = {List.toTuple r {Smlf L nil}}
   Z = {Record.waitOr Zs}
in
   {Wait Z} {R Z#Zs.Z}
end
```

The code of the SimpleMergeL operator makes use of the Append function of the List module of the Mozart system, and of the toTuple function of the same module (that

transforms a list into a tuple – note that in Oz a tuple is just a record with consecutive integer features).

An alternative to the definitions above, which only relies on the asynchrony of thread execution in Oz, is given below. In this case, with operator SimpleMergeBis, we expect tasks P and Q to be unary procedures accepting a pair Id#G as argument, where Id is an integer (the index of the task in the list of tasks to merge), and G is an Oz port on which the termination values of the tasks to merge are sent. Each merged task is expected to terminate with a termination value of the form Id#M, where Id is the index of the task (which was passed as argument to the task procedure), and M is some value.

```
proc{SimpleMerge2 P Q R}
   S G = {Port.new S}
in
   thread {P 1#G} end
   thread {Q 2#G} end
   case S of Id#M | _ then {R Id#M} else skip end
end

proc{SimpleMergeL2 L R}
   S G = {Port.new S}
   proc{Smlfb L I}
      case L of
         H | T then thread {H I#G} end {Smlfb T I+1}
      [] nil then skip
      end
   end
in
   {Smlfb L 0}
   case S of Id#M | _ then {R Id#M} else skip end
end
```

A note on the above modelling of the simple merge pattern. In [111], it is strangely stated that the tasks to merge "will never be executed in parallel". However, this is clearly wrong since in [124] the Petri net-like description of the simple merge pattern behaviour, with two tasks, allows for the two tasks to run in parallel. We follow [124] as the reference for pattern descriptions.

15.10.2 Advanced branching and synchronization patterns

Multi-Choice

> Pattern description: The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is passed to one or more of the outgoing branches based on the outcome of distinct logical expressions associated with each of the branches.

The multi-choice pattern can be modelled by the MultiChoice operator below. MultiChoice takes a list of pairs of the form BF#P, where BF is a nullary boolean function (modelling a triggering condition), and P is a nullary task procedure. Note

that, in contrast to ExChoiceL which only triggers one task, provided its triggering condition is true, MultiChoice triggers all tasks in the list whose triggering condition evaluates to true.

```
proc{MultiChoice L}
   case L of
      BF#P | T then if {BF} == true
                    then thread {P} end {MultiChoice T}
                    else {MultiChoice T}
                    end
   [] nil then skip
   end
end
```

Structured synchronizing merge

> Pattern description: The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

The modelling of this pattern in Oz is different from the previous ones. The description of the pattern in [124] highlights a number of so-called context conditions for the pattern:

1. There must be a single Multi-Choice construct earlier in the process model with which the Synchronizing Merge is associated, and it must merge all of the branches emanating from the Multi-Choice.

2. The Multi-Choice construct must not be re-enabled befor the asociated Synchronizing Merge construct has fired.

3. Once the Multi-Choice has been enabled, none of the activities in the branches leading to the Synchronizing Merge can be cancelled before the merge has been triggered. The only exception is that it is possible for all of the activities leading up to the Synchonizing Merge to be cancelled.

4. The synchronizing Merge must be able to resolve the decision as to when it should fire based on local information available to it during the course of execution. Critical to this decision is knowledge of how many branches emanating from the Multi-Choice are active and require synchronization.

Because of these context conditions, we model this pattern with a higher-order procedure that encompasses both the initial Multi-Choice construct and the Synchronizing Merge that follows. The SyncMerge operator is defined as follows:

```
proc{SyncMerge L R}
   S G = {Port.new S}
   fun{MCh L LT I}
      case L of
         BF#P | T then if {BF} == true then thread {P I#G} end {MCh T {List.append LT [I]} I+1}
```

```
                else {MCh T LT I+1}
                end
        [] nil then LT
        end
    end
    fun{SMg S LT LR}
        case S of Id#M | T then
            LTn = {List.subtract LT Id} in
                if LTn == nil then LR else {SMg T LTn {List.append LR [Id#M]}} end
        end
    end
    LT = {MCh L nil 0}
in
    if LT == nil then skip else LR = {SMg S LT nil} in {R LR} end
end
```

In the code of the SyncMerge operator above, the function MCh corresponds to the Multi-Choice construct associated with the Synchronizing Merge. The synchronizing merge proper is realized by the function SMg which receives on port G and its stream S termination values from all the tasks which have been launched by the Multi-Choice construct. It is only after all these tasks have terminated that the task R is launched, with argument the list of all received termination values, with their index.

Multi-Merge

> Pattern description: The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

The Multi-Merge pattern can be modelled by the MultiMerge operator below.

```
proc{MultiMerge L R}
    S G = {Port.new S}
    proc{Launch L}
        case L of
            H | T then thread {H G} end {Launch T}
        [] nil then skip
        end
    end
    proc{Handle S}
      case S of M | T then thread {R M} end {Handle T} end
    end
in
    {Launch L} {Handle S}
end
```

Structured discriminator

> Pattern description: The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent

enablements of incoming branches do not result in the thread of control being passed on. The discriminator construct resets when all incoming branches have been enabled.

The discriminator pattern comes with a number of context conditions that complete its semantics:

1. The Discriminator is associated with precisely one Parallel Split earlier in the process and each of the outputs from the Parallel Split is an input to the Discriminator.

2. The branches from the Parallel Split to the Discriminator are structured in form and any splits and merge in the branches are balanced.

3. Each of the incoming branches to the Discriminator must only be triggered once prior to it being reset.

4. The Discriminator resets (and can be re-enabled) once all of its incoming branches have been enabled precisely once.

5. Once the Parallel Split has been enabled none of the activities in the branches leading to the Discriminator can be cancelled before the join has been triggered. The only exception to this is that it is possible for all of the activities leading up to the Discriminator to be cancelled.

The structured discriminator pattern can be modelled by the StructDiscrim operator below. Each task in the list L passed as argument to StructDiscrim, is supposed to repeatedly produce results, which are sent on a result port G which is passed as argument to the unary procedure that models the task when it is instantiated (as part of the Launch procedure execution). It embeds: a parallel split in its first phase, manifested by the Launch procedure; a synchronization condition, manifested by the Barrier procedure, that waits for all the launched task to have yielded some result to trigger a new phase of the discriminator; the discriminator itself, manifested by the Discrim procedure, that recursively waits for the result from one of the launched tasks before triggering procedure R with the obtained result as argument.

```
proc{StructDiscrim L R}
   fun{Launch L Lg}
      case L of
         H | T then S G = {Port.new S} in thread {H G} end {Launch T {List.append Lg [G#S]} }
      [] nil then Lg
      end
   end

   fun{Result Lg}
      Rg = {List.toRecord r Lg}
      G = {Record.waitOr Rg}
   in
      case Rg.G of M | _ then G#M end
   end
```

```
      fun{Barrier Lg Ld}
         case Lg of
            G#S | Tg then case S of _ | T then {Barrier Tg {List.append Ld [G#T]} } end
         [] nil then Ld
         end
      end

      proc{Discrim Lg}
         GM = {Result Lg} in thread {R GM} end {Discrim {Barrier Lg nil}}
      end

      LG = {Launch L nil}
   in
      {Discrim LG}
   end
```

15.10.3 Structural patterns

Arbitrary cycles

> Pattern description: The ability to represent cycles in a process model that have more than one entry or exit point.

Modelling a process model with multiple entry and exit points, and arbitrary cycles between its tasks, can be done directly in Oz e.g. by modelling each task in the process model as a port object (see [144] chapter 5 for a definition of port object). It is also possible to model each task as a FrucToz component (that extends the port object idea), and to model connections between tasks as FructOz bindings (see Section 15.6). This pattern is not amenable to a formalization in the form of a single higher-order Oz procedure as the preceding patterns, because of its free form character. However, the bindings in FructOz illustrate how to link an arbitrary output (client) stream to an input (server) port. For the sake of illustration, the StrangeLoop procedure below shows a simple loop: when being called, it creates a port object, which can receive messages from its environment on two ports G and B; each message received on port G is forwarded to port B, and vice-versa (it is strange because messages received are endlessly forwarded around the loop...).

```
proc{StrangeLoop G B}
   Gs Bs
   proc{Handle Is O}
      case Is of M | Ts then {Port.send O M} {Handle Ts O} end
   end
in
   G = {Port.new Gs} B = {Port.new Bs}
   thread {Handle Gs B} end
   thread {Handle Bs G} end
end
```

Implicit termination

> Pattern description: A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future.

This pattern essentially stipulates that a task or process is deemed terminated when it can effect no further action. Modelling tasks as procedure executions in Oz means that a task is completed when the procedure returns and the threads it has spawned have terminated. Hence the implicit termination pattern is supported in our Oz interpretation. Note that it is possible to define in Oz threads with explicit termination detection: see [144] chapter 5 for the definition of a thread abstraction with termination detection.

15.10.4 Multiple instances patterns

Multiple Instances without Synchronization

> Pattern description: Within a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion.

The pattern can be modelled by the MultiInst operator below. Argument N to procedure MultiInst represents the number of task instances to create.

```
proc{MultiInst P N}
   if N > 0 then
      thread {P} end {MultiInst P N−1}
   else skip
   end
end
```

Alternatively, one can have a list version, modelled by the MultiInstL operator below. Argument L to procedure MultiInstL represents the list of inputs to be passed as arguments in calls to the unary procedure P. Each call to P represents an instantiation of same task (albeit with potentially different initial inputs).

```
proc{MultiInstL P L}
   case L of
      H | T then thread {P H} end {MultiInstL P T}
   [] nil then skip
   end
end
```

Multiple Instances with a priori design-time knowledge

> Pattern description: Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the activity instances at completion before any subsequent activities can be triggered.

This pattern is modelled by the MultiInstLD operator below, where the size of the argument list L, and the task procedure P that is part of the MultiInstLD environment, are known at design time. Note that MultiInstLD is a variation on the ParSyncL operator that models the Synchronization pattern. In contrast to the Synchronization pattern, note that the list L which appears as argument to the MultiInstLD procedure represents a list of argument to be passed to the different instances of task P created. Also, the inner function Mlf, which is used to create the different task instances, also accumulate in list Lr the different results produced by the different task instances. This list is then passed on to the subsequent task R for further treatment.

```
proc{MultiInstLD L P R}
   fun{Mlf L Z Lr}
      case L of
         H | T then
           ZH ZT in
            thread {P H#ZH} {Wait ZH} ZT = Z end
            {Mlf T ZT {List.append Lr [ZH]}}
      [] nil then Z#Lr
      end
   end
   Z#Lr = {Mlf L unit nil}
in
   {Wait Z} {R Lr}
end
```

Multiple instances with a priori run-time knowledge

> Pattern description: Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

This pattern is modelled by the same MultiInstLD operator above. The difference lies in the fact that the argument list L only becomes known at run time, prior to the call to MultiInstLD.

Multiple instances without a priori run-time knowledge

> Pattern description: Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these in- stances are independent of each other and run concurrently.

> At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

This pattern is modelled by the MultiInstR operator below. The MultiInstR creates a port object, whose port, G, is returned is functions in two phases. The first phase is similar to that of the MultiInstLD operator above. The second phase establishes a server thread that handles requests for new task instances being sent on port G. This second phase ends when the server thread receives an "end of task" (eOT) message on port G. When all task instances have terminated, task R is launched, with all the results from the terminated task instances as argument. Note that procedure MultiInstR terminates immediately after having created the port G, the server thread handling requests received on G, and the thread responsible for triggering the new task R upon the termination of all the task instances.

```
proc{MultiInstR L P R G}
   fun{Mlf L Z Lr}
      case L of
         H | T then
            ZH ZT in
            thread {P H#ZH} {Wait ZH} ZT = Z end
            {Mlf T ZT {List.append Lr [ZH]}}
      [] nil then Z#Lr
      end
   end
   fun{Mlp S Z Lr}
      case S of
         eOT | _ then Z#Lr
      [] r(M) | T then
            ZR ZT in
            thread {P M#ZR} {Wait ZR} ZT = Z end
            {Mlp T ZT {List.append Lr [ZR]}}
      end
   end

   Zf#Lf = {Mlf L unit nil}
   Zp Lp S
in
   G = {Port.new S}
   thread Zp#Lp = {Mlp S Zf Lf} end
   thread {Wait Zp} {R Lp} end
end
```

15.10.5 State-based patterns

Deferred choice

> Pattern description: A point in a workflow process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches present possible future courses of execution. The decision is made by initiating the first activity in one of the branches i.e. there is no explicit choice but rather a

race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

This pattern can be modelled by the DefChoice operator below. The operator takes as arguments: a record Tr of tasks (unary procedures), and a nullary choice function Cf that, when evaluated, returns a record of the form r(index: I input : M), where index designates the index, in the task record, of the task to trigger, and input contains the argument to be passed to the newly created task.

```
proc{DefChoice Cf Tr}
   Z = {Cf}
in
   {Tr.Z.index {Z.input}}
end
```

Interleaved parallel routing

Pattern description: A set of activities has a partial ordering defining the requirements with respect to the order in which they must be executed. Each activity in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time).

This pattern can be modelled by the InParRoute operator below. This operator takes a single argument POr, which is record whose fields are paris of the form P#L, where: P is unary task procedure, whose argument is a pair, with first element a termination variable, and with second element a list of input parameters; and L is a specification of the partial order among tasks, which takes the form of a list of features of POr, which indicate which tasks in POr must terminate prior to launching the task modelled by P.

```
proc{InParRoute POr}
   Zr = {Record.clone POr}
   Lk = {Lock.new}
   proc{Prepare I X}
      P#L = X
      proc{Prep L Lr}
         case L of
            H | T then {Wait Zr.H} {Prep T {List.append Lr [Zr.H]}}
         [] nil then lock Lk then {P Zr.I#Lr} end
         end
      end
   in
      thread {Prep L nil} end
   end
in
   {Record.forAllInd POr Prepare}
end
```

In the code of InParRoute above, we make use of the Mozart environment record module procedure forAllInd, which applies the same binary procedure to each field of

a record, with the current index in the record passed as first actual argument to the procedure. We also use the lock statement from Oz, which ensures an execution in mutual exclusion.

Milestone

> Pattern description: An activity is only enabled when the process instance (of which it is part) is in a specific state (typically in a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated activity can be enabled. If the process instance has progressed beyond this state, then the activity cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

This pattern can be simply modelled using the Milestone operator below. Note that a milestone is simply represented as a pair comprising a state variable V, and an associated result value R. The milestone is reached when the milestone variable is bound a pair with the particular state V as the first element.

```
proc{Milestone M V P}
   R in {Wait M} if M == V#R then {P R} else skip end
end
```

15.10.6 Cancellation patterns

Cancel activity

> Pattern description: An enabled activity is withdrawn prior to it commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed.

This pattern can be modelled with the CancelWrap operator below. CancelWrap essentially create a port object when provided with a procedure P. The latter takes the form of ternary procedure that records its termination by binding its third argument Out to some state value, and that takes as its first two arguments some message M and a state value In. Intuitively, the activity thus obtained takes the form of state machine, whose transition function is given by the procedure P. The operator CancelWrap ensures that the activity can be cancelled at any state.

```
proc{CancelWrap P In G}
   S
   proc{Handle S In}
      case S of
         cancel | _ then skip
      [] r(M) | T then Out = {P M In $} in {Handle T Out}
      end
   end
```

```
in
   G = {Port.new S}
   thread {Handle S In} end
end
```

Cancel case

> Pattern description: A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully.

Assuming that each activity creation in the target process instance takes the form of a call to CancelWrap, and that the list of the corresponding ports has been recorded, the pattern can be modelled by the CancelCase operator below, which just iterates over the list of ports to send the cancel message to every recorded activity.

```
proc{CancelCase Lp}
   {List.forAll Lp proc{$ G} {Send G cancel} end}
end
```

Chapter 16

D4.2a: First report on self-healing support

16.1 Executive summary

The work on self healing in the second year covered the abilities that are needed to complete or to complement the abilities of structured overlay networks. This work was done in three areas: (1) asynchronous failure handling in a network transparent system, (2) network partitioning and merging, and (3) transactional reconfiguration support. This work complements the work on transaction support over structured overlay networks which is reported in deliverable D3.1b.

16.2 Contractors contributing to the deliverable

KTH, UCL, and FT contributed to this deliverable.

KTH KTH contributed to the network partitioning and merge algorithm.

UCL UCL contributed to the Mozart 1.4.0 release.

FT FT R&D contributed to the transactional reconfiguration.

16.3 Introduction

Work on self healing was done in three parts:

- Asynchronous failure handling at the language level. We released the Mozart 1.4.0 system, which has advanced support for building fault-tolerance abstractions. The support is based on the notion of fault streams, a stream of messages from a failure detector attached to a language entity. Together with lightweight concurrency and network transparency, this allows complex fault-tolerance abstractions, such as those in [66] and Erlang's process linking, to be implemented in just a few lines of code.
- Network partitioning and merging. A major gap in the abilities of structured overlay networks was the inability for the overlay to merge together after a temporary network partition. This gap is now closed with the development of the merge algorithm.
- Transactional reconfiguration. A problem in reconfiguration of component-based systems is what to do when there is a fault during the reconfiguration process. One solution is to have transactional reconfiguration.

The work on transaction support over structured overlay networks is also relevant to self healing, but at the application level. This work is presented in deliverable D3.1b. We explain each of these topics in the sections that follow. The appendices contain papers that cover all the work in detail.

16.4 Asynchronous failure handling in a network transparent system

A distributed system can be made much easier to program if the language in which it is written is properly designed. The Oz language, implemented in the Mozart system, was extended before the SELFMAN project with a network-transparent distribution layer. This distribution layer did achieve a simplification of programs, but it exposed the distributed system in an overly complex way to the programmer. In the context of SELFMAN, we are interesting in exploring the simplest possible way that a self-managing system can be programmed. To achieve this, we have redesigned and reimplemented the distribution layer of Mozart to support asynchronous failure handling. This work is reported in the Ph.D. dissertation of Raphaël Collet (see Appendix A.2). We have made a public release of Mozart 1.4.0 in July 2008 (see `www.mozart-oz.org`) and we are currently retargeting all our SELFMAN work for this new system.

In SELFMAN, the abilities of Mozart 1.4.0 are used to simplify the construction of self-managing systems. Self-healing is simplified by the asynchronous failure detection and by the fine-grained concurrency (lightweight threads). Self-

configuration is simplified by the first-class components (based on closures and symbolic values) and the lightweight threads.

16.4.1 Network transparency

The Oz language is implemented in Mozart with a network-transparent and network-aware implementation. Network transparency means that the same program can execute over a network, distributed in any way, and (if there are no failures) it will have exactly the same functionality (only the timing of operations may change). Network awareness means that the language allows to predict and control the distribution and network behavior of the program. These two properties, taken together, give an enormous simplification of distributed programming. If your only experience with distributed programming is in a system such as Java with RMI, then you will have difficulty imagining the simplification that is possible.

Per Brand has given an example to illustrate the simplification that is possible. The full example is available at:

`http://www.sics.se/~seif/JavaVSMozart.html`

In this example, we write a distributed producer/consumer program where the producer generates a stream of data (one million integers) that is read by the consumer. The Mozart program for this is 32 lines of code, and exactly the same program runs on a single machine or over the network (this is network transparency). The Java program is 108 lines in the single-machine case and 220 lines in the distributed case (Java is not network transparent). Regarding performance, the differences are also striking: the Mozart program consistency runs much faster than the Java program. In the single machine case: 17.6 seconds versus 3.9 seconds, and in the distributed case, 1 hour versus 8.0 seconds.

The problems with Java are twofold: first, it has no specific support for asynchronous communication, which is essential for a distributed system, and second, it is not network transparent. The first problem can be overcome with an asynchronous communication library. The second problem cannot be overcome without redesigning the language. Any Java application inherently exposes the distribution structure of the underlying system it runs on. The Oz language implemented by Mozart does not expose the distribution structure. It is possible, for example, to write an application and test it on a single machine, and then to distribute the application over a network without rewriting any of its code.

16.4.2 Relationship with Kompics

The Kompics component model is inspired by the programming model of Mozart. KTH, which developed Kompics, has also worked on Mozart. For example, the channels in Kompics are very similar to ports in Mozart. Ports in Mozart were first introduced by KTH in the late 1990s. Experience of using ports in Mozart

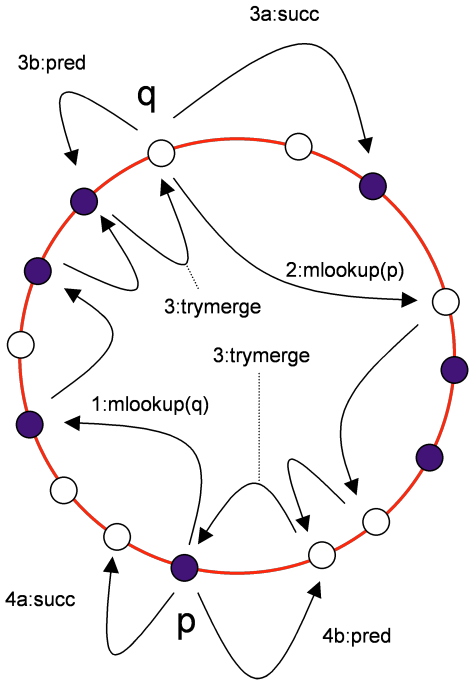


Figure 16.1: The ring merge algorithm

led to the proposal of channels in Kompics. Kompics also uses first-class component values and relies on lightweight threads, two characteristics that are directly derived from Mozart.

16.4.3 Asynchronous failure detection

In Mozart 1.4.0, we have made a major advance with respect to the previous work by making failure detection be asynchronous. Each distributed language entity (object, communication channel, dataflow variable) has an associated fault stream: a stream containing the tokens `ok`, `tempFail`, and `permFail`. Whenever the distribution behavior of the entity changes, a token is added to the stream. Together with a few other operations, the ability to change the distributed protocol that implements the entity, and the ability to globally or locally "kill" the entity, we can build sophisticated distributed abstractions (e.g., all the algorithms in [66]) in just a few lines of code.

16.5 Network partitioning and merging

Network partitioning is a real problem for any long-lived application on the Internet. A single router crash can cause part of the network to become isolated from

another part. SONs should behave reasonably when a network partition arrives. If no special actions are taken, what actually happens when a partition arrives is that the SON splits into several rings. We have observed this behavior for correctly implemented SONs. What we need to do is efficiently detect when such a split happens and efficiently merge the rings back into a single ring. We have designed and implemented an algorithm that does exactly this [129, 130]. Appendix A.15 defines the algorithm in detail. Here we give the main insights.

The merging algorithm consists of two parts. The first part detects when the merge is needed. When a node detects that another node has failed, it puts the node in a local data structure called the passive list. It periodically pings nodes in its passive list to see whether they are in fact alive. If so, it triggers the ring unification algorithm. This algorithm can merge rings in $O(n)$ time for network size n. We also define an improved gossip-based algorithm that can merge the network in $O(\log n)$ average time.

Ring unification happens between pairs of nodes that may be on different rings. The unification algorithm assumes that all nodes live in the same identifier space, even if they are on different rings. Suppose that node p detects that node q on its passive list is alive. Figure 16.1 shows an example where we are merging the black ring (containing node p) and the white ring (containing node q). Then p does a modified lookup operation (`mlookup`(q)) to q. This lookup tries to reduce the distance to q. When it has reduced this distance as much as possible, then the algorithm attempts to insert q at that position in the ring using a second operation, `trymerge`(pred,succ), where pred and succ are the predecessor and successor nodes between which q should be inserted. The actual algorithm has several refinements to improve speed and to ensure termination.

16.6 Transactional reconfiguration of component-based systems

Advanced component models such as Fractal considered in Selfman are fully dynamic and reflexive and make it possible to program dynamic reconfigurations, even unanticipated ones, to be executed in a running application. This is important in order to evolve applications without stopping and redeploying them (for example to update a component or subsystem). However, direct use of the Fractal APIs to program reconfigurations has several drawbacks: verbose and error-prone code due to the lack of language integration and minimalist design of the APIs, compilation and code deployment phases which complicate the process (in the case of Java). Using the bare APIs – especially in a general purpose language – also makes it difficult to guarantee the correctness of the reconfigurations: individually correct Fractal reconfigurations can result in globally incorrect reconfiguration depending on when and how they are executed with respect to each other and to the normal execution of the application. If tools of the selfman platform are to be

used to reconfigure applications during their execution, it is essential we guarantee reconfigurations are reliable.

Dynamic reconfigurations allow modifications of a part of a system during its execution without stopping it entirely so as to maximise its availability. Thanks to properties of component models like modularity and loose coupling, reconfigurations can rely on component-based architectures. However, runtime modifications can let the system in an inconsistent state and we identified three main reliability problems when reconfiguring systems:

1. A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functionnal execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hotswap example with a stateful component, calls on the old component must be blocked until a "quiescent state" is reached, then the state must be transfered, and finally previous calls are forwarded towards the new component.

2. A second problem at the model level is about consistency violation by reconfigurations. Component models and application models should define what this consistent system is. So we must ensure the conformity of the system to the model and what we call integrity constraints after reconfigurations.

3. The third and last problem is linked to the composition of reconfiguration operations. The semantics of reconfiguration operations implies there can be some conflicts between them in case of compostion and for synchronization between several reconfigurations.

Well-defined transactions associated with structural and behavioral constraints verification is a mean to guarantee the reliability of reconfigurations in component models. We revisit the ACID properties in the context of component-based systems:

- **Atomicity**: either the system is reconfigured or it is not. Each reconfiguration operation must specify its reversible operation. Thus if a reconfiguration transaction is rollbacked, it is possible to come back in a previous stable state by undoing operations. Transaction demarcation is either programmed in the language or automatic.

- **Consistency**: a transaction must be a correct transformation of the system state. So the reconfigured application must be conform to the component model and application-specific constraints. A reconfiguration transaction can be commited only if the resulting system respects the constraints. Other faults like software and hardware failures are the responsibility of the commit protocol.

- **Isolation**: several reconfiguration transactions are independant and any schedule of reconfiguration operations must be equivalent to their serialization. The scheduling must respect the operation semantics and conflicts.
- **Durability**: once a reconfiguration completes with success (commit), the new state is persistent. For every transaction, operations are logged in a journal so that reconfigurations can be redone in case of failure. The application state (architecture and component state) is periodically checkpointed so that any component can be recovered in its last stable state resulting from the last successful reconfiguration.

In our proposal, system consistency relies on **integrity constraints** both at the application and at the model level. An integrity constraints is a predicate which concerns the validity of an assembly of architectural elements but it can also concern component state. An example of such a constraint at the component model level is hierarchical integrity (bindings between components must respect the component hierarchy). Constraints must be checked both at compile time on the ADL configuration and at runtime. We represent the Fractal component model as a typed graph and then each fractal-based application is also a graph which is a well-typed instance of the typed graph and is provided at runtime by the reflexivity of the model. The vertexes are elements from the component model (components, interfaces, etc.) and the edges represent relations between the elements (composition links, binding links etc.) Then integrity constraints can be specified on the graphs with a constraint language "à la OCL", basically an extension of FPath with invariants, preconditions, postconditions.

```
// Example of a precondition for removing a component operation:
void removeSubComponent(Component sub);
preconditions:
// all interfaces of the sub-component are unbound
not(exists(sub/interface::*[not(bound(.))]));
```

To compose operations and regardless of a dedicated reconfiguration language, we consider sequences or parallel executions of intercession operations with conditions expressed by means of introspection operations but all compositions are not always valid. We want to make operation semantics explicit in terms of preconditions and postconditions with our constraint language and we want eventually to be able to change it and to specify new primitive operations. We distinguish two types of conflicts between operations:

- **Parallel conflicts**: for two given reconfigurations *R1* and *R2* executed on the same system, a parallel conflict occurs if *R1* and *R2* modify the same manageable elements in the system model (e.g. bind and unbind operations).

- **Execution dependencies**: an execution dependency occurs if *R1* either need *R2* to be executed first (e.g. stop before unbind)or if *R1* cannot be executed after *R2*. That is to say *R2* postconditions cover or not *R1* preconditions.

To deal with reconfiguration concurrency, we propose a pessimistic approach with locking based on operation semantics to avoid inconsistent compositions of operations. We see two different possibilities for the locking algorithm:

- The first one is to lock directly reconfiguration operations: either conflicts between operations are automatically calculated thanks to their preconditions and postconditions or conflict must be explicitely defined.
- The second one is to use a modified DAG locking algorithm on our instance graph defined in. Then the lock granularity is defined by the manageable elements in the graph representation (e.g., a lock acquisition on a component also locks all its interfaces and every operations in each interfaces).

In the context of SELFMAN we developed a transactional reconfiguration framework (see Appendix A.16) and its integration in the FScript language runtime of this new transactional backend (see Appendix A.17). We also worked on a multi-stage approach that allows for performing static analysis prior to actually execution reconfigurations so as to prevent the (tentative) execution of inconsistent reconfiguration transactions (see Appendix A.18).

Chapter 17

D4.3a: First report on self-tuning support

17.1 Executive Summary

The first part of this deliverable shows how self-tuning capabilities may be useful and used in order to allow for deploying an optimally pre-configured distributed application for best performance. A supporting architecture is described, based on the combination of a self-tuning system and a self-regulated load injection system. While the former system generates possible configurations, the latter system autonomously evaluates the performance of each generated configuration. Such an architecture actually provides a self-benchmarking system, autonomously supporting benchmarking campaigns. Implementation details are given about the self-regulated load injection system.

In the second part of this deliverable, we investigate the current state of load-balancing algorithms for DHTs. A distributed hash table (DHT) extends a structured overlay network (SON) with primitives for storing (key, value)-pairs and for retrieving a value associated with a key. DHTs support both direct key lookups [136, 123, 115] and range queries [127]. In the former case, a hash function is used to distribute the keys evenly among the nodes in the system. However, in a DHT supporting range queries this is not possible since the order of the stored keys would then be destroyed. Depending on the distribution of the keys the DHT can quickly become unbalanced. An unbalanced system can lead to network congestion and unresponsive nodes. In order to avoid this, the SELFMAN storage service is extended with algorithms that balances the load evenly over the nodes.

In the work presented here, we assume that by introducing local knowl-

edge about global state the performance of self-tuning algorithm for DHTs can be improved. We modify an existing load-balancing algorithm [81] with additional knowledge of the average load in the system. With this simple modification, our results showed a decrease of the overall data movement cost induced by the load-balancing. In addition, two centralized algorithms are discussed which can provide optimal solutions and thereby benchmark values used to indicated the performance of the decentralized algorithms. We plan to continue this work by improving the efficiency of the centralized algorithms and by evaluating the introduction of more complex global knowledge such as network topology.

17.2 Partners Contributing to the Deliverable

ZIB (P5), KTH (P2) and FT have contributed to this deliverable.

ZIB (P5) ZIB is contributing the work on self-tuning of DHTs in cooperation with KTH.

KTH (P2) KTH contributed to the work on self-tuning together with ZIB.

FT (P4) FT R&D contributed work on self-tuning before deployment through self-benchmarking.

17.3 Results

17.3.1 Introduction

The first contribution of this deliverable develops a self-benchmarking architecture. It combines self-tuning capabilities with a self-regulated load injection system. This combination allows for optimally configuring a distributed application or service before actually deploying and operating it. Our goal is to avoid deploying a badly configured application that may fail for performance reasons.

The SELFMAN storage service can in addition to exact match queries on a key also support range-searches [128]. With range-query support, a hash-function cannot be applied to the inserted keys since this would violate their order. In addition, the distance between keys is typically not uniform, e.g. the article names in Wikipedia. Therefore, a normal DHT with uniform distance between nodes would quickly become unbalanced when storing different key distributions.

In the second contribution of this deliverable we evaluate current load-balancing algorithms for SON-based DHTs. Our main goals are to a) minimize the overall network utilization and b) investigate how global parameters can be introduced to improve the performance of existing load-balancing algorithms. Furthermore, we present two centralized approaches, one based on tree search [88] and one based on auction algorithms [19]. The centralized algorithms are used to benchmark the decentralized algorithms.

17.3.2 Self-benchmarking

Introduction

Self-tuning is generally considered from a runtime capability perspective, used while an application or any other system is actually deployed and being operated. However, self-tuning is also an interesting autonomic capability before operating an application. As a matter of fact, common (and good) practice requires a qualification/validation step before deployment, including for performance requirements. This validation process typically involves a load test campaign, consisting in using a load injection tool to generate a flow of requests on the System Under Test (SUT) and observe its behavior (response times, throughput, call rejections, errors). The use of these performance measurements is twofold:

1. the SUT is qualified or not qualified according to the performance requirements;

2. the evaluation is used to size the SUT, for instance in terms of replication level (e.g. in the case of a distributed application, set the right number of servers, with the right power, for the right features).

But, in both situations, the question of the SUT tuning arises. In the first situation, a SUT may have been unfairly unqualified because of a bad configuration with regard to performance. In the second situation, the optimal sizing requires an optimization of the SUT in order to avoid a proliferation of servers. Optimal sizing is not only important with regard to the cost of buying servers, but also (and increasingly) with regard to the operation costs: human resources for system management and maintenance, energy consumption (including for air conditioning), hosting space, etc.

This pre-deployment tuning step can be considered as a benchmarking activity, since it relies on measuring the performance of a number of SUT configurations and compare results in order to choose the best configuration. Next section describes the benchmarking process and explains why self-tuning is relevant to this activity. Next sections focus on specific autonomic features that are necessary for self-benchmarking, namely self-regulated load injection and self-tuning.

Introduction to load testing and benchmarking

Load testing campaigns consist in generating a flow of client requests on a SUT in order to assess its performance and sustainable throughput. As shown by figure 17.1, a load testing infrastructure is typically composed of:

- one or several load injectors sending requests to a SUT and waiting for responses to measure the corresponding response times;
- probes measuring the usage of computing resources, at the SUT side, to help detecting performance problems, as well as at the load injection side, in order to check that it is performing as expected;
- a supervision user interface to deploy, control and monitor the distributed set of load injectors and probes;
- a storage space to gather all measures (e.g. as a set of log files or a database);
- tools for post-mortem analysis and report generation.

The traffic generated by the load injectors are commonly modeled through the definition of virtual users, i.e. programs that emulate the behavior of

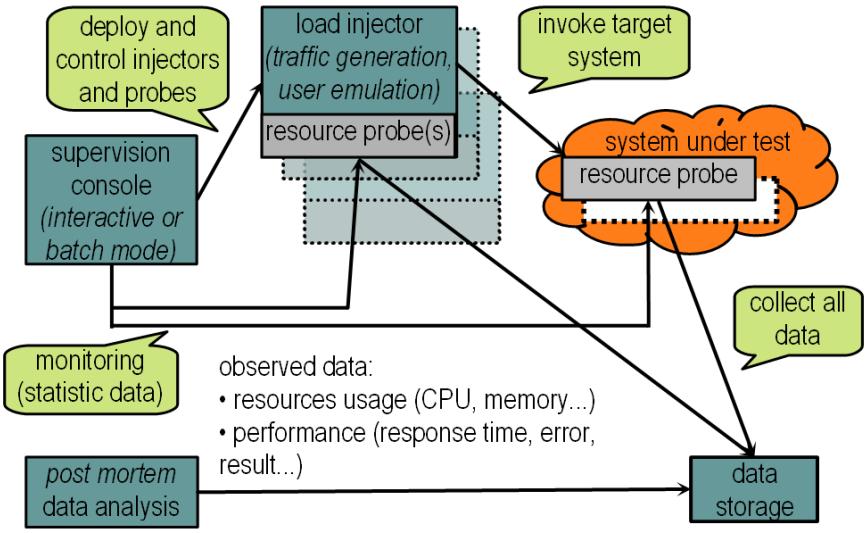


Figure 17.1: Big picture of a typical load testing infrastructure.

real users, through successions of requests and think times (time spent by a user between 2 consecutive requests). For a given SUT, there are often a number of different typical usages, thus resulting in defining a number of different virtual users exhibiting different behaviors. In the case of web applications, some users may just consult available information, while others will strongly interact and trigger complex processes, resulting in different usage of computing resources and thus different impact on performance. For instance, some behaviors induce database write operations while others don't.

Performance benchmarking aims at comparing and ranking a variety of options such as configuration parameters or alternative implementations of hardware or software implementations, from a performance point of view. Benchmarking relies on load testing campaigns where the SUT must be tuned for optimal performance in order to obtain a meaningful ranking. As a matter of fact, comparing results from an optimally configured SUT with results from a badly configured alternative would make no sense.

Towards self-benchmarking

It typically takes a lot of manpower, skills and time to carry a load test campaign out. The test infrastructure is a complex combination of the load injection system, probe system, and SUT involving a tremendous number of parameters that are likely to strongly interact with each other (e.g. size of buffers, pools of database connections, size and policy of caches, network

configuration, multi-threading policy...). Testers must be experts in every element of the global system (hardware, software, operating system, middleware, network equipments and protocols...) in order to handle troubleshooting and performance optimization. In an empirical and iterative process, tests are repeated again and again with different parameters arrangements and different configurations until sufficient confidence and satisfaction about results are met. Then, we see that testers behave like a feedback/control loop, observing the SUT and the load injection system on the one hand, and modifying the SUT and load injection configuration on the other hand as a reaction to observations.

Self-benchmarking consist in considering that the tremendous complexity of the whole computing system used in a benchmarking campaign justifies an autonomic computing approach, that is: try to use computing power to autonomously deal with the computing system complexity. In other words, self-benchmarking shall carry out test campaigns by autonomously controlling the load injection system and the SUT configurations, with the objective of maximizing performance[1].

Self-regulated load injection

The first step in benchmarking generally consists in searching the approximate performance limit of the SUT in a given configuration. For example, in the context of application servers, the tester would typically try to find out the maximum number of users that the SUT may sustain with regard to given saturation criteria. Common criteria are expressed in terms of response time, request rejection, error occurrence, or computing resource shortage. A common way of looking for the saturation limit is to run a variable (generally growing) number of virtual users and look for the saturation point. Another way of varying the load injection is to change the proportion between the different virtual user families (i.e. of different behaviors). During this experimental search, the tester plays an empirical feedback role on the load injection system: according to the distance between the observation and the given saturation criteria, the tester more or less increases (or possibly decreases) the load.

Figure 17.2 presents a load testing infrastructure featuring autonomous search of saturation point. This infrastructure is composed of:

- the system under test;

[1]Of course, the concept of performance may be practically mapped to a variety of criteria, such as request throughput, rejection or error rate, response times, number of users, etc.

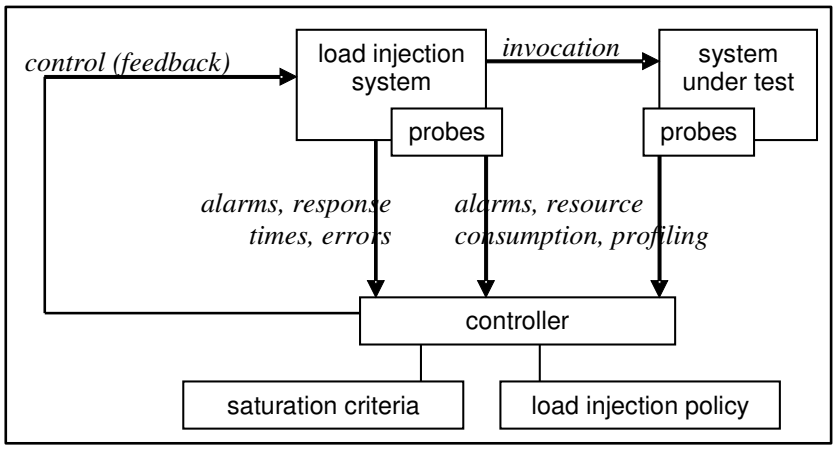


Figure 17.2: Self-regulated load injection for autonomic search of system performance limits.

- the load injection system, made of one or several traffic generators depending on the required load level;
- probes measuring computing resources usage at the SUT, including for instance typical system probes (CPU, memory, network...) as well as possibly probes related to specific software elements involved in the environment (middleware, database...);

Since the concept of saturation may be practically characterized in a number of ways, the architecture of our self-regulated load injection system also exhibits a component dedicated to provide and isolate the saturation criteria and feedback (load injection) policy.

To practically implement this self-regulated load injection system, we can rely on the open source, component-based CLIF load injection framework [43]. This framework provides generic load injection and probe components, that can be controlled (e.g. to dynamically change the load injection level) and monitored through a supervisor component. We just have to define a load injection controller component and bind it to the supervisor component to implement the self-regulated load injection feature. This architecture has been presented in [70] and used to automatically find the saturation limit of an XML appliance in the context of Service Oriented Architectures. For illustration purpose, we reproduce the self-saturation experiment results in figure 17.3. The load injection system starts with a minimal workload of 1 virtual user. Step by step, the number of virtual users is increased (or decreased) in order to reach 80% system load (chosen saturation criteria)

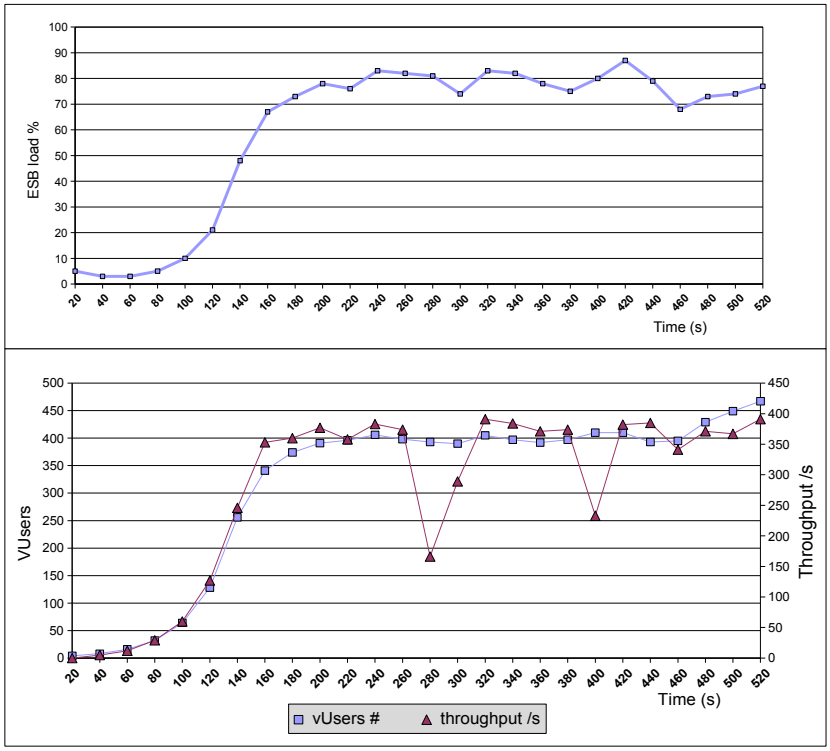


Figure 17.3: Autonomic saturation search with self-regulated load injection applied to an XML appliance.

for the XML appliance (named ESB load in the figure). It takes about 3 minutes to reach and maintain this limit, dynamically adjusting the number of virtual users.

Adding self-tuning

The ultimate goal of benchmarking is to qualify the optimal performance of a system and to compare it to other similar but different (in configuration or implementation) system. Therefore, finding the optimal settings of the tested system, in other words tune it, is key to the autonomic benchmarking principle. This second step includes step one since benchmarking requires to reach the maximum performance (i.e. the saturation limit). For instance, the result of step one could be the conclusion that the SUT in a given configuration sustains a number of active virtual users representing a given mix of a number of behaviors. Then, the question is: is that result the best the SUT can deliver, or is it improvable by tuning the SUT? In other words, the

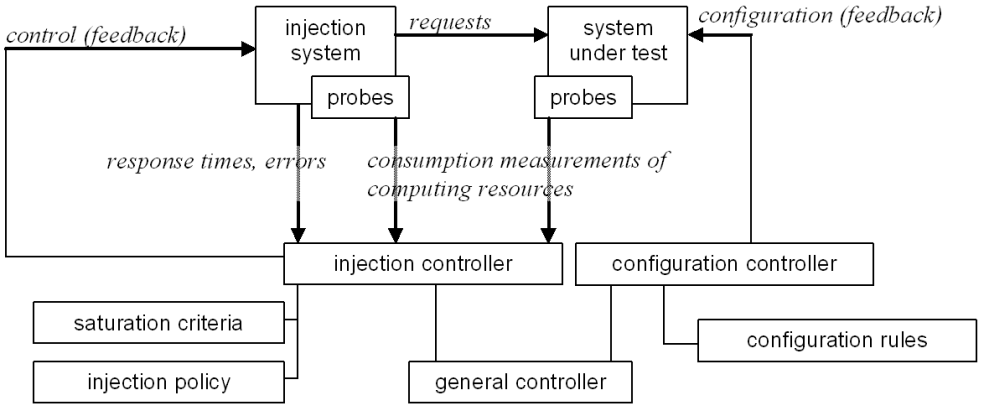


Figure 17.4: Autonomic benchmarking: self-tuning of a system under test and autonomic search of performance saturation

tester will try to optimize the SUT and rerun the load test until s/he has the conviction that the maximum performance has been reached. This is, of course, a matter of estimation, whose accuracy depends on the skill and experience of the tester, since the combinatory of tuning parameters, and the complexity of interactions between them are so huge that a full exploration of the solution space is not humanly feasible.

Self-tuning allows for replacing the tester with a second control loop introducing an autonomous optimization of the SUT. Synchronization must be achieved between self-tuning and self-regulated load injection processes in order to alternate in a consistent way optimization phases and saturation search phases. The resulting architecture (see figure 17.4) adds a configuration controller, observing the maximum system performance (i.e. achieved at the saturation limit) reached for current configuration, and generating new possible system configurations. SUT-specific configuration rules must be provided in order to identify possible tunable parameters and their possible values. A general controller component orchestrates the configuration controller and the load injection controller.

Self-benchmarking relies on a classical generate-evaluate process which is supposed to explore a multi-dimensional space of solutions. Roughly speaking, the tested elements can be configured by setting a number of parameters, with a range of possible values. A number of issues arise then:

- some parameter values may be incompatible with each other;
- some parameters may be correlated;

- the number of parameter values combinations may be far too huge to be able to explore them all.

Factor analysis statistical techniques may be used in order to identify and eliminate parameters that don't influence performance. Heuristics may also be introduced in order to guide the exploration in an efficient way. Finally, a sufficiently good configuration, albeit not the best configuration, shall be found.

Conclusion

This section has presented how to apply self-tuning capabilities before the actual deployment and operation of an application. It consists in generating possible application configurations and evaluate them. This approach requires to add a self-regulated load-injection system that allows for autonomously finding the performance limits of each configuration. This combination can be considered more generally as a self-benchmarking system: not only it allows for deploying an optimally configured application (with regard to performance concerns), but it also allows for comparing performances of alternative configurations which is the key consideration of benchmarking.

Finally, note that the self-regulated load injection system, based on CLIF load injection framework, is likely to be very useful when the Selfman project will perform some experimental assessments of their applications and autonomic scenarios, in order to generate a user traffic.

17.3.3 DHT Load-balancing

A distributed hash table (DHT) extends a SON with primitives for storing (key, value)-pairs and for retrieving a value associated with a key. DHTs support both direct key lookups [136, 123, 115] and range queries [127]. In the former case, a hash function is used to distribute the keys evenly among the nodes in the system. However, in a DHT supporting range queries this is not possible since the order of the stored keys would then be destroyed. Depending on the distribution of the keys the DHT can quickly become unbalanced. An unbalanced system can lead to network congestion and unresponsive nodes. In order to avoid this, the DHT is extended with algorithms that balances the load fairly over the nodes depending on their capacity [114].

Figure 17.5 shows an example of a load-balanced system storing a data-set from the wikipedia demonstrator presented in D5.2a. In this scenario, entries containing text with a certain language are more likely to be accessed from the countries with a high population speaking the language. Thus, placing

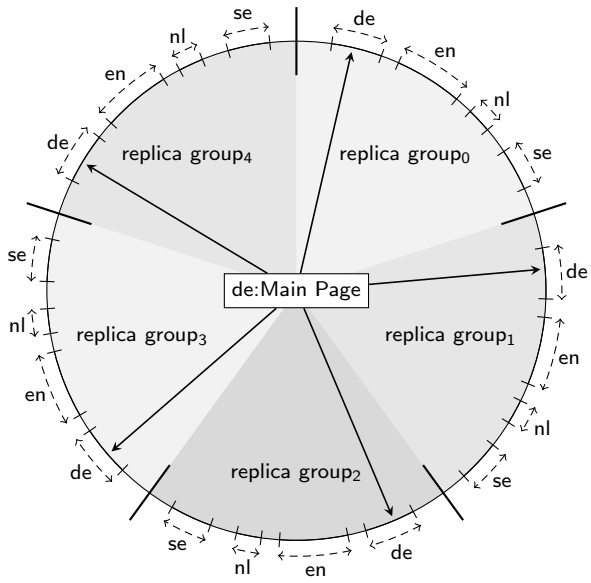


Figure 17.5: Geographic Load-Balancing for Wikipedia.

replicas in the country or nearby tends to improve client latency. A load-balancing algorithm that considers the data locality can therefore indirectly be used to improve server placement.

Recent developments in gossipping for unstructured P2P-networks has shown that it is possible to estimate global properties with high confidence [151, 150]. For load-balancing, proximity information has proven useful in order to improve the network utilization [159, 133]. However, not only the node topology is interesting, knowledge about for example the average node utilization could potentially be used to improve load-balancing efficiency.

With I/O intensive applications using the DHT, such as the wikipedia demonstrator, the network can easily become a bottleneck. It is therefore important that the algorithms used for tuning and DHT maintenance use the network efficiently. This is especially the case for load-balancing algorithms since their main operations trigger data movements [81, 53]. By introducing more global properties there is a trade-off between the gain and the cost in terms of network usage.

This deliverable explores the effects of introducing knowledge of global properties to a well-known decentralized load-balancing algorithm [81] which only considers local knowledge. In addition, we discuss two centralized approaches, one based on the auction algorithm [19] and one on tree-search [88]. The centralized approaches finds the optimal solution or a close approxima-

tion which can be used as a comparative benchmark of the decentralized algorithms.

Related Work

Load-balancing algorithms in DHTs focuses on three different problems. First, in a DHT where each item is hashed uniformly over the ID space, some nodes can have an $O(\log N)$ imbalance in terms of stored items [114, 61]. Second, in DHTs with range-query support, the items must be mapped to the ID space in order, keeping their original distribution. Therefore, for the system to be balanced, i.e. nodes store an equal number of items, their IDs must be distributed according to the key distribution [81, 53]. Third, independent of the item distribution, certain items can have much higher request rates than others. This is typically solved through caching, replication together with exploiting redundant network routes [38]. The last issue is not covered further in this deliverable.

Virtual Servers is a technique where each physical node maintains a set of virtual nodes. Balancing of the load is done by moving virtual servers from overloaded physical nodes to more lightly loaded physical nodes. The assignment of virtual nodes to new physical nodes is typically performed at a directory node. A directory node periodically receives load information from random nodes in the system. When it has received load data from a sufficient amount of nodes it executes the load-balancing algorithm [114, 61, 31].

Virtual Servers increases the routing table state maintained at each node. In [62] Godfrey, et. al. introduces a scheme where physical nodes hosts virtual servers which have overlapping links in the routing table. With this placement restriction, a physical node only needs $\theta(logN)$ while hosting $\theta(logN)$ virtual servers.

Another issue with virtual servers is that a physical node failure causes the hosted virtual nodes to fail as well. This increases the churn in the system and must be considered when selecting global parameters such as the replication factor. In [94], Ledlie et. al. presents the k-Choice algorithm wherein each node samples load from a small set of IDs and directs joining Virtual Servers to overloaded nodes. They show through simulation that this decreases the amount of load-balancing induced churn.

The above approaches uses simple metrics for the cost of the load-balancing operations, e.g. the number of transferred items or bytes. However, a better cost-metric should include the overall network utilization. [159, 133] are both investigating the effects of proximity-aware load-balancing algorithms for Virtual Servers. In [31], the assignment of Virtual Servers to physical

nodes is modeled as an optimization problem which allows for an arbitrary cost function.

Item-balancing Most of the research on load-balancing in DHTs have focused on Virtual Servers. However, these approaches assumes that items are uniformly distributed over the ID space using a hash-function. For a data-structure without hashed items, a single virtual server can be overloaded if it is responsible for a popular ID range. For example, when storing a dictionary, keys with the prefix "e" are more common than "w", resulting in the node responsible for "e" to store more items. The goal of item-balancing schemes is to adapt the location of the nodes in the system to correspond to the item distribution. This is done using two operations, jump and slide. Jump allows a node to move to an arbitrary ID in the system, while a slide operation only exchanges items with a nodes direct neighbors.

In [81] Karger et. al., presents a randomized item balancing scheme where each node contacts another random node periodically. If the load of the nodes differ with more than a factor $0 < \epsilon < \frac{1}{4}$, they share each others load by either jumping or sliding. Karger provides a theoretical analysis of the protocol, but does not evaluate the algorithms in an experimental or real-world setting.

Ganesan et. al. [53] uses a reactive approach which triggers a recursive algorithm when the node utilization super-seeds a threshold value. A node executing the algorithm first checks if it should slide by comparing the load with its neighbor's load. If this is not possible, it finds the least loaded node in the system and request that it jumps to share the overloaded node's load.

We are basing our algorithms on the work presented by Karger, but introduce global parameters available at each node through a gossipping protocol. In addition, we develop two different centralized algorithms used to evaluate the global parameters applied to the decentralized approach.

System model and problem definition

A DHT consist of N nodes, where each node has an ID in the range $[0, 1)$. A node has a *successor*-pointer n_{i+1}, to the node with the next largest ID, and a *predecessor*-pointer n_{i-1} to the node with the previous largest ID. The node with the largest ID has the node with the lowest ID as successor. Thus, the nodes and their pointers can be seen as a ring or double linked list where the first and last item links to each other.

The DHT stores a set of items I, where each item has an ID in the range $[0, 1)$ and a weight. Each node stores a subset of items with IDs that falls in a nodes range $(n_{i-1}, n_i]$, i.e. the node is *responsible* for the ID. Figure

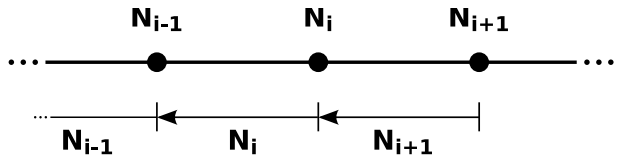


Figure 17.6: A node N_i with successor and predecessor and their respective responsibilities.

17.6 shows three nodes and their respective responsiblilities. Each node has a capability $c(n_i)$ indicating the amount of data it can store. The utilization for a node is defined as the fraction $u(n_i) = \frac{\sum_{i=0}^{I} w(I_i)}{c(n_i)}$.

The system has an average utilization $U(N) = \frac{\sum_{i=0}^{N} u(n_i)}{N}$. We say that a system is balanced when the utilization of all nodes fall within a range $U(N) \pm \epsilon$, where ϵ is a user-defined parameter. Increasing ϵ leads to a system with higher variation of node utilization, but it also lowers the cost to reach a load-balanced state. When $u(n_i) > U(N)+\epsilon$, we say that node i is *overloaded* and when $u(n_i) < U(N) - \epsilon$, the node is *underloaded*. The remaining nodes that fall within the range $U(N) - \epsilon \leq u(n_i) \leq U(N) + \epsilon$ are *balanced*.

In order to change the utilization of nodes in the system, two type of operations are used: jump and slide.

Jump allows a node to move to an arbitrary position in the ID space. A jumping node n_i first leaves its current position and re-joins at its new location, l_k. Data is moved two times, first the range $(n_{i-1}, n_i]$ is transferred to n_{i+1}. Second, when n_i joins at l_k, all data in the range $(n_{j-1}, l_k]$ is transferred from n_i's new successor n_j.

Slide is a specialized form of jumping where a node moves to an ID in the range (n_{i-1}, n_{i+1}). When moving to an ID $< n_i$, the node move the items in $(ID, n_i]$ to n_{i+1}, while when moving to an ID $> n_i$ the node becomes responsible for the items in (n_i, ID) in addition to its current responsibilities.

Problem definitions. The load-balancing problem can be summarized as follows: given a configuration $\mathbf{C}_0$ with a set of nodes $\mathbf{N}$ and items $\mathbf{I}$, where each item i_j is assigned to a responsible node, find a configuration $\mathbf{C}_b$ that only contains balanced nodes using the operations join and slide. A solution to the load-balancing problem is a set of (operation, iteration, node, ID)-tuples indicating what operation at which iteration the given node should use to reach the new ID.

In addition to the load-balancing problem, we search for an optimal solution set that minimizes the data movement cost of the transition from $\mathbf{C}_0$ to $\mathbf{C}_b$. The cost-function is defined as $cost(n_i, l_i)$, where n_i is a node and l_i is an ID (location) to where the node will jump or slide. The cost-metric can be chosen arbitrarily, but is typically based on the number of bytes moved or the network utilization.

The minimum node utilization required to take the system from any configuration to $\mathbf{C}_b$ is the sum of the distance for all overloaded nodes to the upper utilization limit as well as the sum of the distance for all underloaded nodes to the lower utilization limit. More formally, $\sum_{i=0}^{overloaded} u(n_i) + \sum_{j=0}^{underloaded} u(n_j)$.

Item balancing heuristics In order to reach a load-balanced configuration, we rely on the heuristics introduced for Karger's item balancing algorithm [81]. Expressed in our notation, a load-balance operation is only performed between any pair of nodes n_i, n_j, iff $u(n_i) \leq \epsilon u(n_j)$ or $u(n_j) \leq \epsilon u(n_i)$. When these restrictions are satisfied, the following cases are possible (assuming $u(n_i) \leq \epsilon u(n_j)$).

Case 1 n_i is underloaded and $n_j == n_{i+1}$ and is overloaded. Slide n_i towards n_j, letting n_i take responsibility for a fraction n_j's items.

Case 2 n_i is overloaded and $n_j == n_{i+1}$ and is underloaded. Slide n_i towards n_j, letting n_j reduce n_i's load by taking responsibility for a fraction of n_i's items.

Case 3 When n_j is not a successor, then n_i jumps to a location between (n_{j-1}, n_j).

Heuristics to minimize data transfer. The order in which the join and slide operations are performed influences the total amount data transfered. First, consider a set of neighboring underloaded nodes as depicted in figure 17.7. If node X leaves the system before node Y, the data from X is transferred to Y. When Y leaves the system, the data from X + Y is transferred to node B. While, if node Y leaves before X, its data is transferred to B. Later, when node X leaves, it transfers its data directly to B. Thus, when a set of neighboring underloaded nodes are leaving the system, the nodes should leave in ascending order to avoid redundant transfer of data.

The second heuristic considers when a set of nodes take over responsibility from an overloaded node as depicted in figure 17.8. This is similar to the previous case since if node Y joins first, it will transfer all items in the range

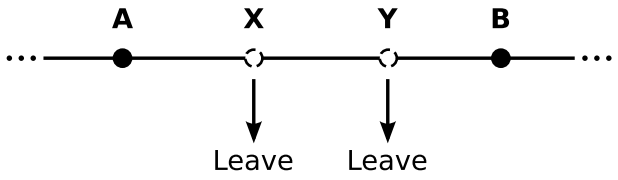


Figure 17.7: A chain of underloaded nodes leaving the system.

$(A, Y]$. Thereafter, when node X joins it will transfer the data in range $(A, X]$ which is now held by node Y. If X joins first, the data in the range $(X, Y]$ will not be transferred twice. Thus, when a set of nodes are joining the system at the same node, the nodes should join in descending order.

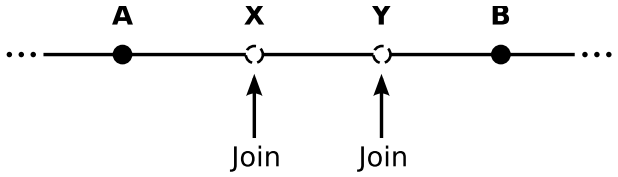


Figure 17.8: Two consecutive free slots being filled by joining nodes.

Centralized Algorithms

In a centralized algorithm the global state of the system is known by an oracle. The centralized approaches presented below are used as comparative benchmarks for the decentralized algorithms. We discuss two approaches for centralized algorithms. Tree-search, where a decision tree describing all possible choices for each configuration is traversed using depth-first search [88]. The second approach is based on auction algorithms [19] where overloaded and underloaded are matched to find an optimal assignment.

Tree-search The goal of the tree-search algorithm is to find a solution with the lowest possible total cost. A node in the tree is a system configuration, C_i, and an arc is representing a node's decision resulting in a new configuration C_j. A node decision is an operation, jump or slide, and a location. Each decision resulting in a new configuration has a cost. The solution with lowest possible cost is therefore the path in the tree with the lowest cost. Figure 17.9 shows an example sketch of a search tree.

The expansion at each node in the search tree is limited to the possible operations that any node can perform. The available operations for a node

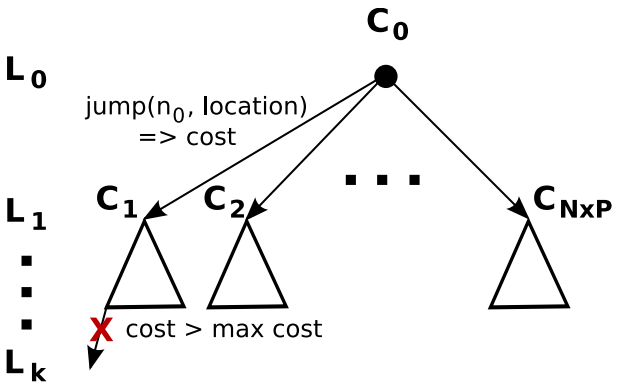


Figure 17.9: Sketch of a search tree.

are as follows: (1) Slide, if the node is underloaded and its successor is overloaded or vice verse, (2) Jump, if the node is underloaded. Note that balanced nodes are not performing any operations. However, depending on the movements of underloaded nodes, balanced nodes can become overloaded. An underloaded jumping node can choose any free position at any overloaded node. Since the number of overloaded and underloaded nodes decrease after performing an operation, there will be fewer choices towards the end of a search. The complexity of the tree search is in the worst case $O((N * d)^o)$, where N is the number of nodes, d is the maximum number of decisions for any node and o the number of operations needed to reach a load-balanced state.

In order to lower the amount of computation, the tree is only searched up to a given max cost, α. This is also known as depth-limited search (DLS). However, since DLS returns after the limit has been reached it will return the first solution, but not necessarily the solution with lowest cost. Therefore, we apply a variant of iterative deepening depth-first search [88], called iterative lengthening, which tries alternative paths up α until the solution with minimum cost is found.

An alternative approach is to use randomization, as done by Karger in [81] for a decentralized algorithm. Instead of expanding the tree with each possible node operation, an operation is chosen at random. Thus, the complexity is lowered to $O(N^o)$, but without the guarantee of finding a solution with minimal cost. However, since Karger's algorithm is a well-known decentralized load-balancing solution, it is interesting to study the variation of its performance with access to global knowledge.

The Auction Algorithm Since the complexity of the tree-search algorithm increases exponentially, we are currently investigating an alternative approach involving auction algorithms [19]. An auction algorithm finds an optimal one-to-one assignment of persons to objects in polynomial time. The assignment depends on the cost of the object and the benefit of the person being assigned to the object. For the load-balancing problem this is analogous to finding a lowest cost match between underloaded and overloaded nodes.

More formally, each person i has a benefit a_{ij} of selecting an object j with price p_j. The net value for a person i of choosing object j is $a_{ij} - p_j$. The goal of the auction is to find an assignment where all persons finds an object which maximizes their net value. Thus, an auction is finished when the equilibrium $a_{ij} - p_j = \max_{j \in Objects}(a_{ij} - p_j)$ is satisfied.

Each iteration of the algorithm consists of a bidding phase followed by an assignment phase. During the bidding phase, each person finds an object resulting in maximum net value after which it computes a bidding increment. The value of the bidding increment is used after the assignment phase to increase the price of the object. In the assignment phase the persons with the highest bids are assigned to the respective objects. When all persons are assigned to an object the algorithm terminates. This also means that the equilibrium has been satisfied [18].

When applying the auction algorithm for the load-balancing problem, an underloaded node translates to a person and an overloaded node is seen as an object. In a single iteration, an auction returns an assignment between underloaded nodes and overloaded nodes that minimizes the data transfer cost. This assignment is used to relocate the underloaded nodes. Relocation is currently done within a single iteration according to the heuristics presented in section 17.3.3. Subsequent iterations are executed in the same way until there are no more unbalanced nodes. That is, for any node i, $\epsilon \max_{i=0}^{N}(u(n_i)) \leq u(n_i) \leq \max_{i=0}^{N}(u(n_i))]$, where epsilon $0 < \epsilon < \frac{1}{4}$.

We implemented the described algorithm in a simulation environment. Figure 17.10 shows the number of moved items for increasing values of epsilon. The system has 100 nodes and the data-set is an American dictionary with 380645 entries. The figure shows that epsilon can be used to control the data movement cost necessary to reach a load-balanced configuration. That is, when epsilon goes towards zero the difference in node utilization when nodes are compared is relaxed.

An advantage of the auction algorithm is that the benefit function and the object price can be chosen arbitrarily. This allows us to explore more complicated costs than e.g. moved data items. Furthermore, the order of the load-balancing operations slide and join are affecting the total price of the

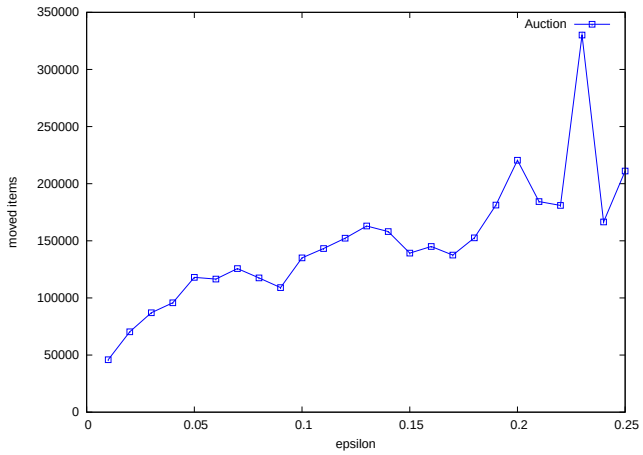


Figure 17.10: An increasing value of epsilon decreases the allowed difference between node utilization.

load-balancing process. Due to the apparent advantages in computational complexity of the auction algorithm approach, we are actively investigating an appropriate cost-function which can include proximity information and the order of operations.

Decentralized algorithms

Unlike the centralized algorithms, a decentralized algorithm can only use the information locally available at each node. We modify Karger's randomized item-balancing algorithm to work with different globally known parameters, for example, the systems average load.

Perfect global knowledge is not available in a distributed system without using expensive aggregation algorithms. However, by using gossiping techniques such as Vicinity and Cyclon [151, 150] it is possible to get a good approximation of a parameter's value with low network traffic overhead. Initially, we are interested in the parameters below.

Average Load The average system load can be used by each node to decide if it is underloaded or overloaded or within the balanced range.

Location Proximity-information allows a node which will transfer load to select a target node which minimizes the network utilization [159, 133].

Over- and Underloaded nodes A list of the k most overloaded and most underloaded nodes. These lists can be used for example in the Karger-algorithm to improve the convergence rate.

Load Error Margin The error margin indicates how much a node's load can differ from the average load. By changing this parameter it is possible to tune the aggressiveness of the load-balancing algorithm.

We implemented Karger's algorithm and one version with knowledge of the systems average load in a simulation. Using the average load, we apply the heuristics to minimize data transfer from section 17.3.3. When a node jumps to another node, instead of splitting the sharing the node load in half, the minimum of the average load and the shared load is used. The effect of using the extra information on average load is especially distinct in figures 17.14 and 17.13.

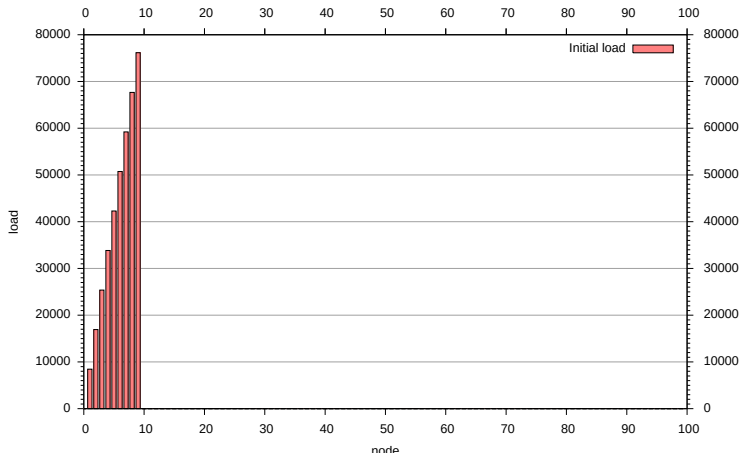


Figure 17.11: Number of items per node.

The simulation was set up with 100 nodes and an initial load distribution as shown in figure 17.11. Each simulation step tries to perform a load balancing operation for each node using a normal Karger load balancing algorithm and the one with knowledge of the system's average load. The algorithms were each tested with different ϵ values in the range of $0 < \epsilon < 0.25$ as suggested by Karger.

Figures 17.12 and 17.13 show how the algorithms perform using different ϵ values taking into account the total number of moved items during the simulation and the standard deviation of the load distribution among the

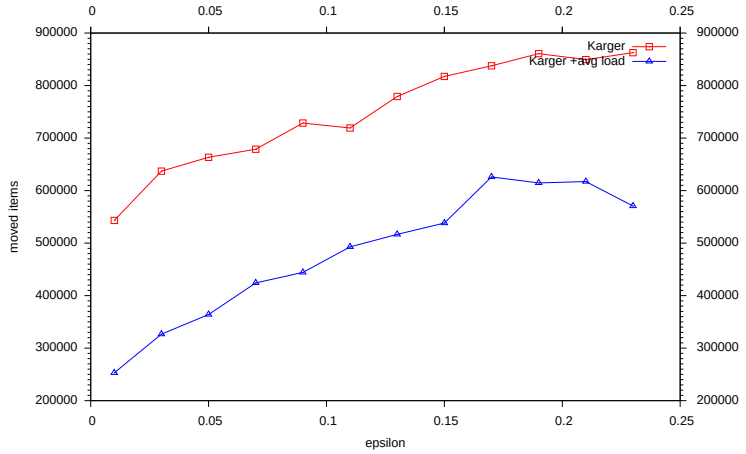


Figure 17.12: Absolute number of moved items for a network with 100 nodes with increasing epsilon for Karger and Karger with average load information.

nodes at the end of the simulation. The standard deviation is used to measure the degree of imbalance of a given load distribution.

Figure 17.14 shows how the standard deviation decreases as the load balancing algorithms move items from one node to another either by shifting the address range of two adjacent nodes or by moving a node to a different position.

Discussion

We showed that it is possible to improve the absolute cost of data movement by introducing simple heuristics and knowledge about basic global parameters. We plan to continue this work by evaluating the effects of more complex global knowledge such as the network topology. In addition, by improving centralized algorithms that can find the optimal solution, we can effectively evaluate the result of changes to the decentralized algorithm. Furthermore, there are also other aspects of the storage system such as replication, transactions and churn that should be considered in combination with load-balancing.

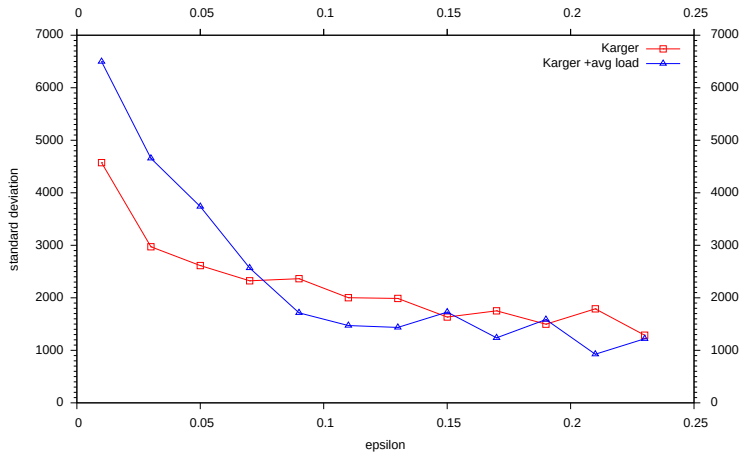


Figure 17.13: The imbalance with increasing number of epsilon for Karger and Karger with average load.

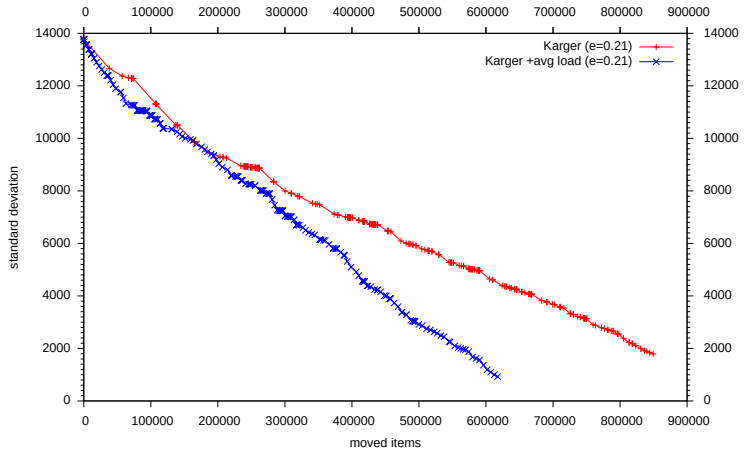


Figure 17.14: The load imbalance as a function of the number of moved items.

Chapter 18

D4.4a: First report on self-protection support

18.1 Executive Summary

In deliverable D1.3a and D1.3b (Section 4), we proposed and built a testbed for testing Small World Networks (SWN). The reason for investigating SWN is that they have better properties with respect to some of the drawbacks of Distributed Hash Table (DHT) based structured overlay networks. SWN have trust and identity relationships which mitigate the serious problems caused by Sybil-type attacks in DHTs. They are also more robust because of the random nature which is partly between a structured network and a random one.

Here, we investigate SWN as a suitable more secure self organizing network which may be able to replace DHT-based structured overlay networks in situations which require self-protection against Sybil and DHT maintenance attacks. In D1.3b, we describe the SWN testbed and simulator as well as giving background on SWNs.

D4.4a investigates self-protection using two kinds of SWNs. One is where the SWN has a global property on the identifier of nodes (the kind of SWN proposed by Kleinberg). So far our experiments indicate that the routing success rate is very high. This suggests that a SWN may be a feasible as a replacement for structured overlay network in scenarios where there can be malicious nodes or users.

The other kind of SWN does not need any special property on the node identifier. Rather it attempts to reorganize the node identifiers so that it resembles the first SWN. We have identified some attacks which attempt to poison the reorganization process and delete node identifiers. We have also

investigated some initial security mechanisms against these attacks based on self-tuning ideas (these happen to fit well with SELFMAN).

18.2 Contractors contributing to the Deliverable

NUS (P7) has contributed to this deliverable.

NUS (P7) NUS has designed and implemented the Small World Network testbed and simulator which has been used to investigate two initial self-protection mechanisms: Small World Network with global identifiers as a SON; and a self-tuning protection mechanism for Small World Networks which use Kleinberg-style reconstruction.

18.3 Introduction

In deliverable D1.3b (Section 4), we introduced the motivation of Small World Network (SWN), described several SWN models, and introduced our SWN simulator testbed. We also saw that routing performance for the baseline case of a static SWN was rather good since even without much structure, apart from the SWN properties, the number of hops was small.

This deliverable goes deeper into the use of a SWN as the base of a self organizing network which does not have the drawbacks of DHT-based structured overlay networks. We look at two kinds of SWNs. One is where the SWN has a global property on the identifier of nodes (the kind of SWN proposed by Kleinberg). So far our experiments indicate that the routing success rate is very high. This suggests that a SWN may be a feasible as a replacement for structured overlay network in scenarios where there can be malicious nodes or users.

The other kind of SWN does not need any special property on the node identifier. Rather it attempts to reorganize the node identifiers so that it resembles the first SWN. We have identified some attacks which attempt to poison the reorganization process and delete node identifiers. We have also investigated some initial security mechanisms against these attacks based on self-tuning ideas (these happen to fit well with SELFMAN). A paper based on this work is attached in Appendix A.19.

18.4 Small World Network Experiment Testbed

In the simulator described in deliverable D1.3b (Section 4), we are able to test different kinds of SWN models, varying the number of nodes, setting the number of malicious nodes, measure the greediness of the greedy routing (explained later), simulate the node and link failure and plot the result into a graph. See Figure 18.1 which shows the GUI of the simulator.

Three kinds of SWN model can be selected by configuring the simulator. The first SWN model is the Kleinberg model which has base-connections, global knowledge of the positions, and constant number of links. The second model is the Normal (DHT like) model which has $log(n)$ links. The third is the Sandberg model [126] using $6 * log(n)$ links. All the links in the models follow Kleinberg-style power-law distribution [86] to make the graph a small world (i.e. having low diameter and high clustering coefficient).

In order to test how resilient the SWN against malicious nodes, the simulator provides a way to simulate any number of malicious nodes. The behavior of the malicious nodes also can be configured to do certain type of

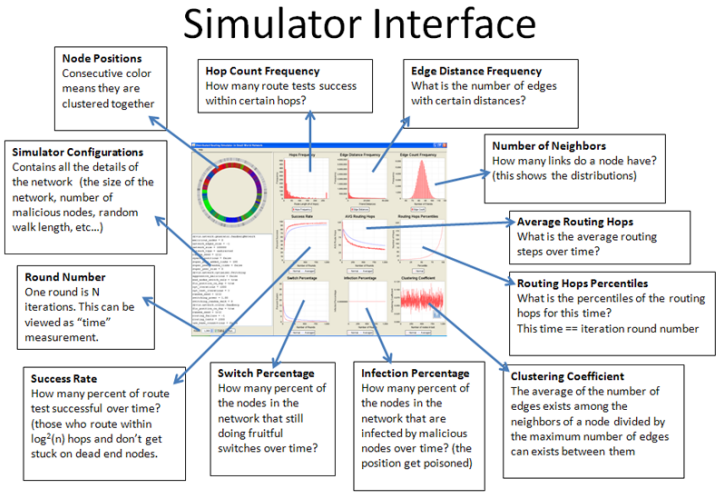


Figure 18.1: Simulator Interface

attacks, for example by doing switching aggressively (active) or defensively (passive).

To measure the robustness of a routing, the greediness of the routing algorithm is tunable. Less greedy means we can make use of routing alternatives (by not picking the best one). By utilising more alternatives, routing can better withstand node failure.

Last but not least, to test the dynaminism of the network, for example churn. The simulator can generate variable number of node or link failures and then freeze the network to be analyzed later by doing 10,000 routing tests. If routing pick a failed link or node the routing length will get longer and the routing is declared as a failure if a certain TTL is exceeded or if it reaches a deadend.

The resulting experiments are visualized as graphs to give immediate feedback. The graphs displaying the "Infection Percentage", "Switch Percentage", "Success Rate", "Average Routing Hops", "Routing Hops Percentiles" are useful in doing the self-protection experiments. For example we need to design the self-protection mechanism such that the resulting network will have high success rate (good routing success rate) with good routing performance (good in average and routing hops percentiles), low infection percentage (low poison), and low switch percentage (low network traffic). These five graphs gives us different insights when designing/tuning the self-protection experiments.

To simulate a very large number of nodes, our simulator has a mode which

can run in parallel on cluster of machines. We make use of the machine cluster at NUS which has about 100 nodes with 200 CPU cores. The simulator can efficiently simulate a large number of nodes (e.g. N = 100,000) together with many different parameters of the SWN within a few to tens of minutes.

18.5 Small World Network as the Network

The motivation for the SWN is to avoid the problems with DHTs regarding the lack of identity, network maintenance including node/link failure, and robustness in routing in terms of alternative paths. We have begun an initial study to examine the question of whether it is feasible to replace a DHT-based structured overlay network with a SWN.

In this section we will measure some aspects of SWN robustness with respect to node/link failure, and routing performance. We assume that the network has the global coordinate position [86], so the routing test will route messages from a particular node to another node with known coordinate in the network (based on the Kleinberg model). An example of coordinate position in a 2-D graph are the cartesian coordinates of a node. Similarly, the coordinate positions in a 1-D graph could be its ring identifier.

The greedy routing algorithm is to route the message to a neighbor that is the closest to the target/destination node coordinate. Note that in a distributed setting, greedy routing only requires local operations (e.g. routing only by knowing the neighbor positions and the target node position).

We already saw in deliverable D1.3b (Section 4) that without node/link failure the routing experiments essentially succeeded all the time (100% success), see Figure 18.2. Note that the number of nodes for all experiments in this section is 100,000.

In Figure 18.2, Kleinberg model indeed can route within $O(log^2(n))$ steps using the greedy routing. However the performance can be improved by increasing the number of links to $log(n)$ as in Normal model and $6 * log(n)$ as in Oskar Sandberg model.

In the presence of node/link failure, the effect on the SWN models can be seen in Figure 18.3. Node failure is equivalent as having all links to the node fail, while in link failure only some of the links of a node fail. Our experiments are performed by freezing the network at a point in time, to do routing tests and then carrying on with network changes.

Figure 18.3 shows our preliminary experimental results on the robustness of a SWN. In the Kleinberg model, the link failure is less damage than node failure. This is because the Kleinberg model only has 2 additional links making it very vulnerable to node/link failure. The success percentage drops

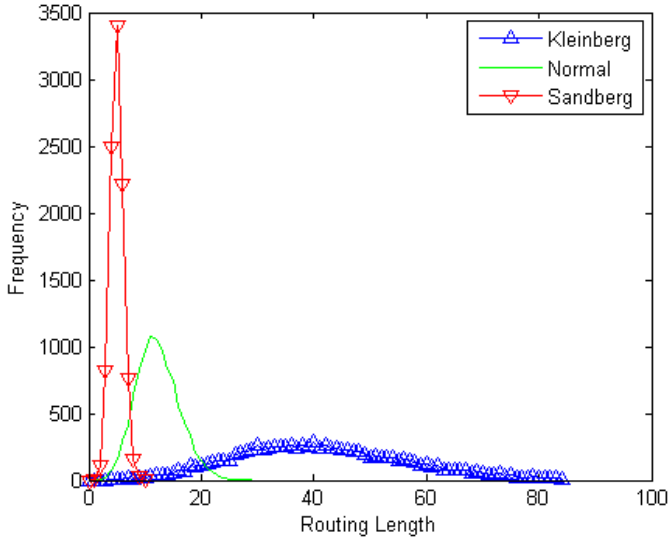


Figure 18.2: Routing Length Distribution

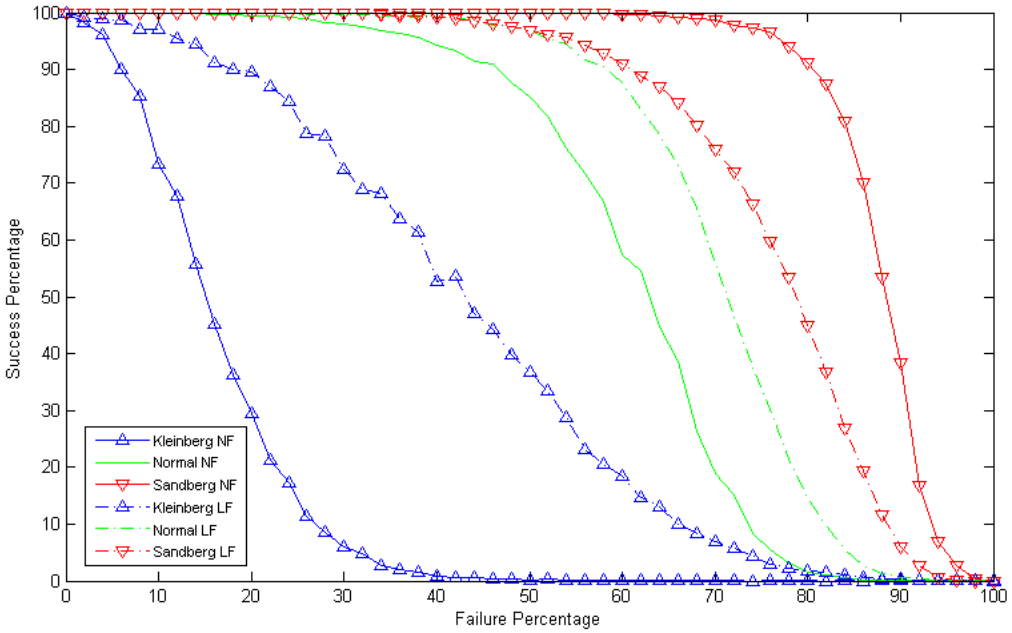


Figure 18.3: Comparisons between 3 models

below 50% with only 15% of node failure or with 40% of link failure. In the Normal model, with $log(N)$ shortcuts instead of 2 constant additional shortcuts, the robustness increases. The success percentage drops below 50% with 60% of node failure or with 70% of link failure. In the Sandberg model, with 6*log(N) shortcuts, the success rate is better again.

To understand the issues of routing robustness, we varied the choice of route from the best to the worst local choice. This shows the effect of route choice on eventual success.

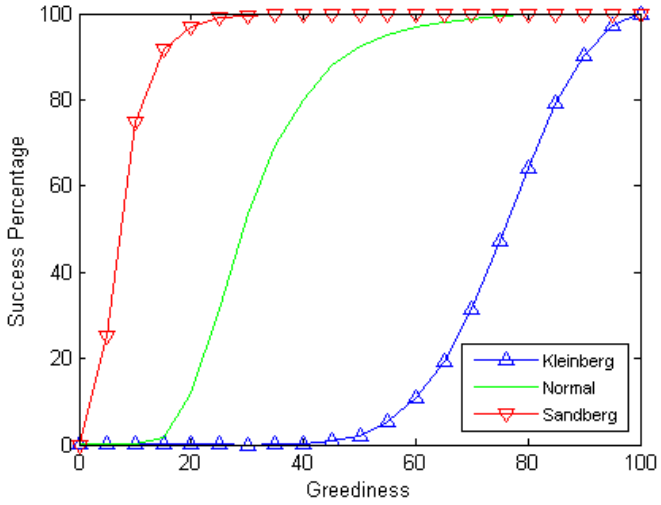


Figure 18.4: Greediness

In Figure 18.4, it seen that in Kleinberg model where the number of edges is very few, the greediness affects a lot on routing success percentage. As the number of edges increases to $log(N)$ and $6*log(n)$ as in Normal model and Sandberg model the success percentage increases a lot. The greediness tells how greedy the routing. For example 100% greedy means the closest neighbor to the target node will be selected. 80% greedy means the closest neighbor to the target node will be selected with 0.8 probability if it is not selected then the next closest neighbor to the target node will be selected with 0.8 probability and so on. If the probability is too low, the neighbor will be selected at random.

Our preliminary results suggest that this version of a SWN seems to be quite promising. Routing works well even in the presences of failure. This can be explained when we look at the robustness experiments since there is quite a lot of leeway in choosing routes before the success rate drops.

18.6 New Security Issues with Small World Networks

The Kleinberg SWN model works under the assumption that the coordinates of the network are given, thus, every node knows where is its coordinate, their neighbors coordinates and the target node coordinate (see Figure 18.5). This property allows greedy routing to work.

In P2P settings, it might be the case that global information such as coordinate either does not exist or is not known. As we will see, this causes a problem for greedy routing.

Sandberg [126] proposed a method to partially recover the node positions (coordinates) by using only local knowledge at each node. Thereafter, greedy routing can be applied, and as we show below, it works almost as well as the original network with coordinates as identifiers. Thus can be thought of as a Self Organizing Network that can reorganize the positions in the network. To give an illustration on how the self-organization works, see Figure 18.5, 18.6, 18.7.

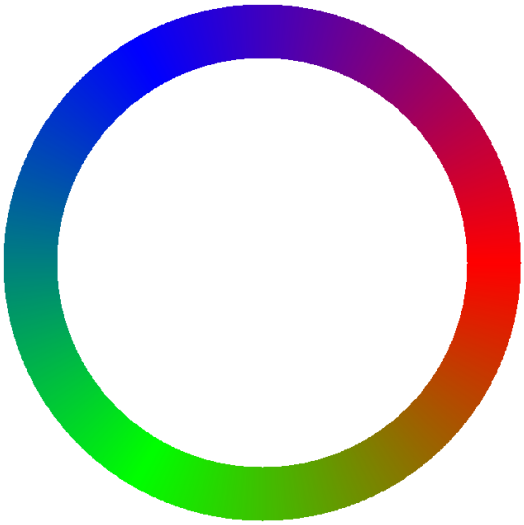


Figure 18.5: Perfect node positions

Figure 18.5 is a graph with perfect coordinates where the coordinates are colored continuously along the ring using red-green-blue hues. The figure depicts 1000 nodes, if the position coordinate space is from 0 to 1000 then the nodes with position ranged between 0 to 333 are colored as reddish, those with position ranged from 334 to 666 are colored as greenish, and those with

position ranged from 667 to 999 are colored as bluish. In the figure, the reddish nodes are on the right side, greenish are on the bottom left, and the bluish are on the top left side of the ring. This coloring scheme makes it easy to see the node position and its coordinates.

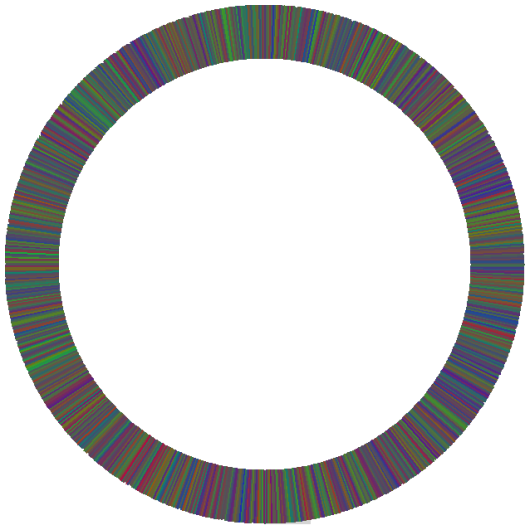


Figure 18.6: Shuffled node positions

Then the coordinates are shuffled yielding a randomized color along the ring as in Figure 18.6. In this graph, the greedy routing doesn't work since the coordinates are random. The coordinates cannot be used to guide the greedy routing to the target node hence the routing performance of this graph is very poor. The situation is similar to an initial peer-to-peer system that doesn't have any coordinates and then begin to generate their own unique coordinates (which are essentially random coordinates just like the shuffled ring above). The network then is expected to recover the coordinates back for the greedy routing to work well.

Sandberg [126] shows that the network can be restored as in Figure 18.7. We can see in the picture that the ring colors are not the same with the original perfect ring in Figure 18.5, in fact without having a global information it is very difficult if not impossible to get back to the perfect state. Even though the restored ring is imperfect, nevertheless, greedy routing works very well on the restored network. Routing performance on the restored ring is close to the perfect ring performance.

The recovery (self-organizing) algorithm was not designed to defend against attacks. In this deliverable, we have begun an initial investigation on the attacks against the self-organizing recovery algorithm and protection mech-

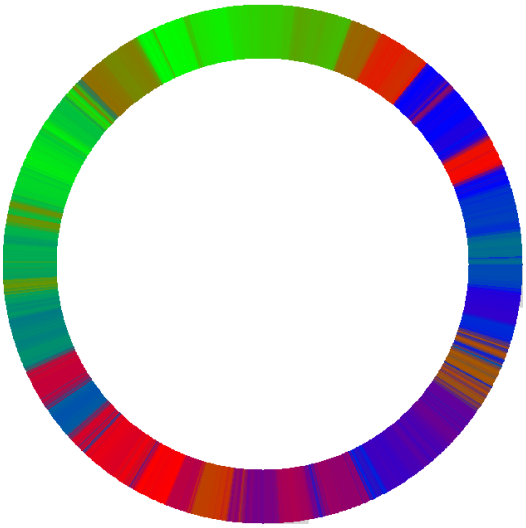


Figure 18.7: Restored node positions

anisms against attack on the Sandberg reorganization mechanism. More details can be found in the attached paper in Appendix A.19.

The self-organizing algorithm works by first assigning the node a random position. Then on each iteration each node will try to do a random walk of certain length to another node and attempt to switch position with that node if it's beneficial. Basically the switch is determined by the switching probability which is calculated by having the product of edges before switch divided by the product of edges after switch. If the resulting probability is larger than 1, then they always do switch (see Figure 18.8).

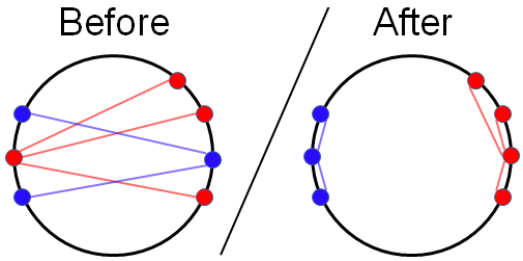


Figure 18.8: Switching Probability

This self-organizing algorithm (without any protection) is vulnerable to even with a small number of malicious nodes. This can happen because the malicious nodes can fake positions, drop positions, create similar positions,

etc. Basically these attacks try to poison the positions of nodes as the network is doing self-reorganisation of the positions. Even with small number of malicious nodes, a small amount of poisoning can eventually propagate to infect the entire network. Hence, self-protection mechanisms are necessary.

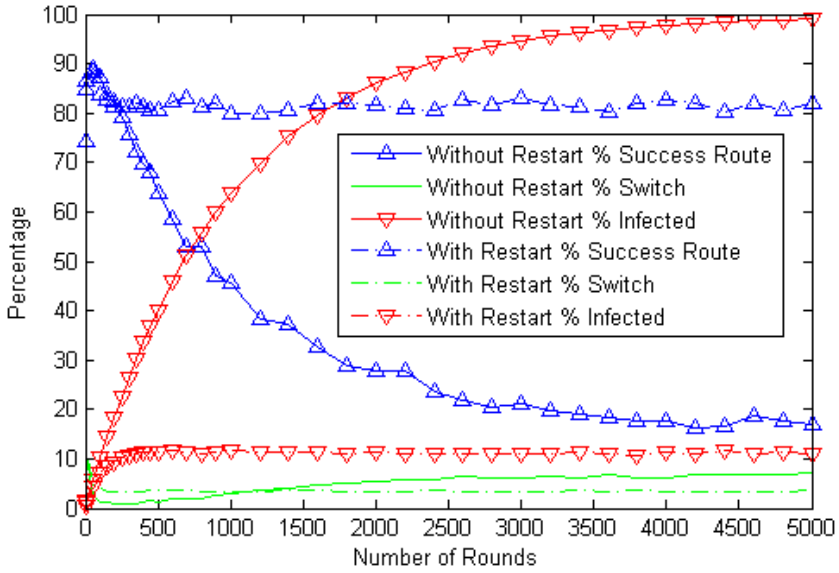


Figure 18.9: Partial Restart Strategy

We proposed a partial restart strategy to minimize the effect of poison. Each node will generate a new position with probability r. Using this strategy, the position distribution in the network will remain uniformly distributed. Since the convergence of the recovery algorithm is faster than the restart strategy, the network is able to maintain good routing performance even though the positions are being randomized. Our preliminary experiments showed that simple decentralized security mechanisms helps in minimizing the effect of the attack. Figure 18.9 shows the effect of having self-protection by partial restart strategy. It keeps the infection below 10% and the routing success above 80%.

Our preliminary experiments showed that simple decentralized security mechanisms helps in minimizing the effect of the attack. It is interesting that this both the self-organizing algorithm and the self-protection mechanisms here are also instances of self-tuning algorithms.

18.7 Papers

The paper which describes the work here is as follows:

Felix Halim, Yongzheng Wu and Roland H.C. Yap, "Security Issues in Small World Network Routing". It has been accepted by the *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*. It is attached in Appendix A.19.

Chapter 19

D5.2a: Application design specifications

19.1 Executive Summary

The purpose of the WP5 is:

1. to provide use cases and requirements in different applicative contexts (applications),
2. so as to help the Selfman project as a whole to choose which application(s) will be demonstrated,
3. and then to perform evaluations of the Selfman demonstrators and more globally the Selfman technologies and overall approach (autonomics based on components and overlays).

Four applications were considered in WP5[1]. The first one proposed by France Telecom concerns M2M systems. The second one proposed by ZIB concerns a distributed database system. Two additional applications were investigated as replacements for the one that should have been proposed for the partner E-plus which left the Selfman project in its first year. The first one, proposed by the Peerialism company (previously named PeerTV and then Stakk) (contact established by KTH(P2)), concerns P2P video streaming (P2P TV). The second one, proposed by the Bull company (contact established by FT R&D(P4)), concerns a J2EE application server. After investigation, applications from Bull and FT R&D(P4)finally appeared as not

[1]More elements on the history/process of Selfman WP5 in the 1st year of the project is detailed in the corresponding deliverable documents of the first year.

suitable as Selfman demonstrators and were then discarded (more details in next of this deliverable concerning the M2M application by FT R&D(P4)).

This deliverable provides design specifications for the wiki distributed database application and P2P TV application.

19.2 Contractors Contributing to the Deliverable

ZIB(P5), Peerialism and FT R&D(P4)have contributed to this deliverable.

ZIB(P5) has contributed on the design specification and implementation of the wiki distributed database application. This work is based on the development in WP3 on transactions in structured overlay networks and user requrements specified in WP5 (D5.1).

Peerialism has contributed on the design specification and implementation of the P2P TV application. This work is based user requrements specified in WP5 (D5.1).

FT R&D(P4) has contributed on the edition of this deliverable. FT R&D(P4)provided very detailed user requirements on a multi-service M2M application in D5.1a. This application was finally not implemented by FT R&D(P4)in the context of Selfman. It is perhaps worth giving here the main reasons why:

- This was not what FT proposed to do in Selfman to start with (cf. DOW: FT was supposed to provide component technology, user requirements and evaluation technology, not applications).
- FT did its best on user requirements for M2M application (not to mention additional work in automn 2007 because the deliverable was not accepted) but did not have the manpower to develop an M2M application (defining user requirements for an application is not the same activity as actually implementing this application).
- As discussed in the conclusion of D5.1, Selfman technologies might be interesting for M2M in the long term but M2M, in FT R&D(P4)settings at least, is not mature enough to make use of largely distributed architectures (M2M applications today send direclty sensor data to one centralized J2EE server
- - they are years away from using overlays).
- FT R&D(P4)would not see how the implement an M2M application with the current Selfman technologies and (lack of) architectural vision (several component models, several overlays).

19.3 Results

19.4 Wiki Application Design Specifications

The distributed wikipedia is a reimplementation of the wikipedia using the principles and techniques developed within the Selfman project. Wikipedia is particularly interesting for us, because it is the only large scale web application which gives access to its source code and data. The code is written in PHP and available on their web site. More important for us, they provide complete database dumps.

For the distributed wikipedia, we are using the replicated storage service (see D3.XX) as the backend and built a layer on top which maps the wikipedia operations on the key-value store. Fig. 19.1 shows the general architecture. At the top we have the transactional key-value store using Chord#. The presentation layer consists of a couple of webservers which handles the rendering of wiki text to HTML and provides forms for changing the contents. A load-balancer will distribute the user requests over all the webservers.

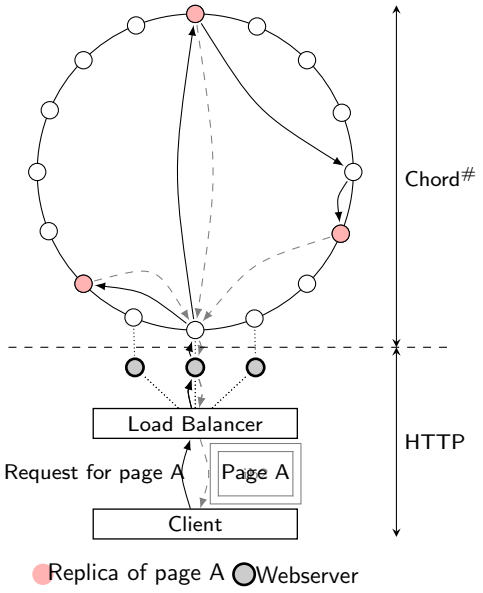


Figure 19.1: Distributed Wikipedia on a transactional data store based on Chord#.

A more detailed description as well as performance results can be found in App. A.20.

We participated in the first IEEE Scale Challenge with the distributed

wikipedia and won the first prize. There we showed a deployment of the Bavarian wikipedia over several dozens of nodes in Europe. A second deployment was running on a cluster in Berlin with a simplified English version.

19.5 P2P TV Application Design Specifications

In this section we present our experience with using Kompics[9], a software framework for building distributed applications developed at KTH. We'll first give a short introduction about the framework itself, then we will describe our system and expose the reasons why we decided to develop part of our application in Kompics. Finally, we will provide a description of the software's design and a preliminary evaluation of it.

19.5.1 Peerialism's system

Peerialism's product is a content distribution platform which performs audio and video streaming directly to the customer's home computer. It does that by building an ad-hoc overlay network between all hosts requesting a certain stream. This network is organized in such a way that the load of the content distribution is shared among all the participating peers.

The main entities in our system are:

- The Clients, which are the peers where Peerialism's client application has been installed, i.e. the customers home computers. The installed application requests audio and video streams according to the input received from the customer. It then receives streams from other peers, delivers them to the local media player and streams them once more to other customers.
- The Source. It represents a host which has all data of a certain stream. The Source itself is a Peer. A Peer becomes a source for a specific stream when it has received all the data of that same stream.
- The Tracker. It is the central coordinator of the system. It is not part of the overlay network but it organizes it. It receives requests from the clients, forwards them to an optimization engine and issues directions to the peers once the request has been satisfied.

- The Optimization Engine. It receives the forwarded requests from the tracker and performs decisions according to the overall status of the network. In addition, it periodically redefines the structure of the overlay network to optimize the flow of streams to normalize the load of the delivery among the peers.

A typical example of the steps needed to deliver a stream are described as follows: the customer requests a specific audio or video stream to the client application running on its home computer, the application translates the request into a message to the tracker. After receiving the message, the tracker forwards it to the optimization engine. The engine builds a map of the overlay network, which represents the global status of the delivery of the stream, and makes a decision on which peers should provide the content to the requesting peer. Once the decision has been made, the tracker notifies the peers involved in the operation. The requesting peer will then start to receive the content from other peers. The providers might be both sources or normal peers which have already received parts of the stream. Once the delivery has been completed, the peer notifies the Tracker and becomes a Source for that stream.

19.5.2 Introduction to Kompics

Kompics is a software framework which allows for programming, configuring, and executing distributed protocols as software components. Kompics components interact using data-carrying events and can be composed into complex architectures. In particular, they can be organized in composite and shared components, as well as in hierarchies. The components are safely reconfigurable at run-time so that the architecture of the system can be dynamically modified and components replaced. Software faults which might occur during execution are safely isolated and handled by supervisor components.

The component programming style used in the Kompics framework allows for reusability, since components can be reused in other applications, and it enables flexible management of computational resources, as it is possible to define various policies to assign threads to components for their execution.

Kompics defines an execution model where components don't share any state and can communicate exclusively using events through predefined channels. The events are then executed atomically with respect to the component instance. This model allows for parallel execution of components and takes advantage of multi-core hardware architectures. Furthermore, it eases the implementation of distributed protocols by offloading the programmer from

the difficulty of programming concurrency with threads.

The Kompics framework also provides a number of components which can be used out-of-the-box to develop any distributed application. These components implement useful abstractions such as reliable and lossy network communication and failure detection.

19.5.3 The Tracker application

As mentioned earlier, the Tracker is the most important entity in the system, since it interacts and coordinates peers in the overlay network. Therefore, the application implementing its behavior must be reliable, to avoid internal software failures, and resilient to failures, both protocol and network ones, that might happen in the interaction with the peers. Furthermore, according to the specification of our system, the Tracker application should be able to handle a number of clients on the order of magnitude of hundreds of thousands. Scalability is then a major requirement.

The Tracker application has been currently developed in Java using threads to exploit multi-core architectures and improve performance. It also makes use of the MINA framework[52] to abstract the network layer and provide Non-blocking I/O.

In general, the Tracker greatly differs from all other parts of the system, such as for instance the Client application. The Tracker is in fact a much smaller and less complex application than the Client. This because the latter is meant to fulfill more complicated tasks, such as: receipt and transmission of streams, delivery to the media player, bandwidth measurement and NAT Traversal. However, the role of the Tracker is more critical compared to the one of the Client, since all peers in the network depend on it. Even a temporary Tracker failure would cause great harm, as all peers in the system are unable to request streams. Instead, in case of a Client failure, only the peers which are receiving a stream from that specific host will be affected.

For the aforementioned reasons, software implementing the Tracker behavior has been tested with both unit-tests[80] and with our simulator. However, dealing with Threads and shared structures has made the design of tracker cumbersome and error-prone. Moreover, it has been difficult to obtain scalability and cope with the amount of peers that the Tracker should be able to handle by specification.

Considered the aforementioned problems, we decided to explore the possibility of using Kompics to re-implement our Tracker application. In particular we were attracted by the following reasons:

- Clean design, which could be obtained by organizing our software in

components.

- Reusability, components might be reused in future products.
- No shared state, components logically encapsule the state that they need for execution and can communicate only using events.
- Explicit concurrency, components are concurrency units which can be executed in parallel without need of synchronization.

.

19.5.4 Porting and Design

In general, the biggest challenge encountered in the process of porting the tracker application to Kompics was to split our software into components. In fact, it has been quite difficult to identify and isolate the state that should be directly and exclusively modified by the component enclosing it. Once we did that, we collected the parts of the pre-existing code that were to modify that same state. We packed those parts of code into event handlers and created the events which would trigger them. Finally, we created the components and defined the interconnections between them by creating channels and specifying which events they would carry. The resulting design can be observed in Figure 19.2.

The network component, shown at the top of the figure, constitutes the Kompics abstraction for the network layer. The network component provides network communication both using the TCP and UDP protocols. Its interface is quite simple, components can trigger events which extend an abstract Message event. The network component subscribes to those events and sends them to the destination peers. In turn, components which are interested in receiving messages subscribe to those same events on the other side.

The MemberManager Component instead, shown on the left in Figure 19.2, keeps track of all information regarding the peers. When started, a peer registers to the tracker by sending its credentials, such as username and password of the customer, and its characteristics, namely its download and upload bandwidth capacity, the kind of NAT it is behind and its public and private IPs. It then reports which data it has available locally. In fact, the peer might have stored some content as a result of a previous streaming session. That data might be reused later to serve other peers. The MemberManager Component is not only entitled with the task of storing all this information coming initially from the peers but it also receives periodic notifications from

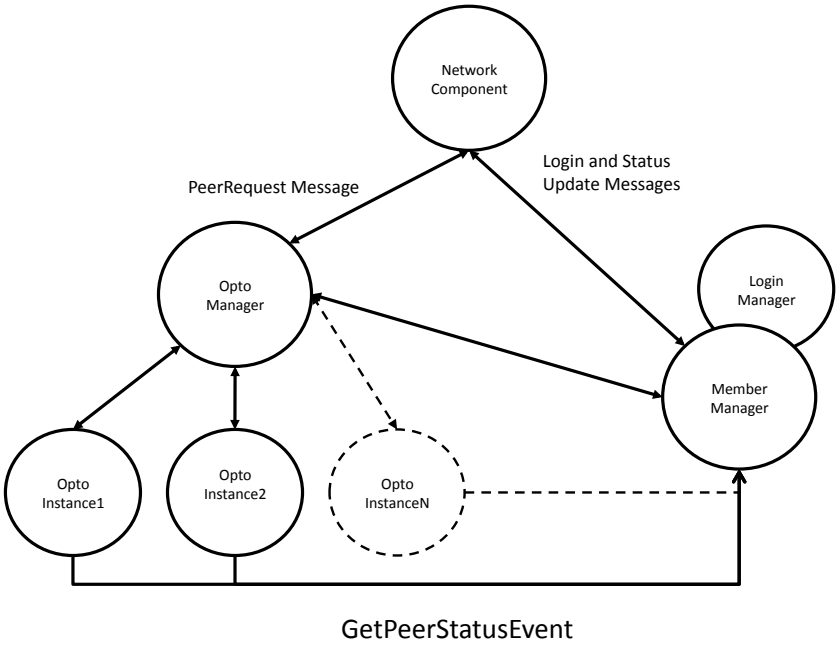


Figure 19.2: Tracker Kompics Design

them. The notifications contains the status of the delivery of the streams requested by each peer. Such information consists of which pieces of the stream have been received and how many of them have been delivered to the media player.

In the right side of Figure 19.2 we can find the OptoManager component and its children components, the OptoInstances. The Tracker application can handle a number of different streams. For each of them, it spawns an OptoInstance component. The latter contains an instance of the Optimization Engine introduced in Section 19.5.1. The engine's instance handles requests concerning a single stream. The OptoManager is instead a supervisor component. It creates and destroys OptoInstance components as new streams are added or removed from the system. The OptoManager also acts as a hub for the requests coming from the peers: it redirects them to the corresponding OptoInstance component.

When an OptoInstance component receives a request forwarded by the OptoManager, it triggers a number of events requesting to the MemberManager information about the peers involved in the delivery of the stream. Once the corresponding response events have been received, the embedded Optimization Engine decides which one of the peers should provide content to the requesting host.

In addition, every OptoInstance is periodically requested to gather information about all peers involved in the delivery of its responsible stream. It does that, as mentioned earlier, by issuing events to the MemberManager. The response data is then passed to the embedded Optimization Engine. The latter builds a map representing the overlay network and the status of its members. Once the operation is concluded, it tries to reorganize the map such that the load of the delivery is shared among all the involved peers. After a decision has been made, notification messages are sent to the peers by triggering the corresponding Message events.

This design explained above has two main advantages:

1. **Isolation of Peer's state**. The MemberManager component is the only component containing information about the status of peers. Changes carried by incoming events are committed atomically with respect to the MemberManager component, as mentioned in 19.5.2. The OptoInstance instead does not retain any information about the status of the peers. It just possesses a list containing the ids of the peers which participate in the delivery of the stream. It then needs to request their status to the MemberManager component before making decisions. After every decision, the data concerning the peers is discarded as it might have changed during the decision process.

   With this design, the OptoInstances and the MemberManager components can be executed concurrently without need of synchronization. Consequently, information about the peers can be stored immediately even if the Optimization Engine is running and the Optimizer does not need to acquire any lock for accessing the peers' status, as it happened in the previous implementation of the Tracker.

2. **Concurrent execution of Optimization Engines**. The design of the Tracker application has been appositely studied to allow parallel execution of OptoInstance components, and consequently of instances of the Optimization Engine. This because the optimization process accounts for most of the load of the Tracker application. In fact, every time a decision must be made, whether triggered by a periodic reorganization of the stream or a direct request from a peer, the Tracker application has to perform a computationally intensive task. With this design, OptoInstance components are independent from one another and from any other component in the system. Consequently, they can be executed concurrently on a multi-core machine. Furthermore, OptoInstance components do not keep threads busy while in idle. In Kompics, event handlers cannot make blocking calls and the executing

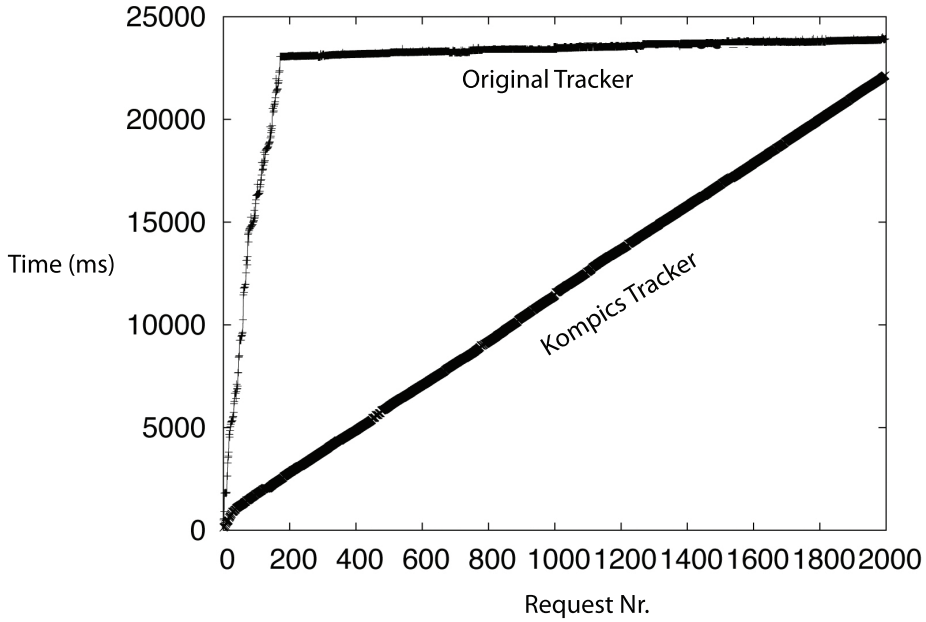


Figure 19.3: Kompics tracker preliminary evaluation, 2000 requests

thread is released after the event-handler has terminated its execution.

19.5.5 Preliminary Evaluation

After having reimplemented the Tracker application following the aforementioned design, we compared it with the original Tracker application. For doing that, we used MINA's VMPipe library. The library provides a fake network layer which can be easily integrated in applications designed for MINA, as the original Tracker application and its Kompics version. We made this choice to be able to test the raw performance of the Tracker, without the burden of connection and message handling. We then implemented a bogus version of the Client application which simply authenticates to the Tracker, makes a request for content and awaits a response. Every Client measures how long it takes to receive a response from the moment it triggered a request.

In our tests, we create a number of those bogus Clients and we make sure that requests are issued sequentially as fast as possible. The Tracker then receives the requests, forwards them to the Optimization Engine and replies back to the Client. Content Requests are processed in the order they have been received. The test starts when the Tracker receives the first request.

Figure 19.3 shows the results of an experiment with 2000 bogus Clients which trigger as many content requests. On the X-axis is displayed the number of the request and, on the corresponding value of the Y-axis, the time that it took to satisfy the same request. As we can see from the picture, the Kompics version of the Tracker always outperforms the original version. This is due to the concurrent nature of Kompics which do not rely on synchronization of threads. We believe that the latter is the cause of the original tracker having an almost constant response time, since the application gets flooded with requests and the access to shared structures becomes therefore sequential.

19.5.6 Conclusion and Future Work

We presented our experience in porting Peerialism's Tracker application to the Kompics framework developed at KTH. We outlined the requirements of the application and we detailed the design choices that we made for meeting those requirements. We then presented the results of a preliminary evaluation of the software.

We now would like to perform a more extensive evaluation of the Kompics Tracker software before integrating it in our product. In particular, we would like to test its resilience to software failures and its stability. We then would like to test the application's behavior in our remote test environment, with real Clients, to verify that the performance are the same as showed in the preliminary tests. If expectations are met, we intend to port also our Client application to Kompics to exploit the clear advantages in clean design, reusability and explicit concurrency that the framework provides.

Chapter 20

D6.1c: Second-year project workshop

In the second year we are organizing the Workshop on Decentralized Self Management for Grids, P2P, and User Communities:

`http://www.ist-selfman.org/wiki/index.php/SelfmanWorkshop`

which will be held in conjunction with the Second International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008):

`http://polaris.ing.unimo.it/saso2008`

on Oct. 20-24, 2008. We advertised the workshop widely on mailing lists and internal lists. The Workshop Call for Papers is given in Section 20.1. The submission deadline is July 11, 2008.

The workshop is organized by SELFMAN together with the Grid4All project, with corporate sponsorship by France Télécom Research and Development. The workshop organizing committee consists of Peter Van Roy (SELFMAN), Marc Shapiro (Grid4All), and Seif Haridi (SELFMAN and Grid4All).

20.1 Call for Papers: Decentralized Self Management for Grids, P2P, and User Communities

The Internet is a fantastic tool for information and resource sharing. User communities such as families, friends, schools, clubs, etc., can pool their resources and their knowledge: hardware, computation time, file space, photos, data, annotations, pointers, opinions, etc.

However, infrastructures and tools for supporting such activities are still relatively primitive. Existing P2P networks enable world-wide file sharing but are limited to read-only data and provide no security or confidentiality guarantees. Grids support closed-membership virtual organizations (VOs), but their management remains largely manual. Web 2.0 social networks, blogs, and wikis remain centralized and have limited functionality.

This workshop examines issues of decentralized self management as they relate to these areas. The workshop is co-located with SASO 2008, the Second International Conference on Self-Adaptive and Self-Organizing Systems (http://polaris.ing.unimo.it/saso2008/). The workshop covers the application of:

- Peer-to-peer techniques such as structured overlay networks
- Self-adaptive techniques such as feedback loop architectures
- Agoric systems, collective intelligence, and game theory
- Decentralized distributed algorithms
- Autonomic networking techniques

to:

- Virtual organizations
- Collaborative and social applications
- Computer-supported cooperative work, collaborative editing, code management tools, and co-operative engineering
- Data replication, distributed file systems or distributed databases
- Security and confidentiality in distributed systems

20.1.1 Submission of position paper or technical paper (required for attendance)

To attend the workshop, you must submit a position paper of no more than 5 pages to the workshop organizers. Technical papers may be submitted as well (of any reasonable length). All accepted submissions will be published in an IEEE postproceedings. Submissions should be sent to the following address (preferably by email in pdf format):

Peter Van Roy
Dept. of Computing Science and Engineering
Place Sainte Barbe, 2
Universitécatholique de Louvain
B-1348 Louvain-la-Neuve
Belgium
Email: peter.vanroy@uclouvain.be

Each submission will be reviewed by at least three reviewers. The review will focus not only on the paper's quality but also on its novelty and ability to engender fruitful discussions. All authors of accepted position papers are invited to attend the workshop. Note that workshop attendees must register both to the conference and the workshop. Workshop PC members are also encouraged to submit position papers and these papers will be reviewed to the same standards as outside submissions.

This workshop is sponsored by the European projects Grid4All:

(http://www.grid4all.eu)

and SelfMan:

(http://www.ist-selfman.org/)

and with corporate sponsorship from France Télécom Research and Development.

20.1.2 Organizing committee

Peter Van Roy, Université catholique de Louvain, Belgium
Marc Shapiro, INRIA & LIP6, Paris, France
Seif Haridi, SICS & KTH, Stockholm, Sweden

20.1.3 Program committee

Gustavo Alonso, ETH Zurich, Switzerland
Seif Haridi, SICS & KTH, Stockholm, Sweden
Bernardo Huberman, HP Labs, Palo Alto, USA
Adriana Iamnitchi, University of South Florida, USA
Mark Miller, Google Research, USA
Pascal Molli, LORIA, Nancy, France
Luc Onana Alima, UMH, Mons, Belgium
Nuno Preguiça, Universidade Nova de Lisboa, Portugal

Alexander Reinefeld, Zuse Institut Berlin, Germany
Marc Shapiro, INRIA & LIP6, Paris, France
Peter Van Roy, Université catholique de Louvain, Belgium
Hakim Weatherspoon, Cornell University, Ithaca NY, USA

Chapter 21

D6.5b: Second progress and assessment report with lessons learned

21.1 Executive summary

We can see that the SELFMAN project runs on three different levels that interact and cross-fertilize each other: a vision level, an implementation level, and an application level. In the vision level, we set the long-distance goals of the project: how to build self-managing applications using concurrent components, interacting feedback loops, and reversible phase transitions. In the implementation level, we build a complete application infrastructure with a transaction protocol running on top of a structured overlay network, implemented with a component model. In the application level, we investigated four application scenarios that touch on four different parts of the design space of self-managing applications. In this report, I recapitulate the progress made in each level and how the levels interact with each other and direct each other. The implementation level has strongly interacted both with the vision level and the application level, and it also influences both.

21.2 Contractors contributing to the deliverable

All contractors contributed ideas to this deliverable.

UCL UCL (Peter Van Roy) wrote the present report.

21.3 Results

After the second year of the project, we can judge the effectiveness of the different levels that are guiding the SELFMAN project. There are three levels: the vision level, the implementation level, and the application level. We recapitulate these levels, their results, and how they influence each other.

In essence, the implementation level gives the practical results of the project and it implements both the vision and the applications. The vision level gives a future view of how self-managing applications should be organized and the implementation level has realized part of this vision in a working application infrastructure. The application level gives a practical view of what services the self-managing applications really need and the implementation level has realized these services.

21.3.1 Vision level

The first level is a high-level "vision" thread in the project, fed by four successive papers. We can see how the vision has evolved during the project:

- Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components, CoreGRID technical report TR-0018, Dec. 2006 [147]. This paper gives the initial vision of the project at its start: extending structured overlay networks into full-fledged self-managing systems by using components to organize the extension.

- Self Management and the Future of Software Design, FACS 2006, Sept. 2006 [145]. This paper explains the importance of interacting feedback loops and gives many examples of feedback loop structures in successful biological and software systems.

- Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project, FMCO 2007, 2008 (to appear) [148] (this paper is reproduced as Chapter 2 of the present document). This paper gives ideas how to design and analyze systems with interacting feedback loops and explains how collective intelligence allows to manage users with conflicting goals. It also presents the SELFMAN architecture of a structured overlay network with a replicated storage and transactional protocol, as an application architecture. Finally, it explains how to handle network partitions using the merge algorithm.

- Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions, BCS 2008 (to appear) [146] (see Ap-

pendix A.1). This paper pushes the concept of interacting feedback loops to its logical conclusion. Inspired by physical systems, whose behavior (both non-critical and critical) can be explained with interacting feedback loops, the paper motivates a new architecture for robust software based on reversible phase transitions. Most existing fault-tolerant software does a phase transition when highly stressed, but does not revert to its initial condition when the stress is relieved. The paper argues that this is incorrect and that the phase transition should be reversible. The merge algorithm developed in SELFMAN is the first example of such a reversible system.

These papers present an ultimate vision that is subsequently realized in the rest of the project. We admit that the realization is not complete: we do not yet understand how to program with interacting feedback loops and the notion of reversible phase transition in software is only partially understood. Yet, we have made definite progress. E.g., the different algorithms in the project (relaxed ring, Paxos uniform consensus, etc.) can be considered as feedback loop structures. The merge algorithm gives the first overlay network that actually does reversible phase transitions (previous "folklore wisdom" in the overlay research community did not consider merge to be practical). Furthermore, the algorithm has been simulated extensively and all indications are that it is practical. We are in the process of implementing it over DKS and P2PS. We conclude that the vision level is guiding the project well at this point.

21.3.2 Implementation level

The second level is the building of robust structured overlay networks. In the second year of the project, we have arrived at overlay networks that are practical in applications. To do this, we had to modify them in several ways:

- Modifications of the overlay's self-organization algorithms to reduce the probability of lookup inconsistency (e.g., relaxed ring). At UCL, the relaxed ring has now been completely simulated and implemented in P2PS and it is well understood. Basically, its join and leave algorithms are completely asynchronous and involve only two nodes at a time. This eliminated lookup inconsistency in the case of joins and leaves only and reduces its probability in the case of failures (true failures or false suspicions). At KTH, lookup consistency has been studied in the case of DKS and with some small changes in the self-organization algorithms it can also be greatly reduced. KTH notes that the network

Use Case	Self-* Properties	Components	Overlay Networks	Transactions
Machine to Machine	++	++	+	+
Distributed Wiki	++	+	++	++
P2P Media Streaming	++	+	++	
J2EE Application Server	++	++		+

Table 21.1: Self-managing application requirements

partition (merge) algorithm is essential: at some point when failures happen, the network is actually partitioned.

- Reimplementation of overlay networks using component models, to allow self-configuration. Since the component models themselves, in particular Kompics, were only implemented in the second year (the design started in the first year), the reimplementation of the overlay networks is only partially complete by the end of the second year. The feedback loop needed for self configuration has been implemented by INRIA in Oz with two tools for dynamic component-based systems, FructOz and LactOz, which provide for distributed deployment (actuating agent) and navigation and monitoring (monitoring agent), both essential parts of a self-configuring system.

- Design and implementation of a transaction algorithm, with a replicated storage algorithm, on top of the structured overlay networks. By the end of the second year, the design is complete and several implementations are on the way. ZIB has made significant progress: they have successfully implemented the algorithm in Erlang and written a Distributed Wiki on top. KTH is implementing the algorithm on top of DKS and UCL is implementing it on top of P2PS. ZIB has a head start because of the choice of Erlang. KTH had to spend time designing and implementing its component model, Kompics, and UCL had to spend time for the Mozart 1.4.0 release with its new advanced distribution model.

21.3.3 Application level

The third level is the choice and study of the self-managing applications we would build. In the first year, we decided to explore the design space by choosing four application scenarios (see Deliverable D5.1):

- Machine-to-Machine (M2M) messaging application (FT). This application was inspired by an existing fielded application in France Télécom.
- Distributed database and Wiki (ZIB). This application is inspired by existing work at ZIB on distributed databases.
- Media streaming application (Peerialism, formerly called Stakk). This application is the initial proposal for a produce by the Peerialism company.
- J2EE application server (proposed by Bull). This runs on a cluster and hosts applications in self-managing way.

These applications cover quite different points in the design space of distributed self-managing applications. We summarize this in table 21.1. In the first year, we decided to drop the J2EE Application Server because it is not an Internet application (it does not need overlay networks). In the second year, we made a further choice to drop the M2M application because of lack of resources. It has less need of overlay than the others and FT does not have the resources to develop it further. We therefore keep two applications: the Distributed Wiki and the P2P Media Streaming application. The Distributed Wiki has been implemented in the second year and it won a prize in a scalable computing competition. The Media Streaming application will be a product of Peerialism. In the third year we will evaluate these applications and improve the self-* abilities depending on the results of the evaluations.

Appendix A

Publications

A.1 Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions

Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions

Peter Van Roy
Dept. of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

Programs are fragile for many reasons, including software errors, partial failures, and network problems. One way to make software more robust is to design it from the start as a set of interacting feedback loops. Studying and using feedback loops is an old idea that dates back at least to Norbert Wiener's work on Cybernetics. Up to now almost all work in this area has focused on how to optimize single feedback loops. We show that it is important to design software with multiple interacting feedback loops. We present examples taken from both biology and software to substantiate this. We are realizing these ideas in the SELFMAN project: extending structured overlay networks (a generalization of peer-to-peer networks) for large-scale distributed applications. Structured overlay networks are a good example of systems designed from the start with interacting feedback loops. Using ideas from physics, we postulate that these systems can potentially handle extremely hostile environments. If the system is properly designed, it will perform a reversible phase transition when the failure rate increases beyond a critical point. The structured overlay network will make a transition from a single connected ring to a set of disjoint rings and back again when the failure rate decreases. There is a complete research agenda based on the use of reversible phase transitions for building robust systems. In our current work we are exploring how to expose phase transitions to the application so that it can continue to provide a service. For validation we are building three realistic applications taken from industrial case studies, using a distributed transaction layer built on top of the overlay.

Keywords: software development, self management, feedback, distributed computing, distributed transaction, network partition, Internet, phase transition

1. INTRODUCTION

How can we build software systems that are not fragile? For example, we can exploit concurrency to build systems whose parts are mostly independent. Keeping parts as independent as possible is a necessary first step. But it is not sufficient: as systems become larger, their inherent fragility becomes more and more apparent. Software errors and partial failures become common, even frequent occurrences. Both of these problems can be made less severe by rigorous system design, but for fundamental reasons the problems will always remain. They must be addressed. One way to address them is to build systems as multiple interacting feedback loops. Each feedback loop continuously observes and corrects part of the system. As much as possible of the system should run inside feedback loops, to gain this robustness.

Building a system with feedback loops puts conditions on how it must be programmed. We find that message passing is a satisfactory model: the system is a set of concurrent component instances that communicate through asynchronous messages. Component instances may have internal state but there is no global shared state. Failures are detected at the component level. Using this model lets us reason about the feedback behavior. Similar models have been used by E for

building secure distributed systems [18] and by Erlang for building reliable telecommunications systems [1]. More reasons for justifying this model are given in [24]. For the rest of this paper, we will use this model.

Now that we can program systems with feedback loops, the next question is *how* should these systems be organized. A first rule is that systems should be organized as multiple interacting feedback loops. We find that this gives the simplest structure and makes it easier to reason about the system (see Sections 2 and 3). Single feedback loops can be analyzed using techniques specific to their operation; for example Hellerstein *et al* [10] gives a thorough course on how to use control theory to design and analyze systems with single feedback loops. The problem with systems consisting of multiple feedback loops is their global behavior: how can we understand it, predict it, and design for a desired behavior? We need to understand the issues before we can do a theoretical analysis or a simulation.

In the SELFMAN project [20], we are tackling the problem by starting from an area where there is already some understanding: structured overlay networks (SONs). These networks are an outgrowth of peer-to-peer systems. They provide two basic operations, communication and storage, in a scalable and guaranteed way over a large set of peer nodes (see Section 4). By giving the network a particular topology and by managing this topology well, the SON shows self-organizing properties: it can survive node failures, node leaves, and node joins while maintaining its specification. By using concepts and techniques taken from theoretical physics, we are able to understand in a deep way how SONs work and we can begin to understand how to design them to build robust systems. The concepts of feedback loop and phase transition play an important role in this understanding.

This paper is structured as follows:

- Section 2 defines what we mean by a feedback loop, explains how feedback loops can interact, and motivates why feedback loops are essential parts of any system. We briefly present the mean field approximation of physics and show how it uses feedback to explain the stability of ordinary matter.
- Section 3 gives two nontrivial examples of successful systems that consist of multiple interacting feedback loops: the human respiratory system and the Transmission Control Protocol.
- Section 4 summarizes our own work in this area. We are building a self-management architecture based on a structured overlay network. We conjecture that when designed to support reversible phase transitions, a SON can survive in extremely hostile environments. We support this conjecture by analytical work [14], system design [21], and by analogy from physics [15]. We are currently setting up an experimental framework to explore this conjecture. We target three large-scale distributed applications, built using a transactional service on top of a structured overlay network.

Section 5 concludes by recapitulating how feedback loops can overcome software fragility and why all software should be designed with feedback loops. An important lesson is that systems should be constructed so that they can do reversible phase transitions. Most existing fault-tolerant systems are *not* designed with this goal in mind, so they are broken in a fundamental sense. We explain what this means for structured overlay networks and we show how we have fixed them. We then explain what remains to be done: there is a complete research agenda on how to build robust systems based on the principle of reverse phase transitions.

2. FEEDBACK LOOPS ARE ESSENTIAL

2.1. Definition and history

In its general form, a feedback loop consists of four parts: an observer, a corrector, an actuator, and a subsystem. These parts are concurrent agents that interact by sending and receiving messages. The corrector contains an abstract model of the subsystem and a goal. The feedback loop runs continuously, observing the subsystem and applying corrections in order to approach

the goal. The abstract model should be correct in a formal sense (e.g., according to the semantics of abstract interpretation [5]) but there is no need for it to be complete.

An example of a software system that contains a feedback loop is a transaction manager. It manages system resources according to a goal, which can be optimistic or pessimistic concurrency control. The transaction manager contains a model of the system: it knows at all times which parts of the system have exclusive access to which resources. This model is not complete but it is correct.

In systems with more than one feedback loop, the loops can interact through two mechanisms: *stigmergy* (two loops acting on a shared subsystem) and *management* (one loop directly controlling another). Very little work has been done to explore how to design with interacting feedback loops. In realistic systems, however, interacting feedback loops are the norm.

Feedback loops were studied as a part of Norbert Wiener's cybernetics in the 1940's [29] and Ludwig von Bertalanffy's general system theory in the 1960's [3]. W. Ross Ashby's introductory textbook of 1956 is still worth reading today [2], as is Gerald M. Weinberg's textbook of 1975 explaining how to use system theory to improve general thinking processes [27]. System theory studies the concept of a *system*. We define a system recursively as a set of subsystems (component instances) connected together to form a coherent whole. Subsystems may be primitive or built from other subsystems. The main problem is to understand the relationship between the system and its subsystems, in order to predict a system's behavior and to design a system with a desired behavior.

2.2. Feedback loops in the real world

In the real world, feedback structures are ubiquitous. They are part of our primal experience of the world. For example, bending a plastic ruler has one stable state near equilibrium enforced by negative feedback (the ruler resists with a force that increases with the degree of bending) and a clothes pin has one stable and one unstable state (it can be put temporarily in the unstable state by pinching). Both objects are governed by a single feedback loop. A safety pin has two nested loops with an outer loop managing an inner loop. It has two stable states in the inner loop (open and closed), each of which is adaptive like the ruler's. The outer loop (usually a human being) controls the inner loop by choosing the stable state.

In general, anything with continued existence is managed by one or more feedback loops. Lack of feedback means that there is a runaway reaction (an explosion or implosion). This is true at all size and time scales, from the atomic to the astronomic. For example, the binding of atoms in a molecule is governed by a simple negative feedback loop that maintains equilibrium within given perturbation bounds. At the other extreme, a star at the end of its lifetime collapses until it finds a new stable state. If there is no force to counteract the collapse, then the star collapses indefinitely (at least, until it is beyond our current understanding of how the universe works).

2.2.1. The mean field approximation

The stability of ordinary matter is explained by a feedback loop. An acceptable model for ordinary matter is the mean field approximation, which gives good results outside of critical points (see chapter 1 of [15]). To explain this approximation, we start by the simple assumption that a uniform substance reacts linearly when an external force is applied:

$$\textit{Reaction} = A \times \textit{Force}$$

For example, for a gas we can assume that density n is proportional to pressure p:

$$n = (1/kT) \times p$$

This is the Boyle-Mariotte law for ideal gases, which is valid for small pressures. But this equation gives a bad approximation when the pressure is high. It leads to the conclusion that infinite pressure on a gas will reduce its volume to zero, which is not true.

We can obtain a much better approximation by making the assumption that throughout the substance there exists a force that is a function of the reaction. This force is called the *mean field*. This gives a new equation:

$$\textit{Reaction} = A \times (\textit{Force} + a(\textit{Reaction}))$$

That is, even in the absence of an external force, there is an internal force $a(\textit{Reaction})$ that causes the reaction to maintain itself at a nonzero value. This internal force is the mean field. There is a feedback effect: the mean field itself causes a reaction, which engenders a mean field, and so on. It is this feedback effect that explains, e.g., why a condensed state such as a liquid can exist at low temperatures independent of external pressure. J. Van der Waals applied this reasoning to the ideal gas law, by adding a term:

$$n = 1/kT \times (p + a(n))$$

where n is the density of the gas and p is the pressure. According to this equation, the density n of a fluid can stay at a high value even though the external pressure is low: a condensed state can exist at low temperature independent of the pressure. The internal pressure $a(n)$ replaces the external pressure. Van der Waals chose $a(n) = a \times n^2$ by following the reasoning that internal pressure is proportional to n, the number of molecules per unit of volume, multiplied by the influence of all neighboring molecules on each molecule. This influence is assumed to be proportional to n. This gives a new equation that is a good approximation over a wide range of densities and pressures.

The mean field approach can be applied to a wide range of problems. The limits of the approach are attained near critical points. This is because the correlation distance between molecules diverges. Near a critical point, there is a phase change of the fluid, e.g., a liquid can boil to become a gas. The global behavior of the fluid changes. The behavior of matter near critical points no longer follows the mean field approximation but can be explained using scale invariance laws. We are using this behavior as a guide for the design of software systems (see Section 4).

2.3. Feedback loops in human society

Most products of human civilization need an implicit management feedback loop, called "maintenance," done by a human. Each human is at the center of a large number of these feedback loops. The human brain has a large capacity for creating these loops; some are called "habits" or "chores." If there are too many feedback loops to manage, then the brain can no longer cope: the human complains that "life is too complicated"! We can say that civilization advances by reducing the number of feedback loops that have to be managed explicitly [28]. We postulate that this is also true of software.

2.4. Feedback loops in software

Software is in the same situation as other products of human civilization. Existing software products are very fragile: they require frequent maintenance by a human. To avoid this, we propose that software must be constructed as multiple interacting feedback loops, as an effective way to reduce its fragility. This is already being done in specific domains; here are five example:

- The subsumption architecture of Brooks is a way to implement intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [4].
- IBM's Autonomic Computing initiative aims to reduce management costs of computing systems by removing humans from low-level management loops [11]. The low-level loop is managed by a high-level loop that contains a human.
- Hellerstein *et al* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [10]. Hellerstein shows two examples of adaptive systems with interacting feedback loops: gain scheduling (with dynamic selection among multiple controllers) and self-tuning regulation (where controller gain is continuously adjusted).

- Distributed algorithms for fault tolerance handle a special case of feedback where the observer is a failure detector [16]. The implementation of the failure detector itself requires a feedback loop.
- Structured overlay networks (SONs, closely related to distributed hash tables, DHTs) are inspired by peer-to-peer networks [23]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. Our own work is in the area of SONs; we explain it further in Section 4.

3. EXAMPLES OF INTERACTING FEEDBACK LOOPS

We give two examples of nontrivial systems that consist of multiple interacting feedback loops (for more examples see [25, 26]). Our first example is taken from biology: the human respiratory system. Our second example is taken from software design: the TCP protocol family.

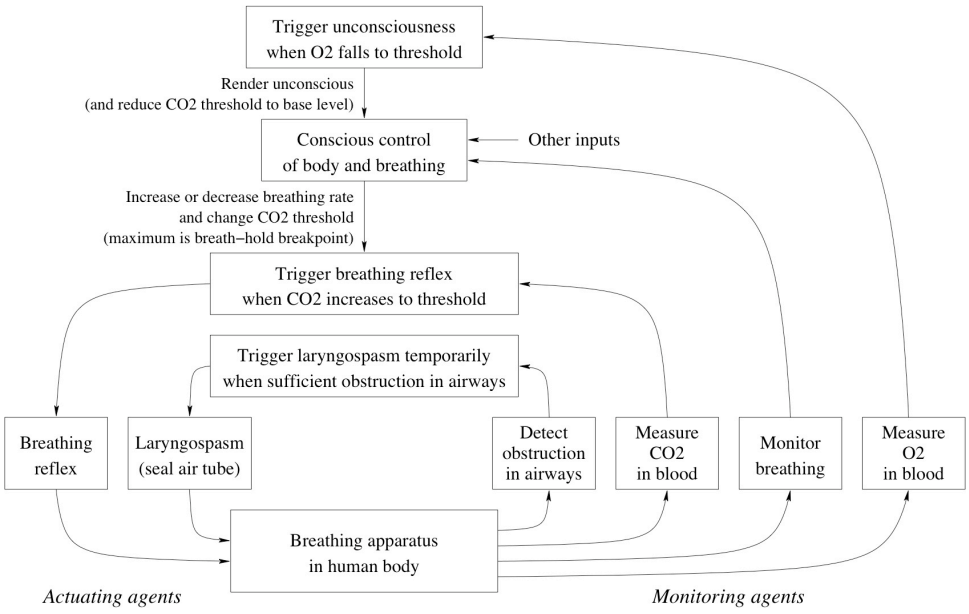


FIGURE 1: The human respiratory system as a feedback loop structure

3.1. The human respiratory system

Successful biological systems survive in natural environments, which can be particularly harsh. Studying them gives us insight in how to design robust software. Figure 1 shows the components of the human respiratory system and how they interact. The rectangles are concurrent component instances and the arrows are message channels. We derived this figure from a precise medical description of the system's behavior [30]. The figure is slightly simplified when compared to reality, but it is complete enough to give many insights. There are four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious). From the figure we can deduce what happens in many realistic cases. For example, when choking on a liquid or a piece of food, the larynx constricts so we temporarily cannot breathe (this is called laryngospasm). We can hold our breath consciously: this increases the CO_2 threshold so that the breathing reflex is delayed. If you hold your breath as long as possible, then eventually the breath-hold threshold is reached and the breathing reflex happens anyway. A trained person can hold his or her breath long enough so that the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the breathing reflex is reestablished.

We can infer some plausible design rules from this system. The innermost loops (breathing reflex and laryngospasm) and the outermost loop (falling unconscious) are based on negative feedback using a monotonic parameter. This gives them stability. The middle loop (conscious control) is not

stable: it is highly nonmonotonic and may run with both negative or positive feedback. It is by far the most complex of the four loops. We can justify why it is sandwiched in between two simpler loops. On the inner side, conscious control manages the breathing reflex, but it does not have to understand the details of how this reflex is implemented. This is an example of using nesting to implement abstraction. On the outer side, the outermost loop overrides the conscious control (a fail safe) so that it is less likely to bring the body's survival in danger. Conscious control seems to be the body's all-purpose general problem solver: it appears in many of the body's feedback loop structures. This very power means that it needs a check.

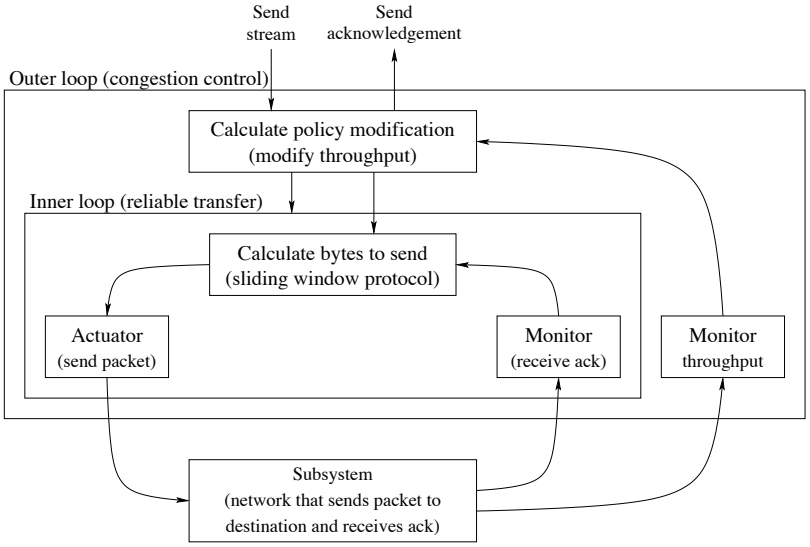


FIGURE 2: TCP as a feedback loop structure

3.2. TCP as a feedback loop structure

The TCP family of network protocols has been carefully tailored over many years to work adequately for the Internet. We consider therefore that its design merits close study. We explain the heart of TCP as two interacting feedback loops that implement a reliable byte stream transfer protocol with congestion control [12]. The protocol sends a byte stream from a source to a destination node. Figure 2 shows the two feedback loops as they appear at the source node. The inner loop does reliable transfer of a stream of packets: it sends packets and monitors the acknowledgements of the packets that have arrived successfully. The inner loop manages a sliding window: the actuator sends packets so that the sliding window can advance. The sliding window can be seen as a case of negative feedback using monotonic control. The outer loop does congestion control: it monitors the throughput of the system and acts either by changing the policy of the inner loop or by changing the inner loop itself. If the rate of acknowledgements decreases, then it modifies the inner loop by reducing the size of the sliding window. If the rate becomes zero then the outer loop may terminate the inner loop and abort the transfer.

4. STRUCTURED OVERLAY NETWORKS AS A FOUNDATION FOR FEEDBACK ARCHITECTURES

Our own work on feedback structures targets large-scale distributed applications. This work is being done in the SELFMAN project [20]. Summarizing briefly, we are building an infrastructure based on a transaction service running over a structured overlay network [19, 26]. We target our design on three application scenarios taken from industrial case studies: a machine-to-machine

messenging application, a distributed knowledge management application (similar to a Wiki), and an on-demand media streaming service [6].

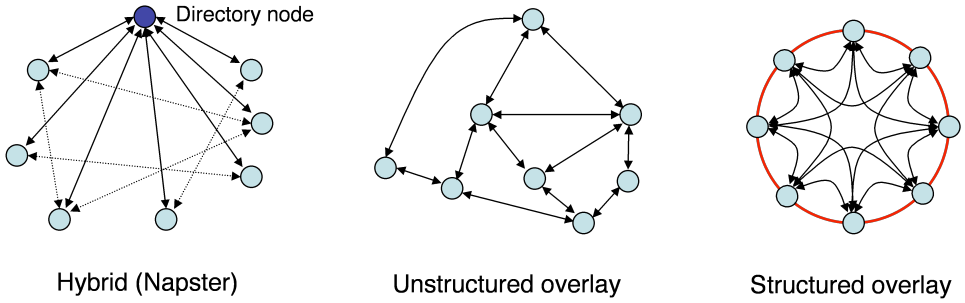


FIGURE 3: Three generations of peer-to-peer networks

4.1. Structured overlay networks

Structured overlay networks are inspired by peer-to-peer networks [23]. In a peer-to-peer network, all nodes play equal roles. There are no specialized client or server nodes. Figure 3 summarizes the history of peer-to-peer networks in three generations. In the first generation (exemplified by Napster), clients are peers but the directory is centralized. In the second generation (exemplified by Gnutella), peer nodes communicate by random neighbor links. The third generation is the structured overlay network. Compared to peer-to-peer systems based on random neighbor graphs, SONs guarantee efficient routing and guarantee lookup of data items. Almost all existing structured overlay networks are organized as two levels, a ring complemented by a set of fingers:

- *Ring structure.* All nodes are connected in a simple ring. The ring is maintained connected despite node joins, leaves, and failures.
- *Finger tables.* For efficient routing, extra links called fingers are added to the ring. The fingers can temporarily be in an inconsistent state. This has an effect only on efficiency, not on correctness. Within each node, the finger table is continuously converging to a consistent state.

Atomic ring maintenance is a crucial part of the overlay. Peer nodes can join and leave at any time. Peers that crash are like peers that leave but without notification. Temporarily broken links create false suspicions of failure.

Structured overlay networks are already designed as feedback structures. They already solve the problem of self management for scalable communication and storage. We are using them as the basis for designing a general architecture for self-managing applications. To achieve this goal, we are extending the SONs in three ways:

- We have devised algorithms for handling imperfect failure detection (false suspicions) [17], which vastly reduces the probability of lookup inconsistency. Imperfect failure detection is handled by relaxing the ring invariant to obtain a so-called “relaxed ring”, which maintains connectivity even with nodes that are suspected (possibly falsely) to be failed. The relaxed ring is always converging to a perfect ring as suspicions are resolved.
- We have devised algorithms for detecting and merging network partitions [21]. This is a crucial operation when the SON crosses a critical point (see Section 4.3).
- We have devised and implemented a transaction algorithm on top of the SON using a symmetric replicated storage and a modified version of the Paxos uniform consensus algorithm to achieve atomic commit with the Internet failure model [19].

4.2. Transactions over a SON

The highest-level service that we are implementing on a SON is a transactional storage. Implementing transactions over a SON is challenging because of churn (the rate of node leaves,

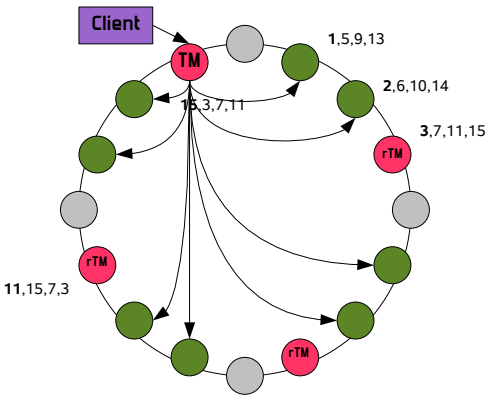


FIGURE 4: Distributed transactions on a structured overlay network

joins, and failures and the subsequent reorganizations of the overlay) and because of the Internet's failure model (crash stop with imperfect failure detection). The transaction algorithm is built on top of a reliable storage service. We implement this using symmetric replication [7].

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of Paxos [19, 8]. We use an imperfect failure detector to change coordinators in this algorithm. This is implementable on the Internet; because the failure detection is imperfect we may change coordinators too often, but this only affects efficiency, not correctness. We have shown that majority techniques work well for DHTs [22]: the probability of data consistency violation is negligible. If a consistency violation does occur, then this is because of a network partition and we can use the network merge algorithm [21].

We give a simple scenario to show how the algorithm works. A client initiates a transaction by asking its nearest node, which becomes a transaction manager. Other nodes that store data are participants in the transaction. Assuming symmetric replication with degree f, we have f transaction managers and f replicas for each other participating node. Figure 4 shows a situation with $f = 4$ and two nodes participating in addition to the transaction manager. Each transaction manager sends a Prepare message to all replicated participants, which each sends back a Prepared or Abort message to all replicated transaction managers. Each replicated transaction manager collects votes from a majority of participants and locally decides on abort or commit. It sends this to the transaction manager. After having collected a majority, the transaction manager sents its decision to all participants. This algorithm has six communication rounds. It succeeds if more than $f/2$ nodes of each replica group are alive.

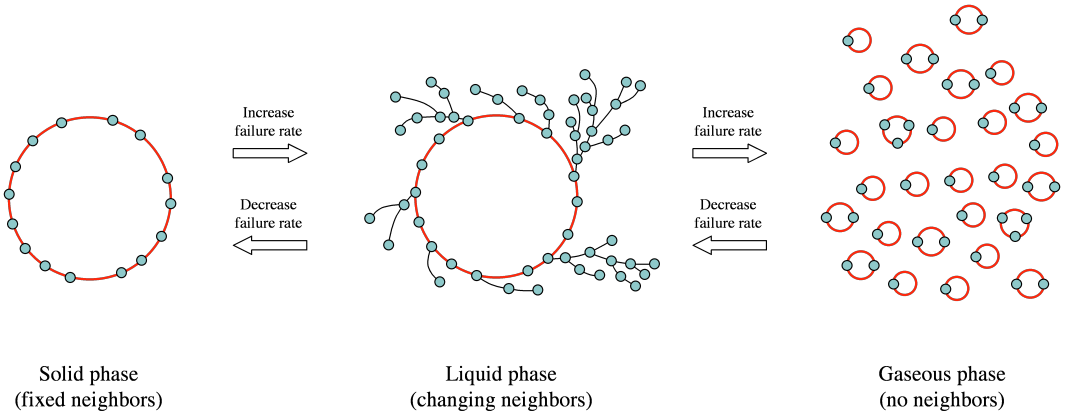


FIGURE 5: Conjectured phase transitions for a relaxed ring SON

4.3. Phase transitions in SONs and their effect on application design

At low node failure rates, a SON is a single ring where each node has fixed neighbors. This corresponds to a solid phase. At high failure rates, a SON will separate into many small rings. At the limit, a SON with n nodes will separate into n single-node SONs. This is the gaseous phase. In between these two extremes we conjecture that there is a liquid phase, the relaxed ring, where the ring is connected but each node does not have a fixed set of neighbors. When a node is subject to a failure suspicion then its set of neighbors changes.

We conjecture that for properly designed SONs phase transitions can occur for changing values of the failure rate. Figure 5 shows the kind of behavior we expect for the relaxed ring. In this figure, we assume that the node failure rate is equal to the node join rate, so that the total number of nodes is stationary. In accord with the Internet's failure model, we also assume that some of the reported failures are not actual failures (they are called failure suspicions [9]). At low failure rates, the ring is connected and does not change (solid phase). At high failure rates, the ring "boils" to become a set of small rings (of size 1, in the extreme case). At intermediate failure rates, the ring may stay connected but because of failure suspicions some nodes get pushed into side branches (relaxed ring).

We support this conjecture by citing [14], which uses the analytical model of [13] to show that phase transitions should occur in the Chord SON [23]. Specifically, [14] shows that three phases are traversed when the average network delay increases, in the following order: a region of efficient lookup, followed by a region where the longest fingers are dead (inefficient lookup), followed by a region where the ring is disconnected. We are setting up simulation experiments to verify this behavior and further explore the phase behavior of SONs.

A SON that behaves in this way will never "fail," it will just change phase. Each phase has well-defined behavior that can be programmed for. These phase transitions should therefore be considered as normal behavior that can be exposed to the application running on top of the SON. An important research question is to determine what the application API should be for phase transitions. At high failure rates, the application will run as many separate parts. When the rate lowers, these parts will combine (they will "condense" using the merge algorithm) and the application should resolve conflicts by an appropriate merge of the information stored in the separate rings. We can see that the application will probably have different consistency models at different failure rates. The transaction algorithm of the previous section will need to be modified to take this into account.

As a final remark, we conclude that the merge algorithm is a necessary part of a SON. Without the merge algorithm, condensation of a gaseous system is not possible. The SON is incomplete without it. With the merge algorithm, the SON and its applications can live indefinitely at any failure rate.

5. CONCLUSIONS

To overcome the fragility of software, we propose to build the software as a set of interacting feedback loops. Each feedback loop monitors and corrects part of the system. No part of the system should exist outside of a feedback loop. We motivate this idea by showing how it exists in real systems taken from biology and software (the human respiratory system and the Internet's TCP protocol family). If the feedback structure is properly designed, then it reacts to a hostile environment by doing a reversible phase transition. For example, when the node failure rate increases, a large overlay network may become a set of disjoint smaller overlay networks. When the failure rate decreases, these smaller networks will coalesce into a large network again. These transitions can be exposed to the application as an API so that it can be written to survive the transition. Important research questions are to determine what this API should be and how it affects application design.

In our own work in the SELFMAN project [20], we have built structured overlay networks that survive in realistically harsh environments (with imperfect failure detection and network

partitioning). We have developed a network merge algorithm that allows structured overlay networks to do reversible phase transitions. We are extending our SON with transaction management to implement three application scenarios derived from our industrial partners. We are currently finishing our implementation and evaluating the behavior of our system. Much remains to be done, e.g., we need to extend the transaction algorithm of Section 4.2 so that it also works correctly during phase transitions.

One important lesson from this work is that all future software systems should be designed so that they can support reversible phase transitions. For example, up to the work reported in [21], SONs could not merge. That means that they could not "condense" (move from a gaseous back to a solid phase) as failure rates decreased. They would "boil" (become disconnected) when failure rates increased and they would stay disconnected when the failure rates decreased. We conclude that network merge is more than just an incremental improvement that helps improve reliability. It is *fundamental* because it allows the system to survive any number of phase transitions. The system is reversible and therefore does not break. Without it, the system breaks after just a single phase transition.[1]

6. ACKNOWLEDGEMENTS

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). Peter Van Roy is the coordinator of SELFMAN. He acknowledges all SELFMAN partners for their insights and research results. In particular, he acknowledges the work on the relaxed ring, network partitioning, symmetric replication, distributed transactions, and the analytic study of SONs, all done by SELFMAN partners. Some of this work was done in the earlier PEPITO and EVERGROW projects.

REFERENCES

[1] Armstrong, Joe. "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, Nov. 2003.

[2] Ashby, W. Ross. "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): `http://pcp.vub.ac.be/books/IntroCyb.pdf`.

[3] von Bertalanffy, Ludwig. "General System Theory: Foundations, Development, Applications," George Braziller, 1969.

[4] Brooks, Rodney A. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, RA-2, April 1986, pp. 14-23.

[5] Cousot, Patrick, and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, 4th ACM Symposium on Principles of Programming Languages (POPL 1977), Jan. 1977, pp. 238-252.

[6] France Télécom, Zuse Institut Berlin, and Stakk AB. *User requirements for self managing applications: three application scenarios*, SELFMAN Deliverable D5.1, Nov. 2007, `www.ist-selfman.org`.

[7] Ghodsi, Ali, Luc Onana Alima, and Seif Haridi. *Symmetric Replication for Structured Peer-to-Peer Systems*, Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Springer-Verlag LNCS volume 4125, pages 74-85.

[8] Gray, Jim, and Leslie Lamport. *Consensus on transaction commit*. ACM Trans. Database Syst., ACM Press, 2006(31), pages 133-160.

[9] Guerraoui, Rachid, and Luis Rodrigues, "Introduction to Reliable Distributed Programming," Springer-Verlag Berlin, 2006.

[10] Hellerstein, Joseph L., Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. "Feedback Control of Computing Systems," Aug. 2004, Wiley-IEEE Press.

[11] IBM. *Autonomic computing: IBM's perspective on the state of information technology*, 2001, `researchweb.watson.ibm.com/autonomic`.

[12] Information Sciences Institute. "RFC 793: Transmission Control Protocol Darpa Internet Program Protocol Specification," Sept. 1981.

[13] Krishnamurthy, Supriya, Sameh El-Ansary, Erik Aurell, and Seif Haridi. *A statistical theory of Chord under churn*, Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, New York, Feb. 2005.

[1] An interesting question for physicists is to explain why matter behaves in reversible fashion. Software has to be designed for reversibility while simple molecules have this property implicitly.

[14] Krishnamurthy, Supriya, and John Ardelius. *An Analytical Framework for the Performance Evaluation of Proximity-Aware Overlay Networks*, Tech. Report TR-2008-01, Swedish Institute of Computer Science, Feb. 2008 (submitted for publication).

[15] Laguës, Michel and Annick Lesne. "Invariances d'échelle: Des changements d'états à la turbulence" ("Scale invariances: from state changes to turbulence"), Belin éditeur, Sept 2003.

[16] Lynch, Nancy. "Distributed Algorithms," Morgan Kaufmann, San Francisco, CA, 1996.

[17] Mejias, Boris, and Peter Van Roy. *A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks*, XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Nov. 2007.

[18] Miller, Mark S., Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. *E: Open source distributed capabilities*, 2001, `www.erights.org`.

[19] Moser, Monika, and Seif Haridi. *Atomic Commitment in Transactional DHTs*, Proc. of the CoreGRID Symposium, Rennes, France, Aug. 2007.

[20] SELFMAN: Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components, European Commission 6th Framework Programme three-year project, June 2006 – May 2009, `www.ist-selfman.org`.

[21] Shafaat, Tallat M., Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*, Journal of Peer-to-Peer Networking and Applications, Springer-Verlag, 2008 (to appear).

[22] Shafaat, Tallat M., Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks*, CoreGRID Integration Workshop, Heraklion, Greece, Springer LNCS, 2008 (to appear).

[23] Stoica, Ion, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, SIGCOMM 2001, pp. 149-160.

[24] Van Roy, Peter. *Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place*, 8th International Symposium on Functional and Logic Programming (FLOPS 2006), April 2006, Springer LNCS volume 3945, pp. 2-12.

[25] Van Roy, Peter. *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS 2006), Springer ENTCS volume 182, June 2007, pages 201-217.

[26] Van Roy, Peter, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. *Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project*, Springer LNCS, 2008 (to appear). Revised postproceedings of FMCO 2007, Oct. 2007.

[27] Weinberg, Gerald M. "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).

[28] Whitehead, Alfred North. Quote: *Civilization advances by extending the number of important operations which we can perform without thinking of them.*

[29] Wiener, Norbert. "Cybernetics, or Control and Communication in the Animal and the Machine," MIT Press, Cambridge, MA, 1948.

[30] Wikipedia, the free encyclopedia. Entry "drowning," August 2006. Internet: `http://en.wikipedia.org/wiki/Drowning`.

A.2 The Limits of Network Transparency in a Distributed Programming Language

The Limits of Network Transparency in a Distributed Programming Language

Raphaël Collet

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

December 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis committee:

Yves Deville (chair)	UCL/INGI, Belgium
Marc Lobelle	UCL/INGI, Belgium
Per Brand	SICS, Sweden
Joe Armstrong	Ericsson, Sweden
Peter Van Roy (advisor)	UCL/INGI, Belgium

Abstract

This dissertation presents a study on the extent and limits of *network transparency* in distributed programming languages. This property states that the result of a distributed program is the same as if it were executed on a single computer, in the case when no failure occurs. The programming language may also be *network aware* if it allows the programmer to control how a program is distributed and how it behaves on the network. Both aim at simplifying distributed programming, by making non-functional aspects of a program more modular.

We show that network transparency is not only possible, but also *practical*: it can be efficient, and smoothly extended in the case of partial failure. We give a proof of concept with the programming language Oz and the system Mozart, of which we have reimplemented the distribution support on top of the Distribution Subsystem (DSS). We have extended the language to control which distribution algorithms are used in a program, and reflect partial failures in the language. Both extensions allow to handle non-functional aspects of a program without breaking the property of network transparency.

Acknowledgments

I want to thank all the people that have supported me during those nine years. This thesis took a long time to mature, and would not have been possible without the following people.

I am grateful to my advisor Peter Van Roy, who introduced me to Mozart and its distributed protocols, and trained me as a researcher. His constant enthusiasm has pushed me to complete this work.

I want to thank all my colleagues at the university. In particular, the people with whom I have explored the distribution of Oz: Boris Mejías, Yves Jaradin, Valentin Mesaros, Donatien Grolaux, Kevin Glynn, Iliès Alouini; and people who experienced constraint programming in Oz with me: Fred Spiessens, Luis Quesada, Stefano Gualandi, Renaud de Landtsheer, and Isabelle Dony. I also thank all the people who worked with me, and who I forgot to list here.

I am also grateful to the people of the Mozart consortium, in particular Per Brand, Erik Klintskog, Konstantin Popov, and Christian Schulte. They helped me when I needed some guidance to modify the virtual machine and the DSS. I also thank all the scientists I have met during these years, and with whom I learned so much.

This research has been supported by the projects PIRATES (Walloon Region), PEPITO (European IST FET Global Computing), EVERGROW (European 6th Framework Programme), SELFMAN (European 6th Framework Programme), and CoreGRID (European Network of Excellence on Grid and Peer-to-Peer technologies). I thank all the institutions that funded those projects.

Because my life has changed since I began playing music, I would like to thank all the musicians that made some great music with me. The following bands have been an immense opportunity to broaden my horizons: Glycerinn (with Jack, Sarah, and Felipe), the Confused Deputies (with Boris and Fred), and the Yellows (with Jean-François, Marc, Jérôme, Alexandre, and Christophe).

My last thanks go to my family, my parents, my brother, my sister with her husband and two little boys.

Contents

1 Introduction

This dissertation presents a study on the extent and limits of network transparency in distributed programming languages. A programming language is said to be *network transparent* if a distributed program gives the same result as if it were executed on a single computer, provided network delays are ignored and no network failure occurs. The language is said to be *network aware* if the language definition allows to predict and control how the program is distributed, and its network behavior. The conjunction of both properties aims at simplifying distributed programming by separating a program's functionality, in which distribution can be ignored, from its distribution behavior, which includes network performance, partial failure (when part of the system fails), and security.

Earlier works have shown that network transparency is possible, like [Jul88]. We show that network transparency is also *practical*: it can be efficient, and smoothly extended in the case of partial failure. Efficiency is possible if the programming language supports programming with asynchronicity, which is reasonable in general, and fits well with distribution. Performance can also be tuned by the choice of distributed algorithms used by the underlying system, without affecting functionality in the case when there are no failures. Partial failure can be reflected in the language in a simple way, so that fault tolerance can be added in a modular fashion completely within the language. Security is beyond the scope of this thesis and is a subject of future work. We give a proof of concept with the programming language Oz and the system Mozart [Moz99]. We have extended the language to improve its network awareness, both for controlling distribution and handling partial failures.

This work is a continuation of earlier works on distributed programming languages, mainly done at the Swedish Institute of Computer Science (SICS). Among the results of those works are the system Mozart and the Distribution Subsystem (DSS), and two dissertations:

- Per Brand, in *The Design Philosophy of Distributed Programming Systems: the Mozart Experience* [Bra05], presents the first design, implementation, and evaluation of the distributed system Mozart.
- Erik Klintskog, in *Generic Distribution Support for Programming Systems* [Kli05], presents the design, implementation, and evaluation of the DSS, a middleware which provides efficient distribution support for programming languages.

Per Brand showed that asynchronous stream communication can be orders of magnitude faster than synchronous communication (such as Java RMI). He also showed that an Oz program was almost unchanged when going from centralized to distributed, and much simpler than a corresponding Java program.

This thesis both extends and simplifies the network-transparent distribution in Mozart. We have modified and extended the language Oz, in order to improve the network awareness in the language. We have reimplemented the distribution layer in the platform Mozart on top of the DSS middleware, and completed the latter to make it able to handle and reflect partial failures. We have also redesigned failure handling in Oz to make it completely asynchronous, and showed that it was the right default.

1.1 Distributed systems

Distributed systems are becoming ubiquitous; today all computers are connected to the internet, which provides many collaborative tools and programs. Moreover, many computers today contain multiple processors that run in parallel. This is a consequence of the current limits in increasing processors' speed, which makes manufacturers increase the number of processors instead. Computers with multiple processors can be considered as distributed systems on their own, with fast communication between processors.

Software development is progressively shifting towards concurrent and distributed programs that can take advantage of this available parallelism. Sequential programming is still acceptable for small programs, but not for large applications. By necessity, large programs will be distributed, and therefore concurrent. Alas, the introduction of concurrency into existing systems is rather poor, and inter-thread communication is often based on shared state. This model is difficult and bug-prone for programmers, which are discouraged to program in concurrent style. However, some systems propose different models for concurrent programming, like message passing in Erlang, and dataflow concurrency in Mozart.

Partial failures also make distributed programs more complex than centralized ones. Programs that run on a certain number of machines should be able to deal with faults in parts of the system. On one hand, writing distributed applications without taking partial failures into account was quickly seen as unrealistic. On the other hand, one needs abstractions to avoid the application

code to be cluttered with failure-handling code everywhere. Failure handling should be as modular as possible.

Programming languages and systems. We claim that the design of the programming language is essential when writing such applications. Extending a sequential language may lead to bad surprises, because the distributed program will *not* be sequential. So the language or its libraries should at least provide good abstractions to handle concurrency. Moreover, letting parts of a program share data introduces many subtle problems. For instance, remote references may be provided either via proxy entities, or transparently by the language itself. The programmer needs clear indication about what can be shared between sites[1]. Also, transferring data requires *serialization*.

There are basically two ways for a programming language to support the development of distributed applications. The first approach is to augment the language with libraries for distribution. Those libraries provide abstractions to make sites interact with each other. This typically involves communication channels, abstract representations of distributed entities, and so on. The programmer is responsible to integrate its application with the distribution library.

The second approach is to provide distribution as an inherent property of the programming language itself. In this case, we talk about a *distributed programming language*. A program is seen as a collection of threads and data that are spread over a set of sites. From a functional point of view, the interaction between the parts of the application is not different from the interaction between concurrent threads on a single host. However, the semantics of the language is extended to incorporate new aspects, like network latency and partial failures. This thesis explores some of the possibilities that are offered by these systems.

1.2 Models of distributed programming

The choices made to bring distribution in a programming language clearly determines the *model* that the programmer has to use. We define the programming model as a set of language constructs together with how they are executed [VH04]. It is sometimes called more informally *programming paradigm*. Examples are: declarative programming, object-oriented programming, processes with message passing. Here we are interested in the underlying models of the programming systems (the languages and their libraries) that are used to build distributed applications. We consider a few concepts that may or may not be part of a programming model.

[1] The *site* is the unit of localization. Sites execute code concurrently, and are independent of each other. A typical example is a system process.

Concurrency. By definition, a distributed program involves several activities that run more or less independently on different sites. This implies that the programming model is necessary concurrent and non-deterministic, because those properties are intrinsic to distributed systems. Therefore concurrent programming languages have an advantage over sequential languages when it comes to distribution. A concurrent language makes it possible to test a distributed program in a single site.

Moreover, good language support for controlling concurrency is also an advantage. For instance, in Oz one can synchronize two threads with a dataflow variable. This technique is simple, elegant, and is useful even if the threads are located on different sites.

Synchronous and asynchronous operations. Many programming languages only provide synchronous operations in their model. Synchronous operations are fast and natural in centralized applications, but they can be pretty slow in a distributed setting. Indeed, distributed synchronous operations often require several sites to exchange messages, and cannot proceed immediately. Asynchronous operations do not wait: the operation terminates immediately, and its effect will be performed eventually. This scheme fits well in a distributed environment: the operation may simply prepare a message and terminate, while the message delivery will perform the expected effect. The network latency is partly hidden to the user.

Most programming languages designed with distribution in mind provide asynchronous operations in their model [Van06]. In some of them, like Erlang, synchronous operations are not even part of the core of their model, but are simply defined as a derived concept [AWWV96].

Stateful and stateless data. Does the programming model makes a distinction between stateful and stateless data? This is more important than it seems at first sight. "One size fits all" does not hold for distributed data. On one hand, stateless data can be copied between sites, which provides minimum latency for operations. Once the data are copied, all operations are purely local.

On the other hand, stateful data need different protocols to handle their state and keep it consistent between sites. The state may be stationary, and behave like a server for remote operations, like distributed objects in Java Remote Method Invocation (RMI) [GJS96, Sun97]. But other protocols are useful as well. A migratable state may give better performance when a site has to perform many operations on it. Once the state has moved to the site, it can be considered as a cache, because several operations on that site may be performed in a batch. Replicating the state is yet another option, if read operations are more frequent than updates.

Multiplicity of paradigms. The ability to choose between several paradigms when writing a distributed application may lead to better programs. If a problem has a natural solution in a given programming model, its implementation will be simpler. The programmer may also choose the paradigm depending on the problem to solve *and* how various concepts of the language are distributed. The system does not force the programmer to emulate a distributed protocol on top of inappropriate concepts.

Distributed and local references. In programming systems that provide distribution as a separate library, distributed and local references often have different interfaces. An example is given by Java RMI, where distributed objects introduce exceptions where equivalent local objects would not. Distributed objects have a slightly different semantics, too. Reference integrity of remote objects is not guaranteed in general, for instance [Sun97]. Turning local objects into distributed ones may break an application.

Making no visible difference between local and distributed data allows the runtime system to choose the right representation for a given datum. A local reference that is sent to a remote site automatically switches to a distributed representation for the corresponding data. The conversion may be reverted once the distributed reference is used by one site only. This implies less effort from the programmer.

Partial failures. They are inherent to distributed systems, so reflecting failures in the programming model is very important. It basically provides the programmer with a semantic representation of the failure. This semantic support allows the programmer to *reason* about failures, and handle them properly. Besides a semantic representation, the programming model should also provide a way to detect failures. Failure detectors are the basic ingredient of failure handlers.

We believe that causing a partial failure by program can be useful: it may simplify a failure handler. Sometimes a component cannot be fixed easily because it strongly depends on another component, which has failed. Making the former one fail may accelerate the failure recovery, which can be handled at a higher level in the application.

1.3 Network transparency

As we said, network transparency states that the result of a distributed program is the same as if it were executed on a single computer, in the case when no failure occurs. The meaning can be more precise if we consider network transparency at the level of the programming language. A given entity or piece of code is network transparent if its semantics does not depend on whether it is run in a distributed environment, provided no failure occurs.

Several forms of transparency have been proposed in the literature. The following ones are taken from [ISO98]. They use the term *resource* in a very general sense. When applied to a programming language, a resource typically corresponds to an object or an agent.

- *Access transparency* masks differences in data representation and invocation mechanisms, to provide a single and uniform access to resources.
- *Location transparency* states that the user of a resource should not be aware of where the resource is physically located. *Migration transparency* states that the user should not be aware of whether a resource or computation has the ability to move to a different location, while *relocation transparency* guarantees that its migration should not be noticeable to the user.
- *Replication transparency* makes a resource appear as unique even if it is replicated among several locations. *Persistence transparency* makes no difference between resources located in volatile and permanent memory.
- *Failure transparency* masks the failure and possible recovery of resources or computations.
- *Transaction transparency* masks coordination of activities amongst a configuration of entities to achieve consistency.

Our definition of network transparency covers the above notions of access, location, migration, relocation, and replication transparency. We also cover transaction transparency in the sense that primitive operations on distributed entity should be atomic, just like in the centralized case. Persistence transparency is rarely present in programming languages, where the program's memory is often considered as volatile. Our definition does not cover failure transparency, and our proposal for a distribution model in Chapter 4 will even make failures explicit in the language.

In practice. Some researchers have maintained that network transparency cannot be made practical, see, e.g., Waldo *et al.* [WWWK94]. They cite four reasons: pointer arithmetic, partial failure, latency, and concurrency. The first reason (pointer arithmetic) disappears if the language has an abstract store. The second reason (partial failure) requires a reflective fault model, which we designed for the Distributed Oz language. The authors of the paper above expected that failures could be always hidden behind abstractions. They were wrong: sometimes failures cannot be resolved locally, and requires some global action.

The final two reasons (latency and concurrency) lead to a layered language design. Latency is a problem if the language relies primarily on synchronized operations, like procedure calls. The authors of the paper explicitly mention the disappointing experience of remote procedure calls, that they see as the

only way to make distributed objects. In the terminology of Cardelli, latency is a network awareness issue [Car95]. The solution is that the language must make asynchronous programming both simple and efficient.

Concurrency is also seen as an obstacle. A closer look at the paper reveals that the authors actually talk about *shared-state concurrency*. Indeed, most people consider that programming languages are always stateful. The problem with concurrency in those languages is how to control concurrent accesses to state, in order to avoid invariant violations and glitches. A solution is to support a form of *stateless concurrency*, known as *dataflow concurrency*. Concurrent threads interact by sharing values, and automatically synchronize on the availability of data. Threads can be programmed as if they were never waiting for data. An example of this kind of communication is pipelining in Unix-like systems. Moreover, values can be copied between memory stores, which substantially reduce their latency. Using dataflow concurrency can reduce the need for shared state to a minimum. We conclude that language design is an important issue for network transparency.

1.4 Thesis and contributions

This thesis proves that network transparency is practical in a distributed programming language. It gives concrete proposals of language extensions that deal with performance and failure handling, and demonstrates their usage with practical examples. It also describes the implementation of the platform Mozart/DSS, with insights on various implementation issues.

Contributions. This thesis extends, simplifies, and completes the past work on network-transparent distribution in Mozart. The initial distribution model and the initial failure detection model [HVS97, VHB99] formed the core of the first distributed release of Mozart in 1999. Erik Klintskog made the first design of a distribution subsystem (DSS) in which the distribution support is completely factored out of the run-time emulator [Kli05]. The work on the DSS was incomplete, however. The present thesis brings the following scientific and technical contributions.

- We extend the distribution model of Oz to make it customizable. We introduce entity annotations, so that the programmer has the ability to choose between several protocols for each entity, including its distributed memory management.
- We design a failure handling model for Oz that is simpler and more expressive than the initial one. Each language entity produces a stream of fault states that is extended asynchronously, whenever the entity's fault state changes.
- We design an effective post-mortem finalization mechanism based on the fault stream. This mechanism did not exist in the language.

- We give distributed programming patterns that show how the system simplifies programming robust distributed systems.

- We complete Erik Klintskog's work by presenting more precise definitions of the distribution protocols that include failure handling, in particular the mobile state protocol.

- We have rebuilt the distribution support of the platform Mozart on top of the DSS library, and implemented the new distributed programming model of the language. The reflection of failures in the language, and the implementation of the new language features (annotations, fault stream, making entities fail) are entirely our work.

- We have also completed the implementation of the DSS. In particular we have rewritten all entity protocols such that they can handle partial failures. We have also extended the DSS interface to handle and reflect entity failures.

- We evaluate the new implementation in a realistic setting.

Publications. The following publications contain substantial contributions by the author on the topics of this thesis. The first two papers focus on the extension of a mobile state protocol to make it handle failures. That protocol is part of the platform Mozart. The implementation of that protocol is now part of the DSS. Its semantics as a migratory protocol are given in Chapter 7, and its implementation is described in Chapter 8.

- Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. *Lecture Notes in Computer Science*, 1686:32–46, 1999 [VBHC99].

- Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. Path redundancy in a mobile-state protocol as a primitive for language-based fault tolerance. Research Report RR2000-01, Université catholique de Louvain, Département INGI, 2000 [BVCK00].

The next two papers propose a formal definition of lazy computations in terms of concurrent constraints. That definition led to an efficient distributed implementation of that concept. Laziness is mentioned in Chapter 3, and a concrete example of its usage is shown in Chapter 5. Its semantics are defined in Chapters 6 and 7, and its implementation is described in Chapter 8.

- Alfred Spiessens, Raphaël Collet, and Peter Van Roy. Declarative laziness in a concurrent constraint language. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03, 2003 [SCV03].

- Raphaël Collet. Laziness and declarative concurrency. 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity PostJava'04, 2004 [Col04].

The fifth paper shows how to design a transactional system for a distributed store of objects, on top of an overlay network. That work put some emphasis on what kind of primitives were desired in the language to handle failures.

- Valentin Mesaros, Raphaël Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Research Report RR2005-01, Université catholique de Louvain, Département INGI, 2005 [MCGV05].

The latter paper is the author's proposal to favor asynchronous failure handling in a distributed programming language. The paper contains the essential contributions of Chapter 4.

- Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In C. Dony et al., editor, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 121–140. Springer-Verlag, 2006 [CV06].

1.5 Structure of the document

Chapter 2 gives an introduction to the programming language Oz. That language is the vehicle we have chosen to explain all our proposals. This chapter may safely be skipped if the reader already knows the language. Note that most code snippets in Oz appear in special figures called "snippets".

Chapters 3 and 4 detail our proposals for dealing with an application's distributed behavior, and failure handling, respectively. The programming model is exposed and explained, together with some practical intuition for all concepts. Concrete examples using those language features are given in Chapter 5.

Chapters 6 and 7 propose a formal definition of the language, in centralized and distributed settings, respectively. Operational semantics are given for the core of the language, and the centralized semantics are refined into distributed semantics that reflect the aspects related to distribution.

Chapter 8 describes the implementation of Mozart/DSS, the reimplementation of Mozart on top of the DSS library. It gives a definition of the protocols that are used to implement basic operations on distributed entities, sketches the DSS application programming interface, and explains how the distribution support is implemented on top of it.

Chapter 9 evaluates the work done so far. Comparisons with other systems are made. Chapter 10 concludes the work. Scientific results are summarized,

and future directions are given. Appendix A gives a summary of the model, and the language extensions proposed in this work.

2

An introduction to Oz

In this text, we use the programming language Oz as a vehicle to express a certain number of concepts related to distributed programming. The concepts themselves are language independent, but few programming languages are able to express them in a natural way. Oz makes it possible, thanks to its support for multiple programming paradigms, among which we find declarative programming, dataflow concurrency, and object-oriented programming [Smo95, VH04]. This chapter gives a quick introduction to the language, and the basic model of its distribution. A formal definition of the language is given in Chapter 6.

2.1 The kernel language approach

The language Oz is based on a small set of concepts, that form a *kernel language*, called Kernel Oz. In the kernel language, all concepts are primitive: they cannot be defined in terms of each other. The full language is defined as the kernel language extended with *language abstractions*, i.e., programming abstractions with syntactic support. All those abstractions are defined in terms of Kernel Oz, so that every program can be reduced to an equivalent program in the kernel language.

The advantage of the kernel language approach is that the language is defined by layers. The bottom layer is the kernel, with all primitive concepts. The upper layers then define abstractions built on concepts defined in layers below. In practice, two layers are enough to define a very expressive language. Figure 2.1 on the following page shows a certain number of concepts that are present in the language. All concepts in bold font are part of the kernel language, while the others are derived concepts. The arrows indicate from what a given concept derives.

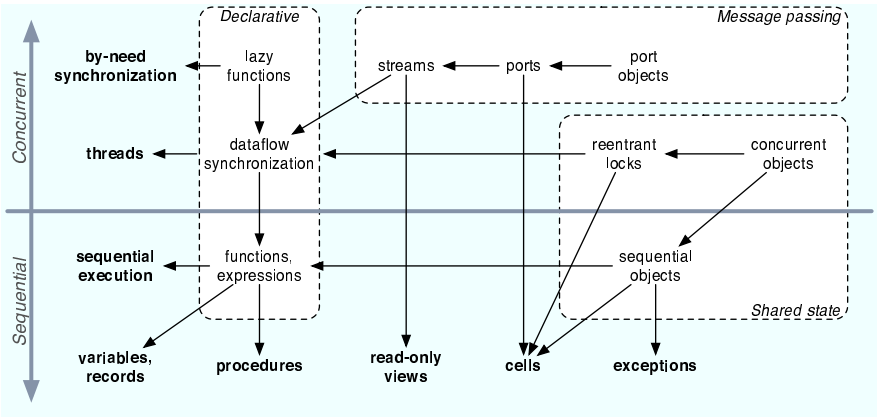


Figure 2.1: Primitive and derived concepts in the language Oz

The paradigm space. When writing a program or module, the programmer will often use only a subset of the concepts provided by the programming language. Every subset of concepts defines a *programming model*, or *paradigm*. Each paradigm comes with its own set of techniques and design rules. As you can see, the derived concepts shown in figure 2.1 are not placed randomly. There are two major axes in the diagram.

The base of all derived concepts is a sequential declarative language, which can be made a functional language with the appropriate language support. It contains no state and no concurrency. The language is mainly enriched by adding state (right direction in the diagram) and concurrency (upper direction). Moreover, the derived concepts are grouped together to form three major paradigms, namely declarative, message passing, and shared state programming.

Section 2.2 introduces the kernel concepts of the core language, together with concurrency. This part of Oz is completely declarative. The non-declarative kernel concepts are introduced in section 2.3. In section 2.4 we present syntactic extensions, and language abstractions used in the text. There we show two important techniques for controlling concurrency in the presence of state: message passing with *ports*, and *locks*. Note that we have chosen to introduce concurrency before state in the presentation. This is because it is possible to make distributed programs that are declarative; all they need is concurrency. The base distribution model is given in Section 2.5.

2.2 Declarative kernel language

An Oz program consists in a set of *threads* computing over a shared *store*. Threads execute statements in a sequential way. The program will be sequential if it contains a single thread. The store is the memory of the program; it

contains logic variables and data structures used by the threads. Threads are connected to the store by variables.

2.2.1 The store

The program store has two parts: the so-called *constraint store* and the *procedure store*. The constraint store contains logic statements about the program's variables. For now consider that those logic statements have the form $x{=}t$, where x is a variable, and t is either a variable or a value. The procedure store contains the procedures created by the program.

Values. The different kind of values are *integers*, *atoms*, *names*, and *records*. Names are primitive entities that have no structure; they are used to give an identity to other data structures, like procedures. The boolean values **`true`** and **`false`** are defined as names.

Atoms are literal constants that are defined by a sequence of characters. Syntactically they are words starting with lowercase letter, like `atom` or `nil`, or they are surrounded by quotes, like `´Hello world´` or `´|´`. A *literal* is either a name or an atom, and a *simple value* is either a literal or an integer.

A *record* is a compound value $l(k_1{:}v_1 \ldots k_n{:}v_n)$ formed by a *label* l (a literal) and *fields* $v_1, \ldots, v_n$. Each field v_i is associated to a key k_i, which is a simple value. An example is `person(name:raph age:32)`. A *tuple* is a record whose keys are the integers 1 to n. Syntactically the keys can be omitted, as in `person(raph 32)`, which is equivalent to `person(1:raph 2:32)`. Lists are defined in terms of tuples and atoms as follows. A list is either the empty list (the atom `nil`), or a head-tail pair `´|´`(xy). The latter can be written infix as $x\,|\,y$.

Variables. The store contains logic variables that can be bound to values and other variables. Upon creation, a variable x is always unbound. It is bound to a value v if the store contains the statement $x{=}v$. It can be bound to at most one value, and the binding cannot change over time. A variable bound to a value is said to be *determined*.

Variables x and y can also be bound together, if the store contains $x{=}y$. The binding relation is transitive: if x is bound to a value v, then y is also bound to the same value v. Note that undetermined variables can be bound together.

2.2.2 Declarative statements

The declarative kernel statements are given in Figure 2.2 on the following page.

S	::=	**skip**	*empty statement*
	|	$S_1 S_2$	*sequential composition*
	|	**local** X **in** S **end**	*declaration*
	|	$X{=}Y$ | $X{=}v$	*unification*
	|	**if** X **then** S_1 **else** S_2 **end**	*conditional statement*
	|	**proc** {P $X_1 \ldots X_n$} S **end**	*procedure creation*
	|	{P $X_1 \ldots X_n$}	*procedure application*
	|	**thread** S **end**	*thread creation*
	|	{`WaitNeeded` X}	*by-need synchronization*
v	::=	s	*simple value*
	|	$l(k_1{:}X_1 \ldots k_n{:}X_n)$	*record*
s	::=	l | i	*literal or integer*
l	::=	*atom* | **true** | **false**	*atom or name*
P, X, Y	::=	*identifier*	*variable identifier*

Figure 2.2: Declarative kernel concepts of Oz

Empty statement and sequential composition. The execution of the statement **skip** simply has no effect. The sequence of statements $S_1 S_2$ first executes S_1, then executes S_2.

Declaration. The statement **local** X **in** S **end** creates a fresh variable x in the store, and reduces to the statement S, where X is associated to x. It defines a *lexical scope* between the keywords **in** and **end** for the identifier X.

In order to be executable, a statement must have all its free identifiers correspond to store variables. For the sake of simplicity, in the rest of the text, we will refer to the variable corresponding to identifier X as "the variable X."

Unification. Variables are bound in the store by unification. The statements $X{=}Y$ and $X{=}v$ add the necessary variable bindings to make their arguments equivalent. For instance, the following statements binds `R` to the record `foo(a:X b:Y)`, then by unification of records makes both `X` and `Y` equal to `Z`, and finally binds them all to 42.

```
R=foo(a:X b:Y) R=foo(a:Z b:Z) Z=42
```

A unification triggers an error if its arguments cannot be made equal. The error shows up as an exception (see below).

Conditional statement. The statement **if** X **then** S_1 **else** S_2 **end** blocks until the variable X is determined. It reduces to S_1 if X equals **true**, and S_2

if X equals **`false`**. Other values trigger an error (in the form of an exception, see below).

Procedure creation. The statement **`proc`** `{`$P\ X_1 \dots X_n$`}` S **`end`** creates a fresh name ξ, adds the procedure $\lambda X_1 \dots X_n.S$ under the name ξ in the procedure store, and reduces to the statement $P{=}\xi$. Since the name ξ is fresh, the procedure store defines a mapping of names to procedures. Note also that the name makes the procedure a first-class entity: it can be passed as arguments and stored in data structures.

The procedure may refer to anything in the lexical scope of its definition. Those external references are determined by the variables corresponding to the free identifiers in S that do not occur in the procedure's parameters $X_1, \dots, X_n$. The **`proc`** statement defines a lexical scope for the identifiers $X_1, \dots, X_n$. Note that the declaration of the identifier `P` is done outside the procedure definition. This implies that the order of creation of procedures that use each other is irrelevant, provided they are all created before their use.

Procedure application. The statement `{`$P\ Y_1 \dots Y_n$`}` blocks until the variable P is determined. If P is the name of an n-ary procedure $\lambda X_1 \dots X_n.S$ in the procedure store, it reduces to the statement S where each X_i corresponds to the variable Y_i. If P is not a procedure, or a procedure with a different arity, the statement triggers an error.

Thread creation. The statement **`thread`** S **`end`** creates a new thread that consists of the statement S. The new thread is independent from the current thread. The threads in a program execute concurrently. The following primitive store operations are guaranteed to be *atomic*: creating a fresh name or variable, binding a variable, and storing a procedure. This makes the concurrency in Oz fine-grained.

A thread is *runnable* if its first statement is executable. When the first statement of the thread blocks on a variable, we say that the thread is *blocked* or *suspended*. It remains blocked until the variable is determined by the store. It then becomes runnable again.

By-need synchronization. We say that a variable X is *needed* when either it is determined, or a thread waits for its determination. This property is *monotonic*: when a variable is needed, it remains needed. The statement `{WaitNeeded` X`}` blocks until the variable X becomes needed. This primitive is used to attach a *lazy* computation to the variable X: a thread blocks until X becomes needed, then computes the value of X.

```
thread
   {WaitNeeded X}     % block until X is needed
   ...                % compute a value and assign it to X
end
```

2.2.3 A few convenient operations

Here are a few extra statements, that complete the declarative kernel language with common primitive operations.

Tests. The statement X=(Y`==`Z) blocks until there is enough information in the constraint store to logically entail Y=Z or $Y \neq Z$. The statement then reduces to X=**`true`** or X=**`false`**, respectively. The operator `\=` is similar, but returns the opposite results. The statement

$$X{=}Y \ op \ Z \qquad \text{with } op \in \{\texttt{<}, \texttt{=<}, \texttt{>}, \texttt{>=}\}$$

blocks until both Y and Z are determined to atoms or integers. It then reduces to X=**`true`** or X=**`false`**, depending on the result of the comparison. Atoms are ordered lexically, and atoms and integers are not comparable.

Arithmetic operations. The statements

$$X{=}\texttt{~}Y \qquad X{=}Y \ op \ Z \qquad \text{with } op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \textbf{\texttt{div}}, \textbf{\texttt{mod}}\}$$

block until their arguments (Y and Z) are determined to integers, and reduce to X=i, where i is the result of the corresponding operation.

Record operations. The statements

`{Label` X Y`}` `{Width` X Y`}` `{Arity` X Y`}`

block until X is determined to be a record $l(k_1{:}v_1 \ldots k_n{:}v_n)$. They then reduce to Y=l, Y=n, and Y=k_1`|` $\ldots$ `|`k_n`|nil`, respectively.

The statement X=Y`.`Z blocks until Z is determined to a simple value and Y to a record $l(k_1{:}v_1 \ldots k_n{:}v_n)$. If Z is equal to k_i for a certain i, the statement reduces to X=v_i. Otherwise an exception is raised.

2.3 Nondeclarative extensions

Figure 2.3 on the next page shows nondeclarative extensions to the language. They include exceptions, state, and read-only views.

2.3.1 Exceptions

Try statement. The statement **`try`** S_1 **`catch`** X **`then`** S_2 **`end`** starts by executing S_1. If S_1 terminates normally, the statement reduces to **`skip`**. If an exception x "escapes" from S_1, i.e., x is raised inside S_1 but not caught, the **`try`** statement reduces to S_2 (the *exception handler*), with X corresponding to variable x. Note that the **`try`** statement can only catch exceptions raised in its own thread.

S	::=	**try** S_1 **catch** X **then** S_2 **end**	*try statement*
	|	**raise** X **end**	*raise statement*
	|	`{FailedValue` X Y`}`	*failed value creation*
	|	X`=!!`Y	*read-only view creation*
	|	`{NewCell` X Y`}`	*cell creation*
	|	X`=`Y`:=`Z	*cell exchange*

Figure 2.3: Nondeclarative kernel concepts of Oz

Raise statement. The statement **raise** X **end** throws an exception with the variable X. The exception will be caught by closest exception handler in the thread, if it exists.

Failed values. A *failed value* is a special value that encapsulates an exception. The statement `{FailedValue` X Y`}` creates a failed value v encapsulating the variable X, and reduces to Y=v. Any statement that attempts to use the value v automatically raises an exception with X.

Failed values are used to pass exceptions between threads. This is particularly useful when an exception occurs in a thread that is responsible to compute the value of a variable Z. If the thread binds Z to a failed value encapsulating the exception, all the other threads that use Z will know why the value of Z could not be determined.

```
thread
   try
      local V in
         ...                  % compute a value V
         Z=V                  % return the value in Z
      end
   catch E then
      {FailedValue E Z}       % return a failed value in Z
   end
end
```

2.3.2 Read-only views

"Bang bang" operator. The statement X`=!!`Y creates a *read-only view* u of variable Y, and reduces to X=u. A read-only view of a variable is logically equal to that variable, but it cannot be bound by the program; every statement that attempts to bind it blocks. When the variable is determined to a value, the read-only view is also bound to that value.

Read-only views are used to protect abstractions from accidental bindings that may break them. They are often used to build robust *streams*. A stream is a list that is built incrementally. During its construction, a read-only view

of its tail variable is put in the list tuple. This prevents the consumer of the stream to bind the tail, and provoke a failure in the producer of the stream.

2.3.3 State

The primitive concept of state in Oz is the *cell*. A cell is a mutable container for a variable. Cells are contained in a new part of the store called the *cell store*. A cell is a first-class value, identified by a name, and its contents is given by a (possibly determined) variable. Similarly to the procedure store, the cell store defines a mapping of names to variables.

A cell can be used by a single thread or shared among several threads. Concurrent accesses to a cell are relatively easy to synchronize, because read and write operations are combined in a single, atomic *exchange* operation. Concurrency control abstractions can be built with cells, like ports and locks (see Section 2.4).

Cell creation. The statement `{NewCell` X Y`}` creates a fresh name ξ, adds the pair ξ:X in the cell store, and reduces to the statement Y=ξ.

State exchange. The statement X`=`Y`:=`Z blocks until Y is determined. If Y is a cell ξ:w, the cell store is updated with ξ:Z, and the statement reduces to X=w.

For instance, if `C` is a cell that contains an integer, the following procedure adds `N` to the contents of the cell. Assume two threads update concurrently `C` with `AddCounter`. If the initial state is x, the first thread takes x and put a variable y in the cell, then the second thread takes y and put a variable z. The second thread computes the new state z with y, and automatically waits for the first thread to determine y. No race condition occurs if all state updates are done this way.

```
proc {AddCounter N}
   local X Y in
      X=C:=Y       % get contents X, and put new contents Y
      Y=X+N        % determine the value of Y
   end
end
```

2.4 Syntactic convenience

This section introduces syntactical convenience that corresponds to the full language, at least the part of the language that is relevant for this text. All the rules in this section suggest how to rewrite statements in the full language to statements in the kernel language.

Comments. All the characters that follow the percent sign (`%`) until the end of the line are comments.

2.4.1 Declarative programming

Declarations. Multiple variables can be declared simultaneously. For instance,

```
local X Y in S end     ⇒  local X in local Y in S end end
```

If a declaration statement comprises the body of a procedure definition or the branch of a conditional, **local** and **end** can be omitted. For example:

```
proc {P} Y in S end    ⇒  proc {P} local Y in S end end
```

Declaration can be combined with initialization through unification:

```
local X=5 in S end     ⇒  local X in X=5 S end
```

Expressions. We first define the statement Z=`{`$P\ X_1 \ldots X_n$`}` as shorthand for `{`$P\ X_1 \ldots X_n\ Z$`}`. Similarly, nesting of record construction and procedure application avoids declaration of auxiliary variables. For example:

```
X=b({F N+1})           ⇒  local Y Z in
                             Y=N+1 X=b(Z) {F Y Z}
                          end
```

Record construction is given precedence over procedure application to allow more procedure definitions to be tail recursive. The construction is extended analogously to other statements, allowing statements as expressions. For example:

```
X=local Y=2 in         ⇒  local Y=2 in X={P Y} end
     {P Y}
  end
```

Procedure definitions as expressions are tagged with a dollar sign (`$`) to distinguish them from definitions in statement position:

```
X=proc {$ Y} Y=1 end   ⇒  proc {X Y} Y=1 end
```

Another common expression is the *anonymous* variable:

```
_                      ⇒  local X in X end
```

Functions. Motivated by the functional notation of procedure calls, we define a function of n arguments as being equivalent to a procedure of n+1 arguments, the extra argument being bound to the result of the function body, which is an expression:

```
fun {Inc X} X+1 end    ⇒  proc {Inc X Y} Y=X+1 end
```

Lazy functions. Function definitions with the decoration `lazy` create a thread that uses `WaitNeeded` to wait for the result to be needed. Chapter 6 proposes a translation that avoids creating threads in the presence of tail recursive calls.

```
fun lazy {Inc X}          ⇒  proc {Inc X Y}
   X+1                          thread
end                                {WaitNeeded Y} Y=X+1
                                end
                             end
```

Lists. Complete lists can be written by enclosing the elements in square brackets. For example, `[1 2]` abbreviates `1|2|nil`, which abbreviates `´|´(1 ´|´(2 nil))`.

Infix tuples. The label `´#´` for tuples `´#´(X Y Z)` can be written infix: `X#Y#Z`.

Pattern matching. Programming with records and lists is greatly simplified by pattern matching. For instance, a pattern matching conditional

```
case X of person(name:N age:A) then S1 else S2 end
```

is an abbreviation for

```
if {Label X}#{Arity X} == person#[age name] then
   N A in X=person(name:N age:A) S1
else S2 end
```

The **else** part is optional and defaults to **else skip**. Multiple clauses are handled sequentially, for example:

```
case X                    ⇒  case X of f(Y) then S1 else
of f(Y) then S1                 case X of g(Z) then S2 end
[] g(Z) then S2              end
end
```

The **try** statements are also subject to pattern matching. For example:

```
try S1                    ⇒  try S1 catch Y then
catch f(X) then S2              case Y of f(X) then S2
end                             else raise Y end
                                end
                             end
```

Loops. Recursive functions are expressive enough to implement all kinds of loops in the language. Oz supports a simple yet powerful **for** loop on lists:

```
for X in L do S end    ⇒  {ForAll L proc {$ X} S end}
```

where `ForAll` is defined as

```
proc {ForAll L P}
   case L of X|T then {P X} {ForAll T P} else skip end
end
```

Waiting for determinacy. A common abstraction is the procedure `Wait`, that blocks until its argument is determined. It is often used to explicitly synchronize threads. It can be defined as follows, using the blocking behavior of `==`.

```
proc {Wait X} _=(X==1) end
```

2.4.2 Message passing

Ports. A port is associated to a stream (defined in Section 2.3.2 above), which lists all the messages sent on the port. Ports are defined by two operations: `NewPort`, which creates a port and its stream, and `Send`, which sends a message on a given port. They can be defined in terms of cells as

```
fun {NewPort S}
   T in S=!!T {NewCell T}
end
proc {Send P X}
   T in X|!!T=P:=T
end
```

Note, however, that ports are not truly defined like that. When it comes to distribution, they do not behave like cells, but have a behavior of their own. This is because ports are intrinsically asynchronous, which cells are synchronous.

Ports are very convenient to handle nondeterminism, since they are asynchronous (they never block), and all the messages sent to a port are serialized into a list (the stream).

Port objects. A *port object* consists in a port and a thread that reads sequentially its message stream. Because the message processor is sequentially reading a list, it can be written as a simple recursive function. The latter can use an accumulator to maintain a state.

Snippet 2.1 on the following page shows a simple abstraction that builds port objects. The argument function `Func` takes the object's current state and a message, and returns the new state of the object. The function `FoldL` is used to apply `Func` on every message. An example is shown below, with a object `Counter`. This object recognizes three kind of messages: `inc`, `inc(N)`, and `get(N)`. See how the latter binds `N` to the current value of the counter.

```
fun {NewPortObject Init Func}
   S in
   thread {FoldL S Func Init} end
   {NewPort S}
end
fun {FoldL L F I}
   case L of X|T then {FoldL T F {F I X}} else nil end
end

local
   fun {F Count Msg}
      case Msg
      of inc    then Count+1
      [] inc(N) then Count+N
      [] get(N) then N=Count Count
      end
   end
in
   Counter={NewPortObject 0 F}
end
{Send Counter inc(3)}     % increment counter by 3
```

Snippet 2.1: An abstraction to create port objects, and an example of a counter object

Active objects. Active objects are similar to port objects, except that they use a stateful object instead of a function to process the messages. This technique is pretty easy to work with, because the underlying object is used sequentially.

2.4.3 Stateful entities

State operations. The full language supports two simplified versions of the state exchange operation, to simply read and write the state.

```
X=@C          ⇒  X=C:=X
C:=X          ⇒  _=C:=X
```

Objects and classes. In Oz an *object* is defined as a unary procedure, that takes a *method* as its argument. The method is represented as a record, and is therefore first-class. An object usually maintains a proper state. A *class* is a value that creates objects. A precise kernel equivalent of classes, and their inheritance mechanism, is given in [VH04].

Snippet 2.2 on the next page illustrates the class syntax with an example with two classes and one object. A base class `Stack` defines an attribute `elements`, which is identified by an atom. It also defines four methods: `init`, `isEmpty`, `push`, and `pop`. The state operators are extended to attributes. The

```
class Stack
   attr elements       % list of elements, from top to bottom
   meth init
      elements:=nil            % initializer
   end
   meth isEmpty(B)
      B=(@elements==nil)
   end
   meth push(X)
      T in T=elements:=X|T     % put X in front of elements
   end
   meth pop(X)
      T in X|T=elements:=T     % extract front element
   end
end

class Stack2 from Stack        % Stack2 extends Stack
   meth top(X)
      {self pop(X)} {self push(X)}
   end
end

S={New Stack2 init}     % create an object of class Stack2
{S push(42)}            % call method push(42) of Obj
```

Snippet 2.2: Two classes and an object

class `Stack2` extends the class `Stack`. It defines a method `top`, which is implemented with the methods `push` and `pop` of the object, which is accessible by the keyword **`self`**. Then the function `New` is used to instanciate the class, and initialize the object.

There is no concurrency control by default in objects. Concurrent method invocations will be executed concurrently, and state updates are subject to the same kind of atomicity as cells. Objects often use *locks* to create critical sections inside methods.

Locks. A lock is a binary semaphore, which controls the access to the lock itself. At most one thread can be in a given lock. The only operation takes the lock, executes a statement, and releases the lock. If another thread already owns the lock, the operation blocks until the lock is available. The **`lock`** statement is translated as follows.

`lock` `L` **`then`** S **`end`** $\Rightarrow$ `{L proc {$}` S **`end`**`}`

The lock `L` is created by the following function, which implements a basic lock with a cell and a procedure.

```
fun {NewLock}
   C={NewCell unit}
in
   proc {$ P}
      X Y in X=C:=Y {Wait X} {P} Y=X
   end
end
```

When the lock is applied, it places a new variable into its cell, and waits until the former variable in the cell is determined. Once it is determined, the lock is available. The lock then executes the statement, which is abstracted by a nullary procedure. It then release the lock by binding the new variable to the value. Several threads applying the same lock will form a chain, and pass the value **unit** between each other via a shared variable. The cell's function is to connect a thread to its successor in the waiting queue.

The language Oz actually provides *reentrant locks*. Those locks permit a thread to take the same lock several times. This is useful when two procedures or methods call each other, and protect a shared state with the same lock. For a definition of reentrant locks in Kernel Oz, see [VH04]. Note that distinct locks are not connected to each other in any way; deadlocks are possible, and the language provides no deadlock detection mechanism.

2.5 Distribution

In Oz, a distributed program is usually defined as a centralized program where entities and threads would be partitioned among *sites*. One can also define a distributed program as a set of centralized programs running on their own sites, and sharing language entities. Both definitions are in fact valid and equivalent, because the language is *network-transparent*.

Most of the distribution is hidden to the programmer, as shared entities keep their semantics almost intact. What happens is that the programming system uses dedicated protocols to implement the entities' semantics. In Mozart the distribution of entities is designed to give the programmer full control over network communication patterns that occur because of language operations. Not all entities use the same protocol, every entity use a protocol that is adapted to its nature: mutable, immutable, or transient. This subject is discussed in detail in Chapter 3.

2.5.1 Application deployment

The deployment of an application covers two situations that are handled differently. The first one is how the distribution between sites that already share entities evolve. The second one is how to create new sites, or connect independent sites.

Sites that know each other. Sites that already share entities evolve by following which entities they share. They can share new entities by *transitivity*: a site obtains a reference to an entity via another entity that it already refer. For instance, if site a sends a value x on a port, and site b reads the message stream of that port, site b automatically has access to x.

Note that sites also connect to each other by transitivity: if sites a and b are connected together, and b and c as well, then a and c will automatically connect to each other, if the entities they share requires so. This depends on which protocols are used by the shared entities.

Sites can also reduce their set of shared references. This is handled by *distributed memory managment*. The system detects when a site no longer refers to a given entity, and can globally remove an entity from the distributed program. This always works, except for distributed reference cycles, i.e., reference cycles that involve several sites.

Connecting sites. The definitions we just gave of a distributed program suggests two ways of deploying an application over new sites: either by splitting a site into several sites, or by connecting distinct sites. Mozart uses the second approach, because it is easier to implement and to control in the program. This is provided by the module `Connection`.

The function call `{Connection.offer X}` returns a *ticket*, i.e., an atom that represents a reference to `X`. Conversely, the call `{Connection.take T}` returns the entity corresponding to the ticket `T`. Those functions allow a site to offer an entity reference to other sites via textual communication means. Indeed, as an atom is nothing more than a string of characters, it can be transmitted by e-mail, via a web site, or even told to the phone.

This mechanism is often used as a *bootstrapping* mechanism for distributing an application. The first entities that sites share are used to transmit other entities, by transitivity.

Managing resources. Not everything can be distributed. Assume a site b executes a procedure sent by site a, and that procedure has to save temporary data in a file. The site b may grant access to its file system, by it needs a way to provide this access to the running procedure. In order to solve this issue, Mozart has a module system based on *functors*. A functor is the specification of a module, with a list of modules to import, exported references, and code. If b receives a functor, it can instanciate it with a module manager that will provide (or possibly deny) the necessary local resources to the new module. Consult Mozart's documentation [Moz99] for more detail about functors.

3

Application structure and distribution behavior

With network transparency it is possible to take a program and distribute it, and it will run correctly. But it might be slow, for instance because its distribution involves much communication between sites. Here the programmer can take advantage of the network-aware aspects of the language to control the communication involved by the program's distribution. These aspects do not break transparency in the sense that the program is always a correct centralized program. So transparency gives two advantages. First, a centralized program is a correct distributed program. Second, tuning a centralized program for best distributed performance can be done by modifying the centralized program, e.g., with annotations that have no effect on centralized meaning. The program always retains a correct centralized semantics.

When tuning distribution, the fundamental distribution behavior is determined by the structure of the program. The latter defines the paradigms that are used: functional, dataflow, message passing, sequential or concurrent object-oriented, etc. The type of shared entities will determine communication patterns between sites. At a finer level, a given entity may be distributed in several ways, for instance stationary or replicated state, each having a specific distributed behavior. Choosing the most appropriate program structure is the way to make network transparency work.

3.1 Layered structure

We assume that a distributed application is structured in terms of *components*. We define a component as a program fragment with well-defined inputs and outputs. A component is itself defined in terms of simpler components. Exam-

ples of components are: a procedure, an object, several objects linked together. In this context, components can themselves be distributed. A distributed component can be decomposed into several components running on different sites, and communicating through shared language entities.

A component can be defined with a mixture of paradigms. Some of its subcomponent can be purely functional, while others use message passing and dataflow variables, for instance. The choice of paradigm is an advantage for reasoning about the program, since simpler components will require simpler reasoning techniques. For example, a declarative component handling lists will not be subject to race conditions, provided no other component concurrently binds its outputs.

The layered structure of the language encourages the programmer to pick the simplest programming paradigm to solve his or her problem. Concurrency with shared state is by far the most complex paradigm to program with. Most components do not need this expressive power. By limiting oneself to a part of the language, the programmer can take advantage of methodological support from the paradigm or the abstractions chosen. The network transparency ensures that this support is independent to whether the component is distributed.

The general advice is to keep the most general paradigm only for the components that need it, and to limit the extent of this paradigm inside the component itself. The usage of shared state concurrency is easier to manage when it is well confined in the program.

3.1.1 Using the declarative model

The simplest declarative components only provide stateless values, like pure functions without stateful dependencies. Dynamicity in declarative components are provided by shared logic variables. An example is a pipeline of components, where inputs and outputs are dataflow streams. Another example is several sites synchronizing on a given event. The sites only need to share one logic variable, and block until it is determined. The site notifying the event binds the variable to a conventional value, which automatically wakes up threads blocking on that variable.

Declarative components communicate by sharing values through logic variables. Communication is therefore purely dataflow and monotonic. From a distribution point of view, the programmer should pay attention to how data is shared among sites. Sharing stateless data is cheap in general, since it can be copied. But sharing too much data may imply much communication between sites.

Using the full power of the declarative model, one can also share lazy computations between components. What is shared is actually a logic variable. The by-need synchronization mechanism works through distribution.

Nondeterminism. Distributed declarative components are subject to nondeterminism in the sense that concurrent threads may bind variables in any

order. However, this nondeterminism is not *observable* from within the declarative model. If a declarative component terminates *without failing*, its outputs (defined by variable bindings) can be expressed as a mathematical function of its inputs. Their value do not depend on the execution order of the various threads in the component.

Note that if a failure occurs, for instance because of incompatible concurrent bindings, a part of the component will be failed state. From a strictly declarative point of view, the whole program has failed. But in the wider model, the failure does not automatically propagate to the whole program. If the rest of the program continues to run, then we have observable nondeterminism. But we are no longer in the declarative model.

3.1.2 Using message passing

When *observable* nondeterminism is required in a component, message passing is a good way to go. Ports provide a easy and efficient way to handle the nondeterminism in the component. The `Send` operation is asynchronous, and therefore it only requires a message to be sent by the system. The port's stream itself is monotonic, and behaves like a declarative component if it is distributed.

An effective use of this model is to let only one thread read the stream of messages, and treat them sequentially. This is the idea underlying *active objects*. An active object is a component that runs on only one site, and communicate with other active objects by sending messages.

Replying with variables. The model naturally offers two ways to reply to a given message. The simplest solution is to use the declarative model, and put a logic variable in the message. This logic variable will be bound to the reply. Snippet 3.1 on the following page shows two abstractions that implement this technique. The function `MakeServer` takes a function, and returns a port with a server. The server thread applies the function to each message, and binds the reply variable to the result of the call. The procedure `SendRecv` sends a message `X` to a port `P`, together with a reply variable `Y`. The code below creates a stateless server which adds 42 to each message it receives. Then the server is called with 54, with `Result` as the reply variable. As you can see, the functional notation allows to call `SendRecv` as a function.

```
Server={MakeServer fun {$ X} X+42 end}
Result={SendRecv Server 54}
```

Replying with continuations. A slightly more general technique is to put a *continuation* in the message, i.e., a procedure that is called by the receiver to reply the message. Note that the procedure is copied to the receiving site, so that it can be applied there. The continuation allows to program more sophisticated patterns of communication, like the *promise pipelining* provided in the language E [Mil06].

```
fun {MakeServer F}
   S in
   thread
      for X#Y in S do Y={F X} end
   end
   {NewPort S}
end

proc {SendRecv P X Y}
   {Send P X#Y}
end
```

Snippet 3.1: Abstractions to create and call a server with a reply variable

Snippet 3.2 on the next page defines a few abstractions that can be used to program in a "promise pipelining" style. The function `MakeServerC` creates a port with a server. The server thread applies a function `F` to each received message, and calls the continuation with the result. Let us create three servers `P`, `Q`, and `R` with that function. The server `P` replies either `Q` or `R`, depending on the message it receives. Servers `Q` and `R` expect an integer as message, and return the message after an arithmetic operation. Note that those servers should be created on different sites.

```
fun {F X} (if X==foo then Q else R end) end
P={MakeServerC F}
Q={MakeServerC fun {$ X} X+42 end}
R={MakeServerC fun {$ X} X div 2 end}
```

Now let us call `P`, thanks to the procedure `SendToPort`, with a continuation `C1` that is not determined yet. We will determine its value right after.

```
C1={SendToPort P foo}
%% is equivalent to: {Send P foo#C1}
```

The server will determine a result for the message, in this case `Q`, and eventually call `{C1 Q}`. During that time, the sender determines what `C1` does: it should send the message 54 to its argument. The fact that the continuation `C1` is called by server `P` makes that the message to `Q` is sent directly from `P`. There is no need to come back to the sender. The procedure `SendToCont` determines its first argument to do exactly that:

```
C2={SendToCont C1 54}
%% is equivalent to: proc {C1 Res} {Send Res 54#C2} end
```

The variable `C2` will thus be sent to `Q` as a continuation, so server `Q` will eventually call `{C2 96}`. We now define this continuation with the procedure `GetResultC`: `C2` binds its result to the variable `X`.

```
X={GetResultC C2}
%% is equivalent to: proc {C2 Res} X=Res end
```

```
fun {MakeServerC F}
   S in
   thread
      for X#Cont in S do {Cont {F X}} end
   end
   {NewPort S}
end

proc {SendToPort P Msg Cont}   % send Msg to port P
   {Send P Msg#Cont}
end
proc {SendToCont C Msg Cont}   % send Msg to promise C
   proc {C Res} {SendToPort Res Msg Cont} end
end
proc {GetResultC C X}          % get result from continuation C
   proc {C Res} X=Res end
end
```

Snippet 3.2: Promise pipelining with continuations

These continuations have defined the following pipeline: the client sends message `foo` with continuation `C1` to server `P`; then `P` sends message 54 with continuation `C2` to server `Q`; then `Q` sends its result 96 back to the client via the variable `X`. Note that this machinery relies on the fact that procedures are copied from site to site.

3.1.3 Using shared state concurrency

This is the most complex model from a programming point of view. And it is also the most demanding for the distribution. Shared state implies that read/write operations can be performed from multiple sites. Moreover, those operations are synchronous, and create many dynamic dependencies between sites. The difficulty stands in managing the state such that it is consistent: all the updates of a stateful entity must be serializable, as in a centralized multithreaded program.

Moreover, the shared state concurrency model is very sensitive to errors. This is because the threads follow an interleaving semantics. Consider the following example, where two threads perform each a read and a write operation on the cell `C`.

```
C={NewCell 0}
thread C:=@C+1 end     % thread A
thread C:=@C+2 end     % thread B
```

The result of the execution does not depend on where the threads and the cell are localized in the distributed system. When both threads terminate, the cell may contain either 1, 2, or 3. Some executions lead to surprizing behaviors. For

instance, the cell contents may decrease in this execution: thread A reads 0, then thread B reads 0, then thread B writes 2, then thread A writes 1.

Managing the nondeterminism in the programming language can be a problem. Using locks can help to create critical sections into the code, but they quickly lead to deadlock avoidance issues. In both the centralized and distributed cases, the programmer should use *transactions* to handle atomicity issues in his or her program. Transactions can be implemented quite efficiently in a centralized setting [ST95, VH04].

Implementing a distributed transaction system with good network behavior is not an easy task. Two systems have been proposed so far by our research team. The "GlobalStore" is fault-tolerant transactional replicated object store designed and implemented by Iliès Alouini and Mostafa Al-Metwally [AM03]. It takes advantage of replication to reduce latency when computing the transaction. Another transactional system was proposed to run on structured overlay networks, and was designed by the author and Valentin Mesaros [MCGV05]. The latter uses transaction priorities to avoid deadlocks, and the two-phase commit algorithm to ensure consistency between sites. Note that these systems only handle permanent failures.

3.2 Classification of language entities

The design of Oz is such that different entities may use different distribution protocols, and thus have different network behaviors. In fact, the distributed behavior of the whole program is determined by what type of entities are used, and how they are shared among sites. This section explores the different kinds of entities in Oz, and how they can be distributed.

Entities can be partitioned into three main categories: mutable, immutable, and monotonic. Each category has specific requirements that influences their possible distributed behavior. All the entities in a given category share the same set of distribution protocols. So the category of an entity determines what possible protocols it may use, and thus what possible network behavior it may have. We present each category, with the protocols available for each, and examples of Oz entities in those categories.

3.2.1 Mutable entities

This is the category of stateful entities in general. Those entities have an internal state, and the distribution must maintain a globally consistent view of the state. Here we sketch three possible distribution strategies for them.

- The simplest way to distribute a mutable entity is to make its state stationary. All operations are sent to the site holding the state, performed there, and a value can be returned.

- In the "mobile state" strategy, the state moves on the sites where operations are attempted. When the state is on a given site, it can be accessed locally. The state behaves like a cache, since several operations can be performed locally before the state leave the site.
- One can also replicate the state through sites. An update of the entity first invalidates all copies, then sends the new state. Reading the state is a pure local operation, and it can be done immediately if the copy is valid. This scheme is efficient when reading the state is more common than updating it.

Read and write operations are synchronous in general. Oz cells, objects, locks, dictionaries, threads belong to that category. Ports can be considered a special subcategory: the operation `Send` is an asynchronous update. The simplest strategy in this case is to leave the state stationary. To make an update it is enough to send a message to the site holding the state.

3.2.2 Monotonic entities

Those are also stateful, but their state is updated in a monotonic way. From a distribution point of view, they are more flexible that mutable entities. Their state can be replicated without the need to synchronize all the sites to perform an update. Single-assignment variables and streams are examples of monotonic entities.

Monotonic entities support the concurrent constraint operations *ask* and *tell* [Sar93]. The ask operation is just like a read. The tell operation updates the state of the entity. To ensure the consistency of the tell, all updates are serialized on one site. This site forwards the operation to all the other sites.

Single-assignment variables are *transients*, which is a subcategory of monotonic entities. Transients have a final state where they become another entity. The entity exists until is is bound. In Oz, logic variables have three states: free and not needed, free and needed, and determined. Note that streams are built from transient entities (read-only views), and therefore inherit from the distributed behavior of transients.

3.2.3 Immutable entities

Those are constants. One can only read them. They are usually copied eagerly or lazily between sites. When the entity has an identity, the protocol can guarantee that the copy is done once. This is useful when the value is large. In some cases the value cannot be copied, for instance because of implementation or security limitations. Read operations are then performed like in the mutable case.

Immutable entities range from simple values (atoms, numbers), to compound values (records), and even closures (procedures, classes). A compound value is copied with its fields, which may also be compound values or closures.

The copy of a compound value should have the same structure as its origin, including possible cyclic references and coreferences. Cycles and coreferences are detected when the value is serialized, so that they are not an issue for the programmer.

Closures are extremely powerful, because they contain both code and external references. The latter are handled just like records. And the code is copied between sites. The promise pipelining example in Snippet 3.2 on page 31 makes use of them, for instance. Note that cycles may also happen with closures, since closures can contain record references, which can contain references to closures. An again, the cycles are gracefully handled by the system, such that each entity in the reference graph of a given entity is copied at most once.

3.3 Annotations

The application is structured into components, and those components are themselves decomposed, layer by layer, up to primitive components, i.e., language entities. In its first implementation of distribution, Mozart was providing its distributed entities with fixed behavior for each. It was considered that those choices were expressive enough for the programmer to code the distributed behavior of his or her choice. Objects were distributed with mobile state. Stationary objects had to be reimplemented in Oz on top of ports, for instance.

We now let the programmer choose the distributed strategy for each language entity in his or her program. This choice is stated by *annotating* the entity. An application can be structured from top to bottom, with the annotation system providing the shaping of the distributed behavior at the lowest level of the structure. Annotations are part of the *network awareness* of the language, since they give some explicit control on an entity's distributed behavior.

Annotations may cover many facets of the distribution system. The first and most evident one is how the state of an entity is distributed, and the impact on primitive operations on that entity. Another facet is the distributed memory management of that entity. The system could also provide some robustness for its entities, and annotations may help to parameterize how robust an entity must be.

How to annotate. Conceptually an annotation is a bit like a declarative statement. It states something about an entity. It can even be thought as a constraint that the user posts about the distributed behavior of an entity. It is not fully declarative, since logic variables have a specific status. Annotating a variable is not equivalent to annotating its value.

In our proposal, which is explained in detail in the next section, annotating an entity is done by the statement

`{Annotate` *entity parameter*`}`

The nice property about annotations is that they can be ignored in case of no failure. They do not change the semantics of the program in that case. Moreover, if the program is not distributed at all, they will not be taken into account by the system.

3.3.1 Annotations and semantics

Annotations describe programmer choices for the distribution of an entity. Network transparency implies that the distribution must be an implementation, or a refinement, of the entity's semantics. The centralized semantics of an entity often admits several distributed semantics. Annotations give the possibility to the programmer to specify which distributed semantics should be used for a given entity. The semantics of the language is given in chapters 6 and 7. The latter also gives the semantics of annotations themselves.

The semantics of a language entity is thus partly reflected in the application. The annotation system let the latter make semantic choices for language entities *at runtime*. We could say that annotations allows the programmer to change the semantics of the language. But one can only *choose* between semantic variants, which are well defined, and do not break the centralized semantics of an entity in case of no failure.

Annotations are thus a limited form of *reflection* in the programming language. Its boundaries are defined by the programming system and by the centralized semantics. The programmer may tweak an entity's semantics within safe boundaries.

3.3.2 Annotation system

In practice an annotation specifies parameters of the distribution subsystem. The actual annotation system goes slightly beyond the conceptual level, because it allows implementation compromises that may break the semantics of an entity. The typical example is the time-lease based garbage collector, which may remove an entity from memory even if some sites in the application still refer to it.

The procedure `Annotate` is called to specify distribution parameters for an entity. We have chosen annotations to be *monotonic*: you cannot change your mind once you have chosen an option. Moreover, once an entity is actually distributed, i.e., when it has been shared by at least two sites, its distribution parameters can no longer be changed. As a consequence, an entity can only be annotated *before* it gets distributed. It is therefore useful to annotate entities into the abstractions that create them.

Distribution parameters are specified as atoms or records. For instance, stationary state is specified with the atom `stationary`, and the use of a mobile access reference is specified by the record `access(migratory)`. Several parameters are combined in a list, like in the example

```
{Annotate E [stationary access(migratory) lease]}
```

We will see in the next section that this statement is equivalent to the following three ones.

```
{Annotate E stationary}
{Annotate E access(migratory)}
{Annotate E lease}
```

3.3.3 Partial and default annotations

Our annotation system is not only monotonic, it is even incremental. Several annotations can be put on a given entity, at different times. The result is that the entity is annotated by the conjunction of them, provided that it is consistent. For instance, mobile and stationary state are inconsistent together, but stationary state and time-lease garbage collection are consistent, because both parameters are orthogonal to each other. In our implementation, Mozart considers three orthogonal distribution parameters, namely the access architecture, the state protocol, and the distributed garbage collection. Those are described in more detail in the next sections.

The system also permits *partial* annotations. It means that some distribution parameters may be left unspecified whenever an entity becomes distributed. In that case, the system *completes* the annotation with default values. For instance, if the programmer annotates a cell with `lease` (time-lease based garbage collection), the system may complete the annotation to

```
[migratory access(stationary) lease]
```

right before distributing the cell.

Each type of entity has a default annotation, giving a value for each distribution parameter. The default annotation must be complete, of course. The system implementation may or may not allow the programmer to modify default annotations. In our prototype, default annotations can be modified by the program at any time.

3.3.4 Access architecture

The access architecture of an entity defines how all the sites sharing the entity coordinate with each other. This architecture is the base of the other protocols (state and reference consistency, see below). It could be anything, as long as one can implement the entity operations and garbage collection on top of it.

In Mozart, all sites sharing the entity own a *proxy*, and all those proxies refer to a unique *coordinator*, which is hosted by one of the sites. It is used by the other protocols as a reference point. When an entity reference is sent from one site to another, a network address of its coordinator is given. This allows the receiver to connect its proxy to the other proxies of the same entity in the whole system. The architecture is similar to a client-server architecture.

Knowing the type of access architecture already gives some information about the network behavior of the entity. For instance, an entity with stationary state will have its state located on the same site as the entity's coordinator. Another example is garbage collection, where the coordinator determines whether the entity is referred to by remote sites. If not, the entity is no longer distributed.

Mozart/DSS defines one parameter for the access architecture, which states whether the coordinator is stationary or mobile.

- `access(stationary)`: the coordinator is located where the entity was created, and remains on that site. This is the simplest strategy to manage the access architecture.

- `access(migratory)`: the coordinator can be moved from one site to another by a specific operation. There are several possibilities on how proxies can find where it is. Details can be found [Kli05].

A single point of failure. The coordinator of an entity is obviously a single point of failure. When it crashes, the entity's proxies are usually no longer capable of finding each other. Note however that we have a single point of failure *per entity*. This design decision is motivated by the fact that entity protocols should not solve all the problems. Entities are generally not robust to failures. Instead, failures are detected, and can be handled at the language level. Fault-tolerant protocols can be implemented in Oz, and hide failures by abstractions.

3.3.5 State consistency protocols

Those protocols implement the operations of the entity itself. The choice of protocol depends on the entity's nature, i.e., mutable, immutable, or monotonic. This parameter is the most important when considering the network behavior of entity operations. Here are the protocol annotations considered in Mozart/DSS.

1. Mutable entities.

   - `stationary`: the state of the entity is located on the site of the entity's coordinator. All operations on the entity's state are performed on this site. Synchronous operations therefore need a full round-trip from the requesting site to the coordinator to complete.

   - `migratory` and `pilgrim`: the state of the entity migrates from one site to another, and a site executes operations locally when the state is on that site. The state behaves like a cache: once the state is on a site, that site may perform several operations without extra network overhead.

- `replicated`: the state of the entity is copied on all the sites that use the entity, and the copies are synchronized by a two-phase commit protocol. This protocol is useful for data structures that are rarely updated. Read operations can be performed locally, while write operations require an atomic update of all sites using the entity.

2. Monotonic entities.

   - `variable`: this corresponds to the protocol described in [HVB+99]. The binding of the variable is performed on the site of the entity's coordinator.
   - `reply`: this is a variant of the `variable` protocol, where the binding is done on the first site that receives the reference. This protocol has the best network behavior when the receiver site attempts to bind the variable. The variable is typically used to a reply to a query.

3. Immutable entities.

   - `immediate`: the value is sent together with the reference of the entity. The unicity of the entity is guaranteed, even if its value is sent multiple times. All values with structural equality (numbers, atoms, records) use this protocol.
   - `eager` and `lazy`: those protocols guarantee that the value is sent at most once. When the entity is sent to a site, only its reference is actually sent. The receiving site requests the entity's value if it does not have it yet. In the `eager` case, the value is requested upon receipt, while in the `lazy` case, it is requested once the value is actually needed.
   - `stationary`: the value is not copied on other sites. Remote operations require a full round-trip to the coordinator. For instance, one can provide access to a chunk without allowing copies on possibly untrusted sites.

3.3.6 Reference consistency protocols

Those protocols ensure the reference integrity, by implementing a distributed garbage collector. Here the choice is not exclusive: one can combine several protocols in a single annotation. An entity is kept in memory if all protocols require so. Mozart proposes three algorithms:

- `persistent` simply keeps the entity alive forever on its coordinator. The entity is simply never removed by the garbage collector. This can be useful for providing a service on a site, which should run until the site terminates.

- `refcount` uses a weighted reference counting scheme. Each reference to the entity is assigned a weight, and the entity's coordinator keeps track of the sum of the weights of all remote references. When this number reaches zero, the entity is no longer distributed. The advantage of using weighted references is that new remote references can be created without notifying the coordinator. It suffices to keep the total weight constant.

- `lease` uses time-lease based mechanism. Sites holding a remote reference to an entity regularly notify its presence to the coordinator of the entity. The time between successive notifications is called the lease period. If the coordinator has not been notified during a long time (typically much longer than the lease), the entity is no longer considered distributed.

The algorithm `refcount` guarantees consistency, i.e., a coordinator remains alive while its proxies are, even in case of long network failures, but is not robust to site failures. The algorithm `lease` does not guarantee consistency in case of network delays, but handles site failures gracefully. It is up to the programmer to choose what fits best for his or her application.

3.4 Related work

How do other systems provide tuning of a distributed program? Are they easy to tune at all? Can the tuning process be programmed in the language, or is it external to it? In the latter case, are the tuning techniques heavily modifying the program?

3.4.1 Erlang

In the Erlang philosophy, everything is a *process*, and the only communication primitive between processes is message passing with values. Processes are independent of each other, and cannot share memory. Every process is sequential and programmed in functional style. This simple model fits pretty well with distribution, and allows efficient implementations.

Process identifiers can be sent between sites, and sending messages to a remote process is transparent. Processes do not migrate between sites, and garbage collection is up to the programmer. The language favors lightweight client-server style. For instance, the Open Telecom Platform (OTP) provides generic server abstractions, which support transactional semantics (crashed servers are restarted with a valid former state) and code swapping (the server code can be changed on-the-fly). In fact, the OTP provides many more abstractions to build large-scale, fault-tolerant, distributed applications [Arm07].

With the simplicity of Erlang's programming model, any communication pattern can be programmed with processes and messages. Libraries like the OTP already provide powerful abstractions for distributing applications. Of course, the programmer should always use this kind of abstraction to build

his or her applications. Changing the network behavior can then be done in a modular way. Using directly the distribution facilities makes the program harder to adapt.

3.4.2 Java RMI

There is no distribution mechanism defined by the language Java itself. But the Java Remote Method Invocation (RMI) library has quickly become the most popular distribution mechanism in the Java community. The library provides two ways to distribute an entity: full serialization and remote method invocation. In the first case, a copy of the object is made once a reference to that object is sent to a site. In the second case, only a reference to the original object is created on the receiving site. When that reference is invoked, the method invocation is sent to the object's site and the calling thread waits for its termination. These objects are said to be *remote*, while fully serialized objects are *non-remote*.

This mechanism imposes synchronous interaction between sites, which can be slow in a distributed setting. But worse, reference integrity is only guaranteed per method invocation, and not in general [Sun97]. The consequence is important: distributed objects have a *different semantics* than centralized objects.

Besides these semantic issues, Java RMI is not transparent to the programmer. Remote objects must implement the interface `java.rmi.Remote`, while non-remote objects implement `java.io.Serializable`. Turning a local object in a distributed one requires to modify its class. The library provides a few abstractions to write servers, though.

3.4.3 E

The language E is an object-oriented programming language designed for secure distributed computing. It was created by Mark S. Miller, Dan Bornstein, and others at Electric Communities in 1997. It combines capabilities and a message passing concurrency model with Java-like syntax. Its concurrency model is based on event loops and promises, in order to prevent deadlocks. More on security aspects of E can be found in Mark Miller's thesis [Mil06].

Objects behave like concurrent sequential agents with synchronous or asynchronous method invocation. Each object is stored into a *vat*, which is the unit of localization. Synchronous invocation can only happen between objects in the same vat, and corresponds to a sequential method call.

```
def result := bob.foo(carol)     /* synchronous call */
println(‘done: $result‘)
```

A method is always executed atomically, and should never block or run forever. This strong limitation to concurrency was chosen to avoid common programming errors due to shared state concurrency.

Objects can also invoke each other asynchronously, with the *eventually* or *send* operator `<-` (see below). If the method returns a result, the operation returns immediately a promise for the result.

```
def result := bob <- foo(carol)     /* eventual send */
when (result) -> {                  /* promise resolution */
   println(‘done: $result‘)
} catch problem {
   println(‘oops: $problem‘)
}
```

One can send to the promise immediately; a promise pipelining mechanism ensures that the resulting object is eventually sent the method (like in Section 3.1.2 on page 29). One can also synchronize on the result with a `when` statement. Once the promise is resolved, the code is eventually run (atomically). The `catch` part allows to handle a promise failure.

The distributed model of E is strongly determined by the security aspect of the language. By default, vats do not trust each other. Therefore objects are never copied or moved between vats. Vats are strongly isolated from each other, and inter-vat communications are encrypted. The encryption also guarantees that object references cannot leak into intermediate vats in the promise pipelining process. The distribution model is thus limited to client-server communication with promises, but the capability system is guaranteed safe.

4

Asynchronous failure handling

We go one step further in the distribution support by reflecting partial failures in the programming language. We propose a language-level fault model that is compatible with network transparency. Because a site or network failure may affect the proper functioning of a distributed entity, our model defines how entities can fail, and how those failures are reflected at the entity level in the language. Here are the principles of the model, each being described in the corresponding subsection below.

- Each site assigns a *local fault state* to each entity, which reflects the site's knowledge about the entity.
- There is no synchronous failure handler. A thread attempting to use a failed entity blocks until the failure possibly goes away. In particular, no exception is raised because of the failure.
- Each site provides a *fault stream* for each entity, which reifies the history of fault states of that entity. Asynchronous failure handlers are programmed with this stream.
- Some fault states can be enforced by the user. In particular, a program may explicitly provoke the global failure of an entity.

This fault model is an evolution of the first fault model of Oz, and integrates parts of another proposal made by Donatien Grolaux *et al.* [GGV04]. The next chapter will demonstrate that it improves the ease of programming and modularity of failure handlers. A comparison with the other fault models of Oz is given in Section 4.5 on page 56.

4.1 Fault model

We first provide a precise description of the kind of faults we consider in the system in which the program runs. This system is composed of sites that communicate through a network. We model a certain number of failures in that system, how they affect the system, and how they can be detected. Failures affect language entities, so we also consider failures at the entity level. The model we use here is inspired from Rachid Guerraoui *et al.* [GR06], and is quite standard in the field of distributed systems.

Note that we will sometimes use the term *process*. A process is simply something that has some autonomous behavior, some internal state, and which interacts with other processes by exchanging messages through the network. Both sites and language entities can be considered as processes.

4.1.1 Failures

Site failures. A site may fail by *crashing*, i.e., at a given time t, it stops doing anything, especially sending and receiving network messages. There is no recovery mechanism by default, the failure is permanent. This kind of failure is called *crash-stop* or *fail-stop*. A site that has not crashed yet is said to be *correct*.

We assume that sites are subject to neither *omission* (where the process may "miss" some messages), nor *Byzantine faults* (where the site may perform any arbitrary action). Those are much harder to handle, and detect. The lack of generic detection mechanism makes any kind of language support for them virtually impossible.

Network failures. The network is considered *reliable* in the sense that a message sent by a site a to a site b will eventually be delivered, unless a or b crashes. Messages are never corrupted nor delivered more than once. However, the communication between two sites may take arbitrary time. The communication link may appear to have failed if no message is delivered to the destination site during a long but finite period of time. In other words, network failures can be defined as communication delays that are longer than expected. We do not consider the case where network failures would be permanent. Such a failure would mean that a site can no longer communicate with any other site. By convention, network failures are always considered to be temporary.

Entity failures. Language entities are subject to the same kind of failures as sites. A failed entity stops being functional. No language operation can have an effect on it. Its state is lost, and there is no recovery mechanism. The failure is permanent and global: a failed entity is unusable for all sites.

We also consider a failure of type fail-stop, but which is valid for a given site: the entity is crashed for that site, but may be functional for other sites.

In other words, that site can no longer use the entity. We say that the entity is *locally failed* on the given site. This failure is triggered by the operation `Break`, which is described in Section 4.3.2. It is typically used to prevent the site from using the entity, and does not affect the other sites.

4.1.2 Failure detectors

In order to handle failures at the program level, we need *failure detectors*. A failure detector is a component that tries to determine whether a given process has crashed. It notifies the program when it *suspects* the process to have crashed. Not all failure detectors are identical, there exists several types of them. We can classify them according to three properties: their completeness, accuracy, and monotonicity.

- A failure detector is *complete* if a crashed process is always eventually suspected.
- A failure detector is *accurate* if a suspect process has actually crashed, and *inaccurate* if it may suspect a correct process. It is *eventually accurate* if no correct process is suspected forever.
- A failure detector is *monotonic* if a suspect process is never notified correct later. In other words, the failure detector never "changes its mind."

The completeness is a liveness property, it ensures the eventual detection of a crash. On the other hand, the accuracy is a safety: it prevents erroneous suspicions. Those two properties are essential for reasoning about the correctness of an algorithm that handle failures. The monotonicity also helps for reasoning, because monotonic detectors have a simpler behavior than nonmonotonic ones.

A failure detector is *perfect* if it is complete and accurate. It is *eventually perfect* if it is complete and eventually accurate. Perfect failure detectors are not so common, because they require strong properties of the underlying system. For instance, perfect detectors are possible on a local area network (LAN), but not on the internet in general. For the internet, one has to use eventually perfect detectors.

Three simple failure detectors. Our model proposes a combination of three failure detectors, namely *tempFail*, *permFail*, and *localFail*. Each detector has its own properties in terms of completeness, accuracy, and monotonicity. We will show in the next sections how we use them to handle language entity failures.

- The *tempFail* detector is eventually perfect. It uses two notifications: `tempFail` and `ok`. The first one occurs when the target process is suspected, the second one occurs when the detector has found evidence of correctness of the process. This detector is nonmonotonic.

detector	complete	accurate	monotonic
tempFail	yes	eventually	no
localFail	no	no	yes
permFail	no	yes	yes

Table 4.1: Summary of the properties of the three failure detectors (for global failures)

- The *permFail* detector is accurate but incomplete. It is not guaranteed to detect a crash, but it never erroneously reports a crash. It uses the notification `permFail`. By definition it is monotonic.
- The *localFail* detector is perfect for local failures, and uses the notification `localFail`. However it is neither complete, nor accurate for global failures in general. Its completeness and accuracy depend entirely on the program. However it is monotonic: suspicion remains forever.

A summary of the properties of the three failure detectors is shown in Table 4.1. Note that these properties are given with respect to global failures. This is why *localFail* is neither complete nor accurate.

4.1.3 Entity fault states

For each entity e, every site has a failure detector that combines the three failure detectors *tempFail*, *permFail*, and *localFail*. That failure detector maintains a *local fault state*, which is like a *view* of the actual fault state of the entity. The failure detector sends a notification at every state transition. The notification mechanism is described in Section 4.2.3.

The failure detector has four states, called *local fault states*, or *fault views*: `ok`, `tempFail`, `localFail`, and `permFail`. Valid state transitions are depicted in Figure 4.1 on the facing page. The semantics of the states are the following.

- `ok` is the initial state, and can also be triggered by the *tempFail* detector. It means that the entity is not suspected by any of the basic failure detectors.
- `tempFail` is triggered by the *tempFail* detector. It means that the site is temporarily unable to complete any operation on the entity. This typically happens when this site cannot communicate with other sites that are necessary for performing language operations on the entity.
- `localFail` is triggered by the *localFail* detector. It means that the entity is permanently unavailable for this site. Note that it is local, i.e., other sites may still have access to the entity. This state can be enforced by the program.

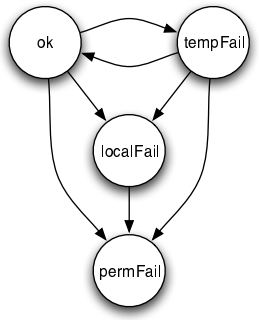


Figure 4.1: Local fault state diagram of an entity

- `permFail` is triggered by the *permFail* detector. It means that the entity has crashed. No site can ever perform an operation on it. This state is final.

The main advantage of this model is that it provides a simple yet precise description of an entity's state, from the viewpoint of one site. It abstracts the type of hardware and system used, the protocols, and even the kind of entity it applies to. It describes the failure from the programming language's point of view. Yet its simplicity still allows to reason about the partial failures in a program.

4.1.4 Concrete interpretation of fault states

Knowing the kind of an entity and its distribution strategy, one can easily give a more precise interpretation of a fault state. Here we give the various concrete reasons for an entity to fail. The only concepts we rely on are the ones given in Chapter 3. All failures can be expressed in terms of sites, protocols, coordinators, and memory management.

Note that the interpretation we provide for fault states is of course related to how the distribution of an entity is implemented. A sophisticated implementation of distribution would have led to a complex fault model. In our work we favor a simple fault model, therefore keeping the implementation simple. The programmer should be able to reason easily about the properties of the distribution. Complex fault-tolerant abstractions should be built at the higher user level, not at the low level.

Mutable entities. Two sites are usually involved when reasoning about mutable entity failures: the coordinator site and the site holding the state. The coordinator is necessary in all protocols to manage the entity's state. It is the site holder if the state is stationary; it manages to bring the state to the requester in case of a mobile state; and it ensures mutual exclusion when the

state is replicated. The state holder is also crucial: its failure always implies that the state is lost.

A mutable entity is in state `tempFail` if the coordinator or state holder is unreachable. The state `localFail` is triggered by the program. The state `permFail` is reached when the coordinator or the site holding the state has crashed, or the coordinator has removed the entity from its memory. The coordinator crash can be provoked by the program (see Section 4.3).

The second reason for the state `permFail` has already been mentioned in Chapter 3: time-lease based garbage collection is not correct in case of network failures. The coordinator considers that the entity is no longer used when no other site has showed interest for a certain duration. A problem arises when a site cannot reach the coordinator because of a network problem. In that case, the entity will fail on that site. The good property is that it is diagnosed properly, and reflected in the language. If the network recovers and the site can reach the coordinator again, the removal of the entity will be notified, and result in the entity failure.

Monotonic entities. Transients are pretty similar to mutable entities when it comes to failures. In the protocol `variable`, the coordinator is also a state holder. The state holder might be different in the `reply` protocol. If only one site refers to the variable besides the coordinator, then that site is the state holder. The same reasoning as with mutable entities applies here.

A property of logic variables is that they conceptually disappear once they are bound. In fact, bound variables have reached their final state, and become invisible to the program. For the sake of consistency, bound variables do not fail, and failed variables remain unbound.

Immutable entities. Immutable entities are simply values. Their possible fault state depend on whether they are copied between sites (protocols `immediate`, `eager`, `lazy`) or not (protocol `stationary`). Note that entities using the `immediate` protocol never fail, since one cannot have a reference to the entity without having its state. As all entities with structural equality (numbers, atoms, records) use this protocol, they are not subject to failure.

Values cannot fail permanently if they are copied between sites. If the site from where the copy is made is unreachable or has crashed, a temporary failure will be notified. The local fault state can even be `localFail`. But the state `permFail` should never be observed, because any other site may provide a copy of the value. The fault state `permFail` requires that no copy of the value is available anywhere, even in a file. This property is difficult to verify in practice.

Values distributed with a stationary state are different. This protocol can be used when copying the whole value is too costly or insecure. Remote sites can still access the value, typically with the dot operation "`.`". In that case, the causes of failure are the same as mutable entities with stationary state.

4.2 Failure handlers

We now discuss the possible ways to handle entity failures in the language. We make a clear distinction between two basic ways of handling failures, namely *synchronous* and *asynchronous* handlers. As we shall see, asynchronous failure handling is preferable to synchronous failure handling.

4.2.1 Definition

A *synchronous* failure handler is executed in place of a statement that attempts to perform an operation on a failed entity. In other words, the failure handling of an entity is synchronized with the use of that entity in the program. Raising an exception is one possibility: the failure handler simply raises an exception. In contrast, an *asynchronous* failure handler is triggered by a change in the fault state of the entity. The handler is executed in its own thread. One could call it a "failure listener". It is up to the programmer to synchronize with the rest of the program, if that is required.

The following rules give small step semantics for both kinds of handlers. The symbol σ represents the store, i.e., the memory of the program. The store is partitioned among the sites, and the elements of the store that are specific to a site a are subscripted by a. Each site a reflects its view of the fault state of an entity in the store through a system-defined function $\mathsf{fstate}_a(x)$, which gives the local fault state of x. Each execution rule shows on its left side a statement and the store before execution, and on the right side the result of one execution step.

Rule (4.1) describes the semantics of a synchronous failure handler. It states that a statement S attempting an operation on entity x can be replaced by a handler H if the fault state of entity x is not ok, i.e., if x has failed.

$$\frac{S_a \;\|\; H_a}{\sigma \;\|\; \sigma} \quad \text{if} \quad \begin{array}{l}\text{statement } S \text{ uses entity } x \\ \text{and } \sigma \models \mathsf{fstate}_a(x) \neq \mathsf{ok}\end{array} \tag{4.1}$$

Rule (4.2) gives the semantics of an asynchronous failure handler. A new thread is spawned with handler H whenever the fault state of x changes. Note that there may be more than one handler on x; we assume all handlers are run when the fault state changes.

$$\frac{\qquad\qquad \;\|\; H_a}{\sigma \wedge \mathsf{fstate}_a(x){=}fs \;\|\; \sigma \wedge \mathsf{fstate}_a(x){=}fs'} \quad \text{if} \quad fs \rightarrow fs' \text{ is valid} \tag{4.2}$$

4.2.2 No synchronous handlers for Oz

In Oz, when the fault state of a given entity is not ok, operations on that entity may not succeed. Raising an exception in that case might look reasonable, but our experience suggested that it is not. Because of the highly concurrent nature of the language, raising exceptions quickly creates race conditions between

threads. The functional code is cluttered with failure handling code. Other kinds of handlers have been tried, but without success.

We have chosen to use asynchronous failure handlers only. We propose the following model.

> "Failure causes blocking": an operation on a failed entity simply *blocks* until the entity's fault state becomes `ok` again.

The operation naturally resumes if the failure proves to be temporary. It suspends forever if the failure is permanent (`localFail` or `permFail`). With this model, nothing extra can happen in a program that does not handle distribution failures.

4.2.3 Entity fault stream

In our proposal, asynchronous failure handlers are programmed as threads that monitor entities, and take action when an entity changes its local fault state. On every site, each entity is associated with a *fault stream*, which reflects the history of the fault state's view of the entity. The system maintains the *current* fault stream, which is a list $fs|s$, where fs is the current view of the fault state, and s is an unbound variable. It is defined semantically as a system-defined function $\mathsf{fstream}_a(x)$ that returns the current fault stream of the entity x on site a. The semantic rule

$$\frac{}{\sigma \wedge \mathsf{fstream}_a(x){=}fs|s \;\Big\|\; \sigma \wedge \mathsf{fstream}_a(x){=}s \wedge s{=}fs'|s'} \quad \text{if} \quad fs \to fs' \text{ is valid} \tag{4.3}$$

reflects how the system updates the fault state to fs'. The dataflow synchronization mechanism wakes up every thread blocked on s, which is bound to $fs'|s'$. An asynchronous handler can thus observe the new fault state simply by reading the elements of the fault stream.

To get access to the fault stream of an entity x, a thread simply calls the function `GetFaultStream` with x, which returns the fault stream of x on the current site. A formal definition is given below. To read the current fault state, one simply takes the first element of the returned list.

$$\frac{(y{=}\{\texttt{GetFaultStream}\ x\})_a}{\sigma} \;\Big\|\; \frac{(y{=}fs|s)_a}{\sigma} \quad \text{if} \quad \sigma \models \mathsf{fstream}_a(x){=}fs|s \tag{4.4}$$

Figure 4.2 on the next page shows an example of how an entity's fault stream may evolve over time. The stream is a partially known list, and the underscore "_" denotes an anonymous logic variable. In the last step, the stream is closed with `nil`. This may happen in two situations, which are explained below. Snippet 4.1 on the facing page shows a thread monitoring an entity `E`, and printing a message for each fault state appearing on the stream. The printed message is chosen by pattern matching. The thread is woken up each time the stream is extended with a new state.

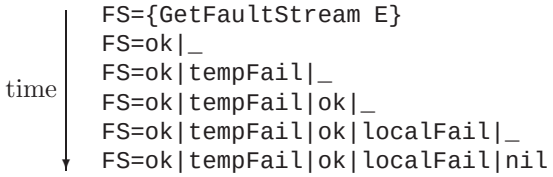


Figure 4.2: An example of a fault stream evolving over time

```
thread
   for S in {GetFaultStream E} do
      T = case S                          % pattern matching on S
          of ok        then "entity is fine"
          [] tempFail  then "some problem, don´t know"
          [] localFail then "no longer usable locally"
          [] permFail  then "no longer usable globally"
          end
   in
      {Show T}
   end
end
```

Snippet 4.1: A thread that prints messages when entity `E`'s fault state changes

Special case: variables. Monitoring variables requires a bit more care than other entities. This is because variables conceptually disappear once they are bound: they *become* what they are bound to. The question is: what happens to the fault stream of a variable once the latter is bound? There are two distinct cases to consider, as the variable is bound to either a value, or another variable.

Consider a variable x that is bound to another variable y. From a programmer's point of view, the binding is transparent: x remains a variable. For a thread monitoring x, it seems quite natural to smoothly switch to monitoring y. We propose to make this transition automatic by *merging* the fault streams of x and y. Basically the tail of the fault stream of x is bound to the tail of fault stream of y, prepended by y's current fault state if it is different from x's current fault state. This binding makes sure that the monitor does not miss a fault state.

The other case we have to consider is the binding of the variable to a value. We think that merging the fault streams is not a good idea here, because the entities are of different nature, and this may lead to confusion. However, we need a clear mechanism to notify the monitoring thread that the variable has been bound. We propose to *close* the fault stream by binding its tail to `nil`, because the variable has conceptually disappeared. Once this happens, calling `GetFaultStream` on the variable will return the fault stream of its value.

Failure history. The fault stream of an entity e on a site a reifies the history of fault state observations of e by a. Moreover it transforms the nonmonotonic changes of a fault state into monotonic changes in a stream. It provides an almost declarative interface to the fault state maintained by the system. This interface looks much simpler and more elegant than registered handlers, which is what Mozart used before [VHB99]. In particular, the fault stream guarantees that the failure handler cannot miss a state transition.

Note that the fault stream may also be closed, i.e., its tail bound to `nil`, whenever it is no longer maintained by the system. This is performed by the system when the entity is no longer in memory. See Section 4.4.

4.2.4 Discussion

Synchronous failure handlers are natural in single-threaded programs because they follow the structure of the program. Exceptions are handy in this case because the failures can be handled at the right level of abstraction. But the failure modes can become very complex in a highly concurrent application. Such applications are common in Oz and they are becoming more common in other languages as well. Because of the various kinds of entities and distribution protocols, there are many more interaction patterns than the usual client-server scheme. Handlers for a given entity may run in many threads at the same time, and those threads must be coordinated to recover from the failure.

All this conspires to make fault tolerance complicated to program if based on synchronous failure handling. This mechanism was in fact never used by Oz programmers developing robust distributed applications [GGV04]. Instead, programmers relied on the asynchronous handler mechanism to implement fault-tolerant abstractions. One such abstraction is the "GlobalStore", a fault-tolerant transactional replicated object store designed and implemented by Iliès Alouini and Mostafa Al-Metwally [AM03].

4.3 Making entities fail

Failures in distributed systems are often partial. This will be the case with entity failures in a distributed application, especially if the programmer defines components that are spread among many sites. In many cases, if a subset of the entities of a component have failed, the component itself might no longer function. The components that use the failed component must be able to detect the failure, and trigger a recovery mechanism. The question is: which entity should they monitor?

A component should not monitor all entities of another component explicitly. This would prevent any encapsulation in the monitored component. But monitoring the entities it has access to might not be enough, if none of the monitored entities fails. One possibility is to design a component-level protocol that makes sites consider the component as failed. Another possibility is to

```
proc {SyncFail Es}
   Trigger in
   for E in Es do
      thread
         if {List.member permFail {GetFaultStream E}} then
            Trigger=unit
         end
      end
   end
   thread
      {Wait Trigger}
      for E in Es do {Kill E} end
   end
end
```

Snippet 4.2: Synchronize the failure of a set of entities

make the monitored entities fail on purpose. We propose to provide support for the second alternative in our failure model, i.e., the program can make an entity fail.

4.3.1 Global failure

We provide a new operation to make an entity fail. The statement `{Kill` e`}` attempts to make the entity e permanently failed, i.e., in fault state `permFail`. The operation is asynchronous, which means that it returns immediately, and is idempotent. It initiates a protocol that tries to make the entity globally failed. Once it is done, the local fault state of e becomes `permFail`. Because of the definition of the state `permFail`, the operation may require some synchronization with other sites that refer to e. The operation must ensure that no other site can perform operations on the entity. Therefore the operation `Kill` is not guaranteed to succeed.

The example in Snippet 4.2 shows a simple abstraction, yet quite powerful. It basically tries to ensure that all entities in a list eventually fail when one of them fails.

4.3.2 Local failure

Sometimes it is not possible to make an entity fail globally, for instance because a site that is involved in the operation `Kill` has silently crashed. We therefore provide the operation `Break`. The statement `{Break` e`}` has a pure local effect. It makes the entity e fail locally, and forces its fault state to be at least `localFail`.

The first motivation for `Break` is that it is irreversible. Once an entity is permanently failed, even locally, it cannot go back to the fault state `ok`. This is useful when a site triggers a recovery mechanism, based on the state `tempFail`

```
proc {FailAfter E TimeOut}
   proc {Loop L}
      case L of H|T then
         if H==tempFail andthen {WaitTwo {Alarm TimeOut} T}==1
         then {Break E}
         else {Loop T}
         end
      else skip end
   end
in
   thread {Loop {GetFaultStream E}} end
end
```

Snippet 4.3: A failure handler that provokes local failure after a certain duration of temporary failure

of an entity e. Enforcing the failure of e simplifies the task of recovering. The threads blocked because of the failure of e will never wake up, for instance. This is useful if a service is backed up, and at most one instance of the service can run at any given time.

The second motivation is resource management. By making an entity permanently failed, the programmer gives a hook to its memory management system. Threads that block because of the failure will block forever, unless they can be woken up explicitly by other threads. The system can therefore use the permanence of the failure to detect parts of the program (threads and data) that will no longer affect its behavior. Those parts can be safely removed from memory. Some issues about memory management are described in detail in the next section.

Snippet 4.3 shows a small failure handler that can be used together with other failure handlers. Basically it uses a timeout to make an entity locally failed if it remains temporarily failed for a certain time. The timeout duration is specified in the parameter `TimeOut`. Other failure handlers waiting for state `localFail` are thus automatically triggered after the given inactivity duration.

4.4 Failures and memory management

4.4.1 Blocked threads and fault streams

Entity failures have an effect on the memory management of a program. First, a failed entity can make a thread block. If the failure is temporary, that thread must be kept in memory for its possible resumption. As that thread normally refers to the entity, it keeps the entity alive. However, if the failure is permanent, the thread will block forever, unless it is referred to by another living entity. As we already mentioned in Section 4.3.2, a thread blocking

forever can be safely removed from memory.

Something similar happens with fault streams. An entity keeps its own fault stream alive in memory. This guarantees that the threads monitoring the entity do not silently disappear. But the fault stream itself does not keep the entity alive, so the entity can be removed from memory anyway. Once the entity is removed from memory, the fault stream will no longer be kept alive, and the monitoring threads may block forever. In order to clearly reflect that the fault stream has been "disconnected" from the entity, we make the system *close* the stream, i.e., its tail is bound to `nil`. This action is perfectly consistent, since once the entity is gone, the fault stream will no longer be updated.

Finalization. The closing of the fault stream provides a simple and effective *post-mortem finalization* mechanism. The following abstraction executes a procedure `P` once the entity `E` is no longer in memory. The closing of the stream simply lets the loop exit.

```
proc {Finalize E P}
   thread
      for X in {GetFaultStream E} do skip end
      {P}
   end
end
```

This is particularly useful to recollect memory from failed components in a program. A thread monitoring an entity can already remove references to the entity when it fails, and once it is removed from memory, the monitor can perform some extra actions.

4.4.2 Entity resurrection

It is possible for a site to remove an entity e from its memory, even when that entity is still used by other sites. Indeed, if the site owns a proxy for e that is not necessary for the distribution of the entity, it can safely remove the entity's proxy from its memory (see Sections 3.3.4 and 3.3.6). When this happens, the site simply no longer refers to e, which remains alive on a global scale.

Now assume that the entity e was removed from the memory of site a, and that a reference to e is sent again to that site. A new proxy for e is created on a, and that proxy creates a new fault stream for e on a. This reintroduction of e on site a brings a few issues. First, there is no connection between the new fault stream of e and its former fault stream, which was bound to `nil` by the finalization mechanism described above. The instances of the fault stream in memory correspond to different *sessions* of the entity on the site.

Second, it is possible that the fault state of e was `localFail` before e's removal, and `ok` after its reintroduction. This state transition is normally forbidden by the fault model. To avoid this situation, site a should have kept some information about entity e in memory. But keeping that information in memory is unreasonable in general, because site a's memory would grow beyond

any limit. Our proposal is to *not* keep that information, but to implement a specific solution to handle the issue at the application level. An application may use a centralized or distributed repository of valid entities. Any occurrence of a non valid entity can then be discarded. The management of the repository and the choice of which entities to check is thus specific to the application.

4.5 Related work

4.5.1 Java RMI

In Java Remote Method Invocation (RMI), every distributed operation is a method call. The standard way in that language to report a problem inside a method call is the exception mechanism. The fault model thus favors synchronous failure handlers, which are implemented as exception handlers.

4.5.2 Erlang

The power and simplicity of failure handling in Erlang was an inspiration for our work. Erlang provides asynchronous detection of permanent failures between processes [Arm07]. Two processes can be *linked* together. When one of them (say a) terminates normally or because of a failure, the other one (say b) is notified by the runtime system. By default, process b will die if a died because of a failure. However, if b is a *system process*, it will receive a message of the form `{'EXIT',Pid,Why}`, where `Pid` is the identifier of process a, and `Why` is a value that describes the reason why a died. A special built-in turns a process into a system process.

Erlang chose to model all failures as permanent failures, in accordance with its philosophy of "Let it fail". That is, keeping the fault model simple allows the recovery algorithm to be simple as well. This simplicity is very important for correctness. We can see our model as an extension of Erlang's model with temporary failures and with a fault stream. Furthermore, our model is designed for a richer language than Erlang, which only has stationary ports (in our terminology). Chapter 5 will show how to program something similar to process linking in Oz.

4.5.3 The first fault model of Mozart

Our argument against the use of exceptions to handle distribution failures comes from the original fault model used in Oz, which was introduced with the first release of Mozart in 1998. The original model overlaps with the model we propose in this chapter. It was providing much more fault information (most of which was not used in practice) and was supporting both synchronous and asynchronous handlers. The major difference was the ability to define synchronous failure handlers, i.e., handlers that are called when attempting an

operation on a failed entity. The programmer could either ask for an exception or provide a handler procedure that replaces the operation. The failure handler was defined for a given entity and with certain conditions of activation.

Instead of the synchronous handlers, programmers favored a kind asynchronous handler, called a *watcher*. A watcher is a user procedure that is called in a new thread when a failure condition is fulfilled. The fault stream we propose in this paper simply factors out how the system informs the user program. It also avoids race conditions related to the watcher registry system, which could make one miss a fault state transition. And finally, a watcher could not be triggered by a transition to state `ok`. The latter soon revealed to be problematic for handling temporary failures.

An alternative model. The original model is criticized in [GGV04], which proposes an alternative model. That paper proposes something similar to our fault stream and an operation to make an entity fail locally. In order to handle faults, it proposes to explicitly break the transparent distribution of a failed entity. The local representative of the failed entity is disconnected from its peers and is put in a fault state equivalent to `localFail`. Another operation replaces that entity by a fresh new entity. This model has the advantage to avoid blocking threads on failed entities, because you can replace a failed entity by a healthy one. But this replacement introduces inconsistencies in the application's shared memory. We were not able to give a satisfactory semantics that takes into account these inconsistencies.

5

Applications

We present several abstractions that show how to program with the model we proposed in the former chapters. We first show how to hide network delays in a lazy producer/consumer situation, with a bounded buffer. We propose two implementations for the buffer: a fully declarative version, and a version that automatically adapts the buffer size.

We also show how to implement Erlang-like processes in Oz. Process linking and monitoring is very easy to implement. We then provide an abstraction that deals with temporary failures, and guarantees a consensus about failures in a set of processes monitoring each other. The consensus is reached by a vote among the correct processes.

5.1 Distributed lazy producer/consumer

Assume we have a component producing a stream lazily, and sharing that stream with other components, possibly on other sites. From a language point of view, those components simply share a logic variable. Consumers make the variable needed, which awaken the producer. The latter binds the variable to a pair `X|T`, where `T` is computed lazily as well.

This scheme is a nice example of a declarative communication channel between components. Moreover, its performance is not bad: making the variable needed typically costs one message from the consumer to the producer, and binding the variable costs one message in the other direction. So the variable imposes a communication delay of one round-trip per element. Note that this delay is independent from the number of consumers.

```
fun {BoundedBuffer N Xs}
   fun lazy {Deliver Xs Xr}
      case Xs of X|Xt then X|{Deliver Xt thread Xr.2 end} end
   end
in
   {Deliver Xs thread {Drop N Xs} end}
end
```

Snippet 5.1: A first implementation of a bounded buffer

5.1.1 A bounded buffer

In order to avoid the communication delay, one may insert a buffer between the producer and the consumer. The buffer triggers the evaluation of n elements ahead of the consumer. If the number n is well chosen, and the consumer does not run faster than the producer, then the consumer will not wait for reading one element from the stream. The value n is chosen such that the average time for producing one element, together with the communication delay, does not exceed the average time for consuming n elements.

Snippet 5.1 shows an implementation of a bounded buffer which is equivalent to the one proposed in [VH04]. The function `BoundedBuffer` takes as input the size of the buffer n, and the lazy stream `Xs`, and returns a lazy stream `Ys`:

```
Ys={BoundedBuffer N Xs}
```

The value of `Ys` is equal to `Xs`, except that if m elements of `Ys` are computed, $m + n$ elements of `Xs` are computed. The function call `{Drop N Xs}` returns the list `Xs` without its first `N` elements.

Behavior analysis. First, let us notice that calling `BoundedBuffer` on the producer's site will not fit our needs. Indeed, in that case, the lazy computation associated to the output variable `Ys` is on the producer side. When the consumer makes that variable needed, a full round-trip to the producer's site is necessary to trigger the lazy computation and send the value back.

Suppose now that `BoundedBuffer` is called on the consumer's site. When the consumer reads an element, it triggers a local lazy computation which returns immediately, if the element is available. At the same time, the lazy computation triggers the need for an element n positions ahead in the stream. However, when the consumer reads an extra element, the element n positions ahead will be requested *whenever the element before is delivered on the consumer's site*.

To illustrate that behavior, assume we have a producer/consumer pair with a bounded buffer of size $n = 5$. Let us analyze what happens when the consumer reads three elements from the stream. The interactions between both sites are shown in the left picture of Figure 5.1 on the facing page. The arrows to the

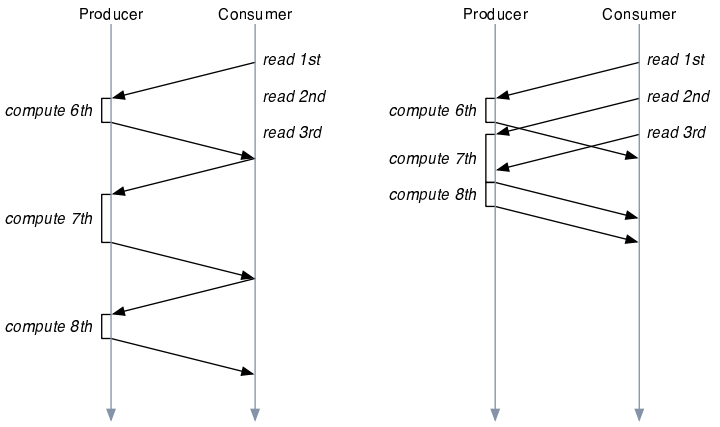


Figure 5.1: Network behavior of two implementations of a bounded buffer of size 5

left represent the messages that make a variable needed, while the arrows to the right are the messages with the binding of the corresponding variable. What we observe is that an element cannot be requested before the list pair containing the former element arrives on the consumer's site.

What we really want is something like the right picture of Figure 5.1. For each element read, the element n positions ahead should be requested as soon as possible. With this behavior, the elements are still produced in a sequential way, but the message round-trips to trigger the production of elements are truly concurrent. In the first behavior, those message round-trips are serialized.

5.1.2 A correct bounded buffer

Snippet 5.2 on the next page shows an implementation that provides the desired behavior. The producer performs the following statement on its site.

```
Es={Encapsulate Xs}
```

The returned value is a pair of variables that will be bound to streams. The first variable is the stream `Xs`, while the second variable is a stream `Rs` that is built by the consumer and read by the producer. For each element appearing on that stream, the producer requests one extra element on the data stream `Xs`. This is done by the thread running procedure `Prepare` on the producer. The consumer makes the following call to get a stream `Ys`.

```
Ys={DecapsulateN N Es}
```

This immediately builds a stream with `N` elements that will be read by the producer. Then, for every element consumed on `Ys`, the stream `Rs` is appended with the statement `Rs=unit|Rt`.

```
fun {Encapsulate Xs}
   proc {Prepare Rs Xs} {Prepare Rs.2 Xs.2} end
   Rs
in
   thread {Prepare Rs Xs} end
   Xs#Rs
end

fun {DecapsulateN N Es}
   fun {Prepend K Xt}
      if K>0 then unit|{Prepend K-1 Xt} else Xt end
   end
   fun lazy {Deliver Xs Rs}
      case Xs of X|Xt then Rt in
         Rs=unit|Rt                  % trigger need at producer
         X|{Deliver Xt Rt}
      end
   end
   Rt
in
   Es.2={Prepend N Rt}               % trigger N elements ahead
   {Deliver Es.1 Rt}
end
```

Snippet 5.2: A correct implementation of a bounded buffer

Let us now check that the behavior of the abstraction corresponds to the picture on the right of Figure 5.1. The arrows from right to left correspond to the bindings `Rs=unit|Rt`, while the arrows from left to right correspond to the bindings of the producer's output stream. Note that the bindings of `Rs` are performed immediately by the consumer. This is because the variables `Rs` are created on the consumer's site, hence the coordinators of those variables are on that site, and variable bindings are performed by the variable's coordinator. A more detailed explanation can be found in Section 8.2.3.

If several consumers are present, the stream can be encapsulated once, and each consumer decapsulates it by applying `DecapsulateN`. The producer will be driven by the consumer that requests the furthest ahead. However, the network behavior involved by the stream `Rs` is more difficult to describe, since the consumers will share that stream. Therefore a binding like `Rs=unit|Rt` might require an intermediate network message to another consumer site, if that other site holds the coordinator of `Rs`.

5.1.3 An adaptive bounded buffer

Snippet 5.3 provides a replacement for the function `DecapsulateN`. The stream is decapsulated on the consumer's site by the statement

```
fun {Decapsulate Es}
   fun lazy {Deliver Xs Rs}
      case Xs of X|Xt then Rt in
         Rs = if {Not {IsDet Xt}} then unit|unit|Rt
              elseif {Not {IsDet Xt.2}} then unit|Rt
              else Rt end
         X|{Deliver Xt Rt}
      end
   end
   Rt
in
   Es.2=unit|Rt
   {Deliver Es.1 Rt}
end
```

Snippet 5.3: An adaptive bounded buffer

```
Ys={Decapsulate Es}
```

This new function no longer takes a buffer size, but instead adapts how elements are requested ahead in order to always have one element ready at the consumer's site.

Let us make a quick comparison between the functions `DecapsulateN` and `Decapsulate`. The main difference is the binding of `Rs`, the second argument of the internal lazy function, which is called `Deliver` in both versions. For each consumed element, the adaptive version checks how many elements are available in front of `Xt`. We use the function `IsDet` which returns **`true`** if its argument is determined, and **`false`** otherwise. If no element is available (`Xt` is not determined), the size of the buffer is increased by triggering the need for two extra elements with the statement `Rs=unit|unit|Rt`. If exactly one element is available (`Xt.2` is not determined yet), we keep the same buffer size by requesting one extra element (`Rs=unit|Rt`). If more than one element is available, we decrease the buffer size by not requesting any extra element (`Rs=Rt`).

This adaptive version of the bounded buffer will work well if the consumer reads the stream at a regular pace.

5.1.4 A batch processing buffer

The reader might be surprised by the solution proposed in the previous sections. The abstractions effectively improve the network behavior of lazy evaluation, but they do it by avoiding the distributed mechanism of lazy evaluation. We were also disappointed by this solution when we realized this. So we came up with a solution that relies on the distributed by-need mechanism.

In order to avoid the sequential "ping-pong" effect illustrated in Figure 5.1, one may let the producer site trigger the evaluation of several elements in a row.

```
proc {BatchBuffer N Xs}
   proc {BatchLoop I Xs}
      if I>0 then {BatchLoop I-1 Xs.2} else
         {WaitNeeded Xs} {BatchLoop N Xs}
      end
   end
in
   thread {BatchLoop 0 Xs} end
end
```

Snippet 5.4: An abstraction that forces the evaluation of a stream in batches

Whenever the first element is requested, an abstraction forces the production of n elements. In other words, we can force the producer to work by *batches*.

The abstraction is shown in Snippet 5.4. One simply has to call

```
{BatchBuffer N Xs}
```

on the producer's site. The procedure creates a thread that detects when an element is needed, and automatically makes the $n-1$ following elements needed. The thread then waits until the element after that batch becomes needed, and requests a new batch.

This abstraction can be used solely, or in combination with the simple bounded buffer given in Snippet 5.1. When used solely, the full round trip delay will occur only once every n elements. If the consumer uses the simple bounded buffer, the round-trip delay can be completely hidden if n is large enough.

5.2 Processes à la Erlang

In the language Erlang, almost everything is a process. A process consists of a port, on which messages can be sent, and a function that processes the messages. A process is created by the primitive `spawn`, and messages are sent with the binary operator `!`:

```
Pid = spawn(F)     % create a process from a function F

Pid ! Msg          % send a message Msg to process Pid
```

The function takes a message from the incoming queue with the statement `receive`. The statement uses pattern matching to specify valid messages, and subsequent actions.

It is pretty easy to write a function `Spawn` in Oz that is similar to the corresponding Erlang primitive. The function, shown in Snippet 5.5 on the next page with an example, creates a port and runs the procedure in its own thread.

```
%% create a process with unary procedure Process
fun {Spawn Process}
   Xs Ys Self={NewPort Xs}
   fun {Loop Xs}
      case Xs of user(M)|Xt then M|{Loop Xt} end
   end
in
   thread Ys={Loop Xs} end
   thread {Process Ys} end
   Self
end

%% send message M to process A
proc {SendProc A M}
   {Send A user(M)}
end
```

Snippet 5.5: Spawning an Erlang-like process in Oz

The procedure takes the stream of messages in argument. The procedure should process the messages in a sequential way. The procedure `SendProc` sends a message `M` to a process `A`. Note that messages are put in a record `user(M)`, in order to distinguish them from system messages that are introduced below.

Here is an example with two processes `A` and `B` sending each other ping-pong messages:

```
proc {ProcessA Xs}
   case Xs of X|Xt then
      case X
      of stop    then skip
      [] ping(P) then {SendProc P pong(A)} {ProcessA Xt}
      end
   end
end
A={Spawn ProcessA}

proc {ProcessB Xs}
   {SendProc A ping(B)}
   case Xs of pong(P)|_ andthen P==A then skip end
end
B={Spawn ProcessB}
```

Process linking. Erlang processes can be *linked* together, such that when one of them terminates abnormally, the other ones die also, unless they are system processes. System processes are explained below. Linking is symmetric, and implements a property which states that a group of processes must crash as soon as one of them crashes. A process `A` adds the process `B` to its link set by evaluating the built-in function `link(B)`.

```
%% link process Self to process A
proc {Link Self A}
   {Send Self link(A)} {Send A link(Self)}
end

%% change the 'system process' flag
proc {SetSystem Self B}
   X in {Send Self system(B X)} {Wait X}
end

%% create a process with unary procedure Process
fun {Spawn Process}
   Xs Ys Self={NewPort Xs} T
   fun {Loop Xs Linkset Sys}
      case Xs of X|Xt then
         case X
         of user(M) then M|{Loop Xt Linkset Sys}
         [] system(B X) then X=unit {Loop Xt Linkset B}
         [] link(A) then {Monitor A} {Loop Xt A|Linkset Sys}
         [] exit(E) then {Notify E Linkset} nil
         [] exit(A E) andthen Sys then X|{Loop Xt Linkset Sys}
         [] exit(A normal) then {Loop Xt Linkset Sys}
         [] exit(A E) then {Kill T} {Notify E Linkset} nil
         end
      end
   end
   proc {Monitor A}
      thread
         if {Member permFail {GetFaultStream A}} then
            {Send Self exit(A crashed)} end
      end
   end
   proc {Notify E Linkset}
      for A in Linkset do {Send A exit(Self E)} end
   end
in
   thread Ys={Loop Xs nil false} end
   thread
      T={Thread.this}
      try {Process Ys} {Send Self exit(normal)}
      catch E then {Send Self exit(E)} end
   end
   Self
end
```

Snippet 5.6: Asymmetric linking and monitoring of processes

In Snippet 5.6 on the facing page we propose a new implementation of `Spawn` that handles linking and system processes. The internal loop of the process handles system messages, and maintains a link set, i.e., a list of processes that are notified of the termination of the current process. The message `link(A)` is sent by the procedure `Link`, and notifies the current process that it is linked to process `A`. The current process adds `A` to its link set, and monitors `A` to detect a failure that `A` itself would not be able to notify.

When the process terminates, it sends the message `exit(E)` to itself, in order to notify its link set. The value `E` describes the reason of the termination. Processes in the link set are notified with the message `exit(Self E)`. The latter message is handled in a different way, depending on whether the current process is a system process. Non-system processes are killed when `E` is not `normal`, while system processes simply receive the message. The message `system(B)` is sent by procedure `SetSystem` by the process itself in order to change its status (system process or not).

5.3 Failure by majority

Failure detectors are extremely useful for writing programs that react to partial failures. Failure handling can be written in a rule-based style. However our detector model is weak in the sense that detectors are not required to be consistent between sites. This can sometimes lead distributed programs to behave strangely, because some sites consider an entity failed, and others don't.

In this section we propose an algorithm that makes a group of N processes find a *consensus* about the failure status of a given process. Group members may themselves fail during the consensus algorithm. However, the algorithm is guaranteed to reach consensus about crashed processes if less than $N/2$ processes crash.

5.3.1 Algorithm

The idea is quite simple. For the sake of simplicity, let us assume that the group wonders about the status of process S. Whenever process P suspects S, it broadcasts $vote(P, +1)$. If it changes its mind about S, it broadcasts $vote(P, -1)$. Every process sums the values received from every other process, and maintains how many of them have a positive account. When this number becomes greater than $N/2$, it means that a majority of processes have suspected S. At that point a message is broadcast to make all correct processes consider S as permanently failed, i.e., `{Break` S`}`. The latter message must be broadcast in a reliable manner, in order to guarantee the consistency between processes.

The algorithm is described in Figure 5.2 on the next page in the style of Guerraoui *et al.* [GR06]. Processes perform actions when some events occur, some of those events being messages. An implementation with objects is given in Snippet 5.7 on page 71. The specificity of the algorithm is that there is

upon S is suspect **do**
broadcast $vote(self, +1)$

upon S is non-suspect **do**
broadcast $vote(self, -1)$

upon event $vote(P, x)$ **do**
$count.P := count.P + x$

upon event $kill(S)$ **do**
execute `{Break` S`}`

upon $\#\{P \mid count.P > 0\} > \frac{N}{2}$ **do**
reliableBroadcast $kill(S)$

Figure 5.2: Majority voting for consensus on failure status of S

only one possible decision, and that decision is taken whenever a majority of processes agreed with that decision. Moreover, the decision is monotonic.

Note. Counting the votes from a given process P consists in summing all the $+1$'s and -1's sent by P. The counter allows to receive votes from P in any order. If all messages from P are received in order, the counter will always belong to the set $\{0, 1\}$.

5.3.2 Correctness

Assume that S has crashed. Because we rely on eventually perfect failure detectors, all correct processes will eventually suspect S forever. So at some point, at least $N/2$ processes will broadcast a positive vote. And all correct processes will eventually sum all the votes broadcast by the positive majority we mentioned. Note that broadcasting the decision is only an optimization in that case.

Now assume that S has not crashed. There may be enough suspicions among the other processes to let one of them observe a majority of positive votes. The latter observation might be temporary if processes change their mind quickly. In that case, broadcasting the decision with a reliable algorithm ensures that all correct processes will eventually consider S as permanently failed.

In order to show the necessity of the final broadcast, let us imagine an extreme case where only one process P observes a majority of positive votes. Such a scenario is depicted in Figure 5.3 on the next page. A group of N_X processes X suspect S, then cancel their suspicion. Their messages reach P faster than another group Y. The group Y has N_Y processes, that also suspect S then revise their judgment. If we have both $N_X, N_Y < N/2$ and $N_X + N_Y > N/2$, then only P will observe a majority of positive votes. If P does not broadcast its observation, its view will not be consistent with the other processes.

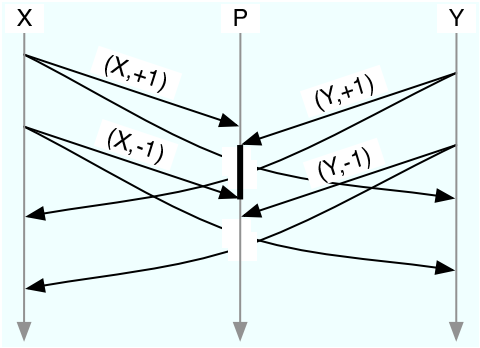


Figure 5.3: A scenario where only one process P observes a majority

5.3.3 The whole code of processes

The process itself is implemented as an object that multiplexes voters. Each process has one voter per other process it monitors. The whole code is given in Snippets 5.8 to 5.12 on pages 72–74.

The class `BaseProcess` is the mother class of all processes. It creates the process' port, and a thread that processes messages. An identifier `id` is also provided. The latter is useful if one wants to use a process as a key in a dictionary.

The class `ProcessWithFailureDetector` extends `BaseProcess` by monitoring other processes. It is initialized with the list of processes, with their identifier and port. A process using this class should implement method `failure()` in order to handle failures. In our example, this method is defined in subclass `MonitoringProcess`.

The classes `BestEffortBroadcast` and `ReliableBroadcast` provide simple methods to broadcast messages to all processes. The latter is *reliable* in the sense that either all correct processes deliver the message, or none of them deliver it. Each process that receives the message broadcast it once, too. This ensures the delivery in case the original sender crashes. This implementation is not efficient, but it fulfills its specification.

The class `MonitoringProcess` is the main class. It multiplexes its voters, and provides all the support they need for communicating. In method `failure`, you can see that the "opinion" of a voter is changed when the failure state of the corresponding process changes.

5.3.4 Variants

The algorithm we gave was kept simple for the sake of explanation. But it is quite flexible, and admits variants, which are easy to implement. Here are a few ideas that can improve the abstraction.

- One may change the number of positive votes that must be reached to trigger the decision. A value of $N/3$ may be considered enough, for instance.
- We have assumed the number of processes to be known and fixed. The algorithm works fine if that number varies over time, and processes regularly update this number. The condition for triggering the decision simply has to be reevaluated.
- One should discard crashed processes when counting votes. One may also discard suspect processes, such that only known correct processes are taken into account. The latter idea requires more attention, because as such, it would allow one process to suspect all other processes.

```
class Voter
   attr
      broadcast       % broadcast procedure
      rbBroadcast     % reliable broadcast procedure
      decide          % decide procedure
      total           % number of processes
      id              % identifier of this process
      opinion         % this process' opinion (true or false)
      votes           % accumulated votes from each voter

   meth init(broadcast:B rbBroadcast:RB decide:D N)
      broadcast := B
      rbBroadcast := RB
      decide := D
      total := N
      id := {NewName}
      opinion := false
      votes := {NewDictionary}
   end

   %% set the process' opinion (true for suspicion)
   meth propose(B)
      if B \= @opinion then
         opinion := B
         {@broadcast vote(@id (if B then 1 else ~1 end))}
      end
   end

   %% receive a vote from Id
   meth vote(Id X)
      @votes.Id := {Dictionary.condGet @votes Id 0} + X
      if X > 0 then N in
         N={Length {Filter {Dictionary.items @votes} IsPos}}
         if N*2 > @total then {@rbBroadcast decide} end
      end
   end

   %% receive decision
   meth decide
      {@decide true}
   end
end

fun {IsPos X} X>0 end
```

Snippet 5.7: Implementation of the majority voting algorithm

```
class BaseProcess
   feat port id

   meth init()
      Xs in
      thread
         try {ForAll Xs self}
         catch _ then {Kill self.port}
         end
      end
      self.port={NewPort Xs}
      self.id={NewName}
   end
end
```

Snippet 5.8: Base class of processes

```
class ProcessWithFailureDetector from BaseProcess
   attr processes

   meth initProcesses(IPs)
      %% IPs is a list of pairs id#port
      processes := {List.toRecord p IPs}
      for Id#P in IPs do
         thread
            {Wait P}
            for X in {GetFaultStream P} do
               {Send self.port failure(Id X)}
            end
         end
      end
   end
end
```

Snippet 5.9: A class for processes that monitor each other

```
class BestEffortBroadcast from ProcessWithFailureDetector
   meth broadcast(M)
      {Record.forAll @processes proc {$ P} {Send P M} end}
   end
end
```

Snippet 5.10: Implementation of best-effort broadcast

```
class ReliableBroadcast from BestEffortBroadcast
   attr delivered

   meth initProcesses(IPs)
      BestEffortBroadcast,initProcesses(IPs)
      delivered := nil
      {self CheckDelivered}
   end

   meth CheckDelivered
      %% some kind of 'garbage collection' on delivered
      delivered := unit|{List.takeWhile @delivered
                         fun {$ Id} Id \= unit end}
      thread
         {Delay 360000} {Send self.port CheckDelivered}
      end
   end

   meth rbBroadcast(M)
      %% note: unicity of messages is guaranteed by user
      {self broadcast(rbDeliver(M))}
   end

   meth rbDeliver(M)
      if {Not {Member M @delivered}} then
         delivered := M|@delivered
         {Send self.port M}
         {self broadcast(rbDeliver(M))}
      end
   end
end
```

Snippet 5.11: Implementation of an “eager” reliable broadcast

```
class MonitoringProcess from ReliableBroadcast
   attr voters

   meth initProcesses(IPs)
      N={Length Ps}
   in
      ReliableBroadcast,init(IPs)
      voters := {Record.mapInd {List.toRecord v IPs}
                 fun {$ I P}
                    proc {B M}
                       {self broadcast(voting(I M))}
                    end
                    proc {RB M}
                       {self rbBroadcast(voting(I M))}
                    end
                    proc {D B}
                       if B then {self kill(I)} end
                    end
                 in
                    {New Voter init(broadcast:B
                                    rbBroadcast:RB
                                    decide:D
                                    N)}
                 end}
   end

   %% relay a message for a voter
   meth voting(I M)
      {@voters.I M}
   end

   %% failure detector notification, maybe change opinion
   meth failure(I State)
      {@voters.I propose(State \= ok)}
   end

   %% decide whether a process has failed
   meth kill(I)
      {Break @processes.I} {Kill @processes.I}
      voters := {AdjoinAt @voters I proc {$ _} skip end}
   end
end
```

Snippet 5.12: Main class, with one voter per process in the group

6

Language semantics

We now give a formal support to the language concepts we presented in Chapters 2, 3 and 4. This chapter defines an operational semantics to Oz without taking distribution into account. The next chapter presents a refinement of that semantics, which models distribution, network, and failures. The refinement also give a semantics to the annotation system and the failure handling primitives.

Section 6.1 defines how to translate a program in Full Oz into an equivalent program in Kernel Oz. Section 6.2 gives the notations and basic ingredients of the semantic definitions. Sections 6.3 details the semantics of the declarative part of the kernel language, while Section 6.4 gives the semantics of the non-declarative part of the language.

6.1 Full language to kernel language

Every Oz program is equivalent to a program in Kernel Oz. In Chapter 2, we have introduced the kernel language, and syntactic sugar of common idioms in the full language. We now see how to formally translate an Oz program into an equivalent Kernel Oz program. The kernel language is given by the grammar in Figure 6.1. Both the declarative and non-declarative parts of the language are given in the grammar.

The translation is defined by the relation $\Rightarrow$, which reduces statements to simpler statements. The kernel program equivalent to a given Oz program is defined as the fixpoint of the program by the relation. This relation is structural: one can reduce a statement inside another statement.

For the rest of the section, D denotes a declaration (statement or identifier), E an expression, P a pattern, S a statement, SE a statement or expression, and X and Y identifiers.

$$
\begin{aligned}
S ::= \ & \texttt{skip} \mid S_1 S_2 \mid \texttt{thread}\ S\ \texttt{end} \\
& \mid \texttt{local}\ X\ \texttt{in}\ S\ \texttt{end} \\
& \mid X\texttt{=}Y \mid X\texttt{=}f(Y_1 \ldots Y_n) \\
& \mid \texttt{if}\ X\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid \texttt{case}\ X\ \texttt{of}\ f(Y_1 \ldots Y_n)\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid \{\texttt{WaitNeeded}\ X\} \\
& \mid \texttt{proc}\ \{X\ Y_1 \ldots Y_n\}\ S\ \texttt{end} \mid \{X\ Y_1 \ldots Y_n\} \\
& \mid \texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \mid \texttt{raise}\ X\ \texttt{end} \mid \{\texttt{FailedValue}\ X\ Y\} \\
& \mid X\texttt{=!!}Y \\
& \mid \{\texttt{NewCell}\ X\ Y\} \mid X_0\texttt{=}Y\texttt{:=}X_1
\end{aligned}
$$

Figure 6.1: Grammar of Kernel Oz

Expanding declarations. The following rules split up declarations into declared identifiers and initializing statements. It simplifies declarations as "**local** `X=foo` **in**". We assume that the statements that appear in the declaration D have already been reduced to kernel statements.

$$
\begin{aligned}
D\ \texttt{in}\ SE &\Rightarrow \texttt{local}\ D\ \texttt{in}\ SE\ \texttt{end} \\
\texttt{local}\ D\ \texttt{in}\ SE\ \texttt{end} &\Rightarrow \texttt{local}\ \mathsf{decl}(D)\ \texttt{in}\ \mathsf{stmt}(D)\ SE\ \texttt{end} \\
\texttt{local}\ X_1\ X_2 \ldots X_n\ \texttt{in}\ SE\ \texttt{end} &\Rightarrow \texttt{local}\ X_1\ \texttt{in} \\
&\qquad \texttt{local}\ X_2 \ldots X_n\ \texttt{in}\ SE\ \texttt{end} \\
&\quad \texttt{end}
\end{aligned}
$$

The functions decl and stmt respectively return the declared identifiers and the statements of a declaration. The function ident returns the set of identifiers in a pattern; each identifier is declared at most once. Those functions are defined as

$$
\begin{aligned}
\mathsf{decl}(X) &= \{X\} \\
\mathsf{decl}(P\texttt{=}E) &= \mathsf{ident}(P) \\
\mathsf{decl}(P\texttt{=}E_1\texttt{:=}E_2) &= \mathsf{ident}(P) \\
\mathsf{decl}(\texttt{proc}\ \{X\ Y_1 \ldots Y_n\}\ S\ \texttt{end}) &= \{X\} \\
\mathsf{decl}(S) &= \emptyset \\
\mathsf{decl}(D_1 \ldots D_n) &= \mathsf{decl}(D_1) \cup \ldots \cup \mathsf{decl}(D_n)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ident}(X) &= \{X\} \\
\mathsf{ident}(f(P_1 \ldots P_n)) &= \mathsf{ident}(P_1) \cup \cdots \cup \mathsf{ident}(P_n)
\end{aligned}
$$

$$\begin{aligned}\mathsf{stmt}(X) &= \epsilon\\ \mathsf{stmt}(S) &= S\\ \mathsf{stmt}(D_1 \dots D_n) &= \mathsf{stmt}(D_1) \dots \mathsf{stmt}(D_n)\end{aligned}$$

Expanding nested expressions. Those are the kernel statements that contain an expression E in place of an identifier. The reduction introduces an identifier X, and expands the evaluation of E before evaluating the statement itself. The identifier X is chosen such as to not occur in the original statement. Notice that in the procedure call, the first non-identifier is expanded. We assume that $m, n \geq 0$.

$$\begin{aligned}E{=}E' &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ X{=}E'\ \texttt{end}\\ \texttt{if}\ E\ \texttt{then}\dots\texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{if}\ X\ \texttt{then}\dots\texttt{end})\ \texttt{end}\\ \texttt{case}\ E\ \texttt{of}\dots\texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{case}\ X\ \texttt{of}\dots\texttt{end})\ \texttt{end}\\ \texttt{proc}\ \{E\dots\}\ S\ \texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{proc}\ \{X\dots\}\ S\ \texttt{end})\ \texttt{end}\\ \{Y_1 \dots Y_m\ E\ E_1 \dots E_n\} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ \{Y_1 \dots Y_m\ X\ E_1 \dots E_n\}\ \texttt{end}\\ \texttt{raise}\ E\ \texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{raise}\ X\ \texttt{end})\ \texttt{end}\end{aligned}$$

Expanding expressions. Function definitions are expanded to procedures. The extra parameter X is chosen to not occur in a free position in E.

$$\texttt{fun}\ \{\dots\}\ E\ \texttt{end} \Rightarrow \texttt{proc}\ \{\dots\ X\}\ X{=}E\ \texttt{end}$$

Then we expand all the statements of the form $X{=}E$. The expansion often brings the assignment to X inside the language constructs, which sometimes declares new identifiers Y_i. If X occurs in those declarations, then we substitute this occurrence of X by another identifier. The result of this substitution is denoted Y_i^* or E^*.

$$\begin{aligned}X{=}(S\ E) &\Rightarrow S\ X{=}E\\ X{=}\texttt{thread}\ E\ \texttt{end} &\Rightarrow \texttt{thread}\ X{=}E\ \texttt{end}\\ X{=}\texttt{local}\ Y\ \texttt{in}\ E\ \texttt{end} &\Rightarrow \texttt{local}\ Y^*\ \texttt{in}\ X{=}E^*\ \texttt{end}\\ X{=}E_1{=}E_2 &\Rightarrow X{=}E_1\ X{=}E_2\\ X{=}\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} &\Rightarrow \texttt{if}\ E\ \texttt{then}\ X{=}E_1\ \texttt{else}\ X{=}E_2\ \texttt{end}\\ X{=}\texttt{case}\ E\ \texttt{of}\ f(Y_1 \dots Y_n)\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} &\Rightarrow \texttt{case}\ E\ \texttt{of}\ f(Y_1^* \dots Y_n^*)\ \texttt{then}\ X{=}E_1^*\ \texttt{else}\ X{=}E_2\ \texttt{end}\\ X{=}\texttt{proc}\ \{\$\dots\}\ S\ \texttt{end} &\Rightarrow \texttt{proc}\ \{X\dots\}\ S\ \texttt{end}\\ X{=}\{E\ E_1 \dots E_n\} &\Rightarrow \{E\ E_1 \dots E_n\ X\}\\ X{=}\texttt{try}\ E_1\ \texttt{catch}\ Y\ \texttt{then}\ E_2\ \texttt{end} &\Rightarrow \texttt{try}\ X{=}E_1\ \texttt{catch}\ Y^*\ \texttt{then}\ X{=}E_2^*\ \texttt{end}\\ X{=}\texttt{raise}\ E\ \texttt{end} &\Rightarrow \texttt{raise}\ E\ \texttt{end}\end{aligned}$$

The `lazy` expansion. Lazy functions can be defined by using **`fun lazy`** instead of **`fun`** in their definition. The simplest way to expand this construct is to create a thread that synchronizes on the demand, then evaluates the function's body expression. We assume that the parameter X does not occur in a free position in E.

$$\begin{array}{l}\textbf{fun}\ \texttt{lazy}\ \{\ldots\}\ E\ \textbf{end} \Rightarrow \textbf{proc}\ \{\ldots\ X\} \\ \qquad\qquad\qquad\qquad\quad \textbf{thread}\ \{\texttt{WaitNeeded}\ X\}\ X{=}E\ \textbf{end} \\ \qquad\qquad\qquad\qquad \textbf{end}\end{array}$$

While being correct, this expansion may suffer a slight performance overhead, especially if the function is recursive. The overhead comes from the fact that every recursive call creates a new thread. Recursive calls in tail position do not need this extra thread. For those, one may let the current thread suspend. This is correct as long as the initial call to the function is in a different thread. The following expansion optimizes tail recursive calls.

$$\begin{array}{l}\textbf{fun}\ \texttt{lazy}\ \{F\ X_1 \ldots X_n\}\ E\ \textbf{end} \Rightarrow \textbf{local}\ F'\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{proc}\ \{F'\ X_1 \ldots X_n\ X\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \{\texttt{WaitNeeded}\ X\}\ (X{=}E)^* \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{fun}\ \{F\ X_1 \ldots X_n\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{thread}\ \{F'\ X_1 \ldots X_n\}\ \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad \textbf{end}\end{array}$$

The identifier F' is chosen to not occur in the definition of F. The extra operation $(X{=}E)^*$ expands the statement $X{=}E$, and replaces every tail call to F by a similar call to F'.

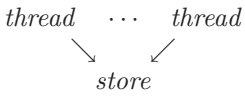
The `$` expansion. The main use of the `$` sign is in expressions that define a procedure, or in a procedure call. At some point such an expression E will be reduced in a statement of the form $X{=}E$. The following rules show how to reduce such a statement. $P(X)$ denotes a pattern containing an identifier X, and $P(\texttt{\$})$ is the same pattern with X replaced by `$`.

$$X{=}\textbf{proc}\ \{\texttt{\$}\ldots\}\ S\ \textbf{end} \Rightarrow \textbf{proc}\ \{X\ldots\}\ S\ \textbf{end}$$
$$X{=}\textbf{fun}\ \{\texttt{\$}\ldots\}\ E\ \textbf{end} \Rightarrow \textbf{fun}\ \{X\ldots\}\ E\ \textbf{end}$$
$$X{=}\{E_1 \ldots E_m\ P(\texttt{\$})\ E_{m+1} \ldots E_n\} \Rightarrow \{E_1 \ldots E_m\ P(X)\ E_{m+1} \ldots E_n\}$$

More linguistic abstractions. The full language provides even more statements, like class definitions, functor definitions, support for constraints, etc. We do not show how to expand those in this work. Material can be found in the book [VH04], and the documentation of Mozart [Moz99].

6.2 Basics of the semantics

Here we give the basics of the operational semantics of the language. The semantic rules prescribe how a running program can go from one state to another. A program state is called a *configuration*. A configuration consists of a set of threads connected to a shared store:



The thread is the basic unit of sequential computation. A computation consists of a sequence of computation steps, each of which transforms a configuration into another configuration. At each step, a thread is chosen, and executes an atomic operation. The choice of the thread is nondeterministic among all the executable threads in that configuration. Thread execution follow the interleaving semantics.

6.2.1 The store

The store is a single-assignment store (or constraint store), extended with first-class procedures, mutable entities, and a few other specific extensions [Smo95, VH04]. The extensions will be introduced step by step, together with their corresponding reductions rules. Those extensions are grouped together under the term *predicate store*.

The constraint store contains variable assignments made by the program. Assignments are between variables ($x{=}y$), or between variables and values ($x{=}v$). The constraint store is a conjunction of such assignments. It has the property of being monotonic, in the sense that one can only *add* assignments; existing assignments cannot be removed.

Store entailment. The constraint store has a logic nature, it can entail information that is not directly present in the store. For instance, the store $x{=}3 \wedge x{=}y$ entails $y{=}3$. We denote a store by σ, and a basic relation like an equality by β. The statement $\sigma \models \beta$ means that the store σ entails β. We assume that the store conjunction is associative, commutative, and has neutral element $\top$, which also denotes the empty store.

What the constraint store entails is defined by the following inference rules. The rules are given with premises on top of a horizontal line, and a conclusion below. The horizontal line is not shown when the premises are true. The very first rule states that the store entails at least what it contains, and in particular, that adding information in the store never reduces entailment.

$$\sigma \wedge \beta \models \beta \tag{6.1}$$

The next rules are specific to the equality relation. Rules (6.2) simply reflect that equality is reflexive, symmetric, and transitive. The metavariables t, u, v

can be either variables or values. The values we consider here are either simple values, or records.

$$\sigma \models t{=}t \qquad \frac{\sigma \models u{=}v}{\sigma \models v{=}u} \qquad \frac{\sigma \models t{=}u \quad \sigma \models u{=}v}{\sigma \models t{=}v} \tag{6.2}$$

We now define rules for record equality. Two records are equal if and only if they have identical labels, arities, and their fields are pairwise equal. The following two rules establish the "positive" side of this statement.

$$\frac{\sigma \models u_1{=}v_1 \quad \cdots \quad \sigma \models u_n{=}v_n}{\sigma \models f(u_1 \ldots u_n){=}f(v_1 \ldots v_n)} \tag{6.3}$$

$$\frac{\sigma \models f(u_1 \ldots u_n){=}f(v_1 \ldots v_n)}{\sigma \models u_i{=}v_i} \quad 1 \leq i \leq n \tag{6.4}$$

The constraint store can also *disentail* some equalities, i.e., inferring that they are false. The following rules state explicitly that records with different labels, arities, or different corresponding fields are unequal.

$$\frac{f{\neq}g \quad \text{or} \quad m{\neq}n}{\sigma \models f(u_1 \ldots u_m){\neq}g(v_1 \ldots v_n)} \tag{6.5}$$

$$\frac{\sigma \models u_i{\neq}v_i}{\sigma \models f(u_1 \ldots u_n){\neq}f(v_1 \ldots v_n)} \quad 1 \leq i \leq n \tag{6.6}$$

Determinacy. We can generalize a bit store entailment, in order to introduce derived concepts like determinacy. We say a variable x is determined by a store σ if σ entails that it is equal to a given value. We note this as $\sigma \models \mathit{det}(x)$. If the store cannot infer the value of the variable, we say that the variable is free.

$$\frac{\sigma \models x{=}v}{\sigma \models \mathit{det}(x)} \quad \text{for a certain value } v \tag{6.7}$$

Ask and tell. The two basic operations on a store are called *ask* and *tell*. The ask operation queries the store to know whether a given constraint is entailed or disentailed. Asking β on store σ returns a positive answer if $\sigma \models \beta$, a negative answer if $\sigma \models \neg\beta$. There is no answer otherwise. The monotonicity of the store guarantees that the answer of an ask never changes.

The tell operation adds a basic constraint to a store, provided that the store remains consistent. The store becomes inconsistent as soon as it infers something like 1=2, for instance. Telling β to σ updates the store to $\sigma \wedge \beta$. The rules that update the store are written such that they never make the store inconsistent. If an inconsistency could be introduced by a program statement, that statement should fail.

Predicate store. The predicate store is subject to the principle of substitution by equals. The following inference rule states that an instance of predicate p is entailed by the store if the store contains a similar predicate whose arguments are pairwise equal.

$$\frac{\sigma \models u_1{=}v_1 \quad \cdots \quad \sigma \models u_n{=}v_n}{\sigma \wedge p(u_1,\ldots,u_n) \models p(v_1,\ldots,v_n)} \tag{6.8}$$

Contrary to the constraint store, elements of the predicate store can be removed, or replaced. The valid ways to update the predicate store depends on each predicate, and is defined by the semantic rules.

6.2.2 Structural rules

The semantics are given by transition rules (or *reduction* rules[1]) that describe valid computation steps. The rules have the form

$$\frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T}' \;\|\; \sigma'} \quad \text{if } C$$

It states that a configuration with a multiset of threads $\mathcal{T}$ and store σ can be reduced to the configuration with threads $\mathcal{T}'$ and store σ', provided the condition C is fulfilled. We often write the left-hand side of the rule as a pattern, so that a configuration must match the pattern for the rule to be applicable. The disjoint union of multisets is written with commas, and singletons are written without curly braces. For instance, "$T_1, \mathcal{T}, T_2$" stands for $\{T_1\} \uplus \mathcal{T} \uplus \{T_2\}$. There is no ambiguity because of the thread syntax.

The following two rules are convenient for simplifying the expression of the rules. The first one expresses the relative isolation of concurrent threads: a subset of the thread may reduce without directly affecting the other threads.

$$\frac{\mathcal{T},\mathcal{U} \;\|\; \sigma}{\mathcal{T}',\mathcal{U} \;\|\; \sigma'} \quad \text{if} \quad \frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T}' \;\|\; \sigma'} \tag{6.9}$$

The second rule states that stores can be considered up to equivalence. This allows to choose the most convenient representation for a store in a reduction rule.

$$\frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T} \;\|\; \sigma'} \quad \text{if } \sigma \text{ and } \sigma' \text{ are equivalent} \tag{6.10}$$

Store equivalence is defined as follows. Let us first consider the constraint store. Two stores $\sigma = \beta_1 \wedge \cdots \wedge \beta_n$ and $\sigma' = \beta'_1 \wedge \cdots \wedge \beta'_{n'}$ are equivalent if

$$\sigma \models \beta'_i \quad \text{for every } i, \text{ and} \qquad \sigma' \models \beta_j \quad \text{for every } j. \tag{6.11}$$

[1] This expression comes from the chemical analogy of transition rules, where the execution takes a statement, and *reduces* it to a simpler statement.

The other part of the store follows a similar rule, except that each instance of a predicate in σ must correspond to *exactly* one predicate in σ'. This is necessary for some predicates, like the one that defines the current state of a cell, which must occur exactly once per cell in the store.

6.3 Declarative subset of the language

6.3.1 Sequential and concurrent execution

A thread is a sequence of statements $S_1 S_2 \dots S_n$. Parentheses are introduced to avoid ambiguities when necessary. The empty thread is written (). The abstract syntax of threads can thus be defined as

$$T ::= () \mid S\,T \tag{6.12}$$

The empty thread reduces to an empty multiset of threads. A nonempty thread reduces by reducing its first statement. The latter rule will again simplify the expression of rules.

$$\frac{() \;\|\; }{\sigma \;\|\; \sigma} \qquad\qquad \frac{S\,T \;\|\; S'\,T}{\sigma \;\|\; \sigma'} \text{ if } \frac{S \;\|\; S'}{\sigma \;\|\; \sigma'} \tag{6.13}$$

The empty statement, sequential composition, and thread statement are tied to the notion of thread. For those rules we have to show explicitly how they modify the structure of the threads. Notice that the latter creates a new thread with the statement S only.

$$\frac{\texttt{skip}\;T \;\|\; T}{\sigma \;\|\; \sigma} \qquad \frac{(S_1\,S_2)\,T \;\|\; S_1\,(S_2\,T)}{\sigma \;\|\; \sigma} \qquad \frac{\texttt{thread}\;S\;\texttt{end}\;T \;\|\; T, S}{\sigma \;\|\; \sigma} \tag{6.14}$$

6.3.2 Variable introduction

The **`local`** statement creates a new variable in the store, and make the declared identifier correspond to that variable. Instead of maintaining an explicit mapping between identifiers and variables, we directly substitute the declared identifier by its corresponding variable. The notation $S[X/x]$ stands for the substitution of X by x in S. The substitution takes care of lexical scope issues.

$$\frac{\texttt{local}\;X\;\texttt{in}\;S\;\texttt{end} \;\|\; S[X/x]}{\sigma \;\|\; \sigma} \quad \text{where } x \text{ is a fresh variable} \tag{6.15}$$

The condition of the rule requires x to be a fresh variable. A fresh variable is a variable that does not appear anywhere in the initial configuration. This can written formally, but we have chosen to keep the rule more readable.

Variable substitution. The identifier substitution operation is quite usual. Assume that θ denotes the substitution $[X/x]$. Let χ denote an identifier or a variable.

$$\chi\theta = \begin{cases} x & \text{if } \chi = X \\ \chi & \text{otherwise} \end{cases} \tag{6.16}$$

We now define the substitution inductively on the syntax of statements. The following statements do not involve lexical scoping.

$$(\texttt{skip})\theta = \texttt{skip} \tag{6.17}$$
$$(S_1\ S_2)\theta = S_1\theta\ S_2\theta \tag{6.18}$$
$$(\texttt{thread}\ S\ \texttt{end})\theta = \texttt{thread}\ S\theta\ \texttt{end} \tag{6.19}$$
$$(\chi_1{=}\chi_2)\theta = \chi_1\theta{=}\chi_2\theta \tag{6.20}$$
$$(\chi{=}c)\theta = \chi\theta{=}c \tag{6.21}$$
$$(\chi{=}f(\chi_1 \dots \chi_n))\theta = \chi\theta{=}f(\chi_1\theta \dots \chi_n\theta) \tag{6.22}$$
$$(\texttt{if}\ \chi\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end})\theta = \texttt{if}\ \chi\theta\ \texttt{then}\ S_1\theta\ \texttt{else}\ S_2\theta\ \texttt{end} \tag{6.23}$$
$$(\{\chi\ \chi_1 \dots \chi_n\})\theta = \{\chi\theta\ \chi_1\theta \dots \chi_n\theta\} \tag{6.24}$$
$$(\texttt{raise}\ \chi\ \texttt{end})\theta = \texttt{raise}\ \chi\theta\ \texttt{end} \tag{6.25}$$

In the following equations, we assume that the lexical scope introduced by the statement does not catch X, i.e., X is different from the identifiers $Y, Y_1, \dots, Y_n$.

$$(\texttt{local}\ Y\ \texttt{in}\ S\ \texttt{end})\theta = \texttt{local}\ Y\ \texttt{in}\ S\theta\ \texttt{end} \tag{6.26}$$
$$\begin{pmatrix}\texttt{case}\ \chi\ \texttt{of}\ f(Y_1 \dots Y_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{pmatrix}\theta = \begin{array}{l}\texttt{case}\ \chi\theta\ \texttt{of}\ f(Y_1 \dots Y_n) \\ \quad\texttt{then}\ S_1\theta\ \texttt{else}\ S_2\theta\ \texttt{end}\end{array} \tag{6.27}$$
$$(\texttt{proc}\ \{\chi\ Y_1 \dots Y_n\}\ S\ \texttt{end})\theta = \texttt{proc}\ \{\chi\theta\ Y_1 \dots Y_n\}\ S\theta\ \texttt{end} \tag{6.28}$$
$$(\texttt{try}\ S_1\ \texttt{catch}\ Y\ \texttt{then}\ S_2\ \texttt{end})\theta = \texttt{try}\ S_1\theta\ \texttt{catch}\ Y\ \texttt{then}\ S_2\theta\ \texttt{end} \tag{6.29}$$

We now define the substitution when X is caught by the lexical scope of the statements. We assume that $X \in \{X_1, \dots, X_n\}$.

$$(\texttt{local}\ X\ \texttt{in}\ S\ \texttt{end})\theta = \texttt{local}\ X\ \texttt{in}\ S\ \texttt{end} \tag{6.30}$$
$$\begin{pmatrix}\texttt{case}\ \chi\ \texttt{of}\ f(X_1 \dots X_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{pmatrix}\theta = \begin{array}{l}\texttt{case}\ \chi\theta\ \texttt{of}\ f(X_1 \dots X_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\theta\ \texttt{end}\end{array} \tag{6.31}$$
$$(\texttt{proc}\ \{\chi\ X_1 \dots X_n\}\ S\ \texttt{end})\theta = \texttt{proc}\ \{\chi\theta\ X_1 \dots X_n\}\ S\ \texttt{end} \tag{6.32}$$
$$(\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})\theta = \texttt{try}\ S_1\theta\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \tag{6.33}$$

6.3.3 Unification

The unification operation in Oz imposes equality between two terms. It incrementally tells basic constraints to the store until the equality is entailed or disentailed by the store. The operational semantics of unification is therefore

non atomic. This lack of atomicity permits a realistic extension of unification in the distributed case.

The following two rules terminate the unification when it is either entailed, or disentailed. The statement **`fail`** is used for the sake of readability; it is shorthand for **`raise failure end`**, which raises a failure exception.

$$\frac{u{=}v \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models u{=}v \tag{6.34}$$

$$\frac{u{=}v \;\|\; \texttt{fail}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models u{\neq}v \tag{6.35}$$

We then give the rule that incrementally tells basic constraints to the store. Those basic constraints are necessary for the unification to succeed. They are of the form $x{=}t$, where x is not determined by the store yet, and t is either a variable or a value.

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge x{=}t} \quad \text{if } \sigma \wedge u{=}v \models x{=}t \text{ and } \sigma \not\models det(x) \tag{6.36}$$

There exists an optional simplification rule, that rewrites a unification as another one. This simplification does not change the effect of unification, but it allows an implementation to simplify it.

$$\frac{u{=}v \;\|\; u'{=}v'}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \wedge u{=}v \models u'{=}v' \text{ and } \sigma \wedge u'{=}v' \models u{=}v \tag{6.37}$$

Example. Executing $x{=}f(y)$ with the store $\sigma \equiv x{=}f(x_1) \wedge x_1 = 2$ tells $y{=}2$ to the store, then reduce to **`skip`**. Indeed, the first reduction applies since the store inference rules give

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\sigma \wedge x{=}f(y) \models x{=}f(y)}{\sigma \wedge x{=}f(y) \models f(y){=}x} \qquad \sigma \wedge x{=}f(y) \models x{=}f(x_1)}{\sigma \wedge x{=}f(y) \models f(y){=}f(x_1)}}{\sigma \wedge x{=}f(y) \models y{=}x_1} \qquad \sigma \wedge x{=}f(y) \models x_1{=}2}{\sigma \wedge x{=}f(y) \models y{=}2}$$

The rule leads to the store $\sigma' \equiv \sigma \wedge y{=}2$, which entails $x{=}f(y)$:

$$\dfrac{\sigma' \models x{=}f(x_1) \qquad \dfrac{\sigma' \models x_1{=}2 \qquad \dfrac{\sigma' \models y{=}2}{\sigma' \models 2{=}y}}{\sigma' \models x_1{=}y}}{\sigma' \models f(x_1){=}f(y)}}{\sigma' \models x{=}f(y)}$$

6.3.4 Conditional statements

Those statements perform an *ask* operation on the store, and possibly block until a condition is entailed or disentailed.

The `if` statement. The classical conditional statement reduces depending on the value of its condition variable. The statement waits until the variable equals **`true`** or **`false`**, then reduces accordingly:

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\|\; S_1}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}\texttt{true} \tag{6.38}$$

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\|\; S_2}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}\texttt{false} \tag{6.39}$$

The value of x is usually determined by a boolean function, like a comparison operator. If x is different from **`true`** and **`false`**, the statement reduces by raising an exception (see Section).

The `case` statement. It can be seen as a linguistic abstraction for pattern matching, expressed in terms of a conditional statement, variable introduction, and record operations `Label` and `Arity`. However the concept is important enough to be presented with its semantic rules.

$$\frac{\begin{array}{r}\texttt{case}\ x\ \texttt{of}\ f(X_1 \ldots X_n)\\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{array} \;\|\; S_1[X_1/x_1]\cdots[X_n/x_n]}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}f(x_1 \ldots x_n) \tag{6.40}$$

$$\frac{\begin{array}{r}\texttt{case}\ x\ \texttt{of}\ f(X_1 \ldots X_n)\\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{array} \;\|\; S_2}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{\neq}f(x_1 \ldots x_n) \tag{6.41}$$

The pattern matches if the store entails the equality $x{=}f(x_1 \ldots x_n)$, for some variables $x_1, \ldots, x_n$. In case of a match, the statement reduces to S_1, where the identifiers X_i are substituted by the corresponding variables x_i in x. If the store disentails any such equality, the statement reduces to S_2. If the store does not contain enough information to decide one way or another, then the statement cannot reduce.

Waiting for determinacy. We have defined above what it means for a store to determine a variable. Waiting for the determination of a variable is the most direct way to show the dataflow behavior of variables. It can be expressed explicitly with the unary procedure `Wait`. Its semantics is extremely simple: it reduces to **`skip`** once its argument is determined.

$$\frac{\texttt{\{Wait}\ x\texttt{\}} \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models \mathit{det}(x) \tag{6.42}$$

6.3.5 Names and procedures

Names are unforgeable constants, and have therefore no textual representation. They are useful to give a unique identity to a language entity like a procedure or a cell. But they can also be used as first-class values by a programmer. Such a value can be confined by lexical scope to the implementation of a data structure, for keeping a feature hidden to the user. For instance, names are used to define private methods in a class, which are by default only accessible from within the class.

Names are created explicitly by the operation `NewName`. Its semantics are given by the following reduction rule. Every fresh name is guaranteed to be different from all other existing names and values. Names are created in a way similar to variables. The semantic statement $x{=}\xi$ is obtained by semantic rule reduction only. This reduction clearly separates the name creation from the binding of the variable x.

$$\frac{\texttt{\{NewName } x\texttt{\}} \;\|\; x{=}\xi}{\sigma \;\|\; \sigma} \quad \text{where } \xi \text{ is a fresh name} \tag{6.43}$$

Procedures. The **`proc`** statement creates a procedure in the store. The procedure value consists in a name ξ that is associated to a statement abstraction in the procedure store by the pair $\xi : \lambda X_1 \ldots X_n.S$. All the free identifiers of S are in the set $\{X_1, \ldots, X_n\}$. The name gives the procedure its identity.

Procedure application performs an ask to the store. For applying procedure p, p must be equal to a name ξ that is associated to a statement abstraction. Procedure application thus blocks if p is not determined by the store. Once the procedure is known, the call reduces to the abstracted statement, where each parameter is substituted by the corresponding argument in the call.

$$\frac{\texttt{proc \{}p\ X_1 \ldots X_n\texttt{\}}\ S\ \texttt{end} \;\|\; p{=}\xi}{\sigma \;\|\; \sigma \wedge \xi{:}\lambda X_1 \ldots X_n.S} \quad \xi \text{ a fresh name} \tag{6.44}$$

$$\frac{\{p\ x_1 \ldots x_n\} \;\|\; S[X_1/x_1] \cdots [X_n/x_n]}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models p{=}\xi \wedge \xi{:}\lambda X_1 \ldots X_n.S \tag{6.45}$$

6.3.6 By-need synchronization

Lazy evaluation, or demand-driven computation, is possible in Oz via the by-need synchronization mechanism. It works as follows. In a producer-consumer scheme, the producer and the consumer are in separate threads, and share a logic variable x. The producer simply blocks until a consumer requires x to be determined in order to reduce. The mechanism that allows the producer to detect the need of a consumer is called *by-need synchronization*. Note that the mechanism is very general, and allows several producers and consumers for a single variable.

The semantics is defined in terms of *ask* and *tell* on the *by-need store*. The latter is an extension of the constraint store, and is monotonic as well, which makes this language concept fully declarative. The predicate $needed(x)$ is used to synchronize producers and consumers: it is automatically told to the store by the consumer if the determinacy of x is required for its reduction. The producer uses the unary procedure `WaitNeeded` to synchronize on the entailment of the predicate by the store.

In our proposal, we consider that determined variables are needed by convention. This simplifies the behavior of a variable. We identify three states, which are ordered in this way: free, needed, and determined. State transitions follow that order, which ensures the monotonicity of the store. Moreover, this convention disambiguates a producer-consumer situation where a consumer would bind the shared variable. The variable automatically becomes needed, and the producer is woken up. We provide this property with the following inference rule.

$$\frac{\sigma \models det(x)}{\sigma \models needed(x)} \tag{6.46}$$

Now consider a statement S. We define $needed(S)$ as the set of variables which must be determined for S to be executable.

$$x \in needed(S) \quad \text{iff} \quad \begin{cases} \text{for every store } \sigma\text{:} \\ \text{if } S \text{ is executable with } \sigma\text{, then } \sigma \models det(x) \end{cases} \tag{6.47}$$

The condition can also be expressed as: the statement S cannot reduce in a configuration where x is not determined. The definition directly applies to the `Wait` statement: $x \in needed(\texttt{\{Wait } x\texttt{\}})$. The variable x is also needed by the statements "**if** $x \ldots$" and "**case** $x \ldots$".

The first reduction rule below describes how the predicate $needed(x)$ is told to the store. The second rule states that the statement `{WaitNeeded` x`}` asks the store for the predicate $needed(x)$, and reduces to **skip** once it is entailed.

$$\frac{S \;\|\; \sigma}{S \;\|\; \sigma \wedge needed(x)} \quad \text{if } x \in needed(S), \text{ and } \sigma \not\models needed(x) \tag{6.48}$$

$$\frac{\texttt{\{WaitNeeded } x\texttt{\}} \;\|\; \sigma}{\textbf{skip} \;\|\; \sigma} \quad \text{if } \sigma \models needed(x) \tag{6.49}$$

6.4 Nondeclarative extensions

6.4.1 Nondeterministic wait

The function `WaitTwo` takes two arguments, and returns the number of the argument that is determined (`1` or `2`). The returned value is nondeterministic in case both arguments are determined. It can be used for merging streams,

for instance.

$$\frac{\texttt{\{WaitTwo}\ x\ y\ z\texttt{\}}\ \|\ z\texttt{=1}}{\sigma\ \|\ \sigma} \quad \text{if } \sigma \models det(x) \tag{6.50}$$

$$\frac{\texttt{\{WaitTwo}\ x\ y\ z\texttt{\}}\ \|\ z\texttt{=2}}{\sigma\ \|\ \sigma} \quad \text{if } \sigma \models det(y) \tag{6.51}$$

6.4.2 Exception handling

We first introduce the **try** statement. In order to simplify the management of the scope defined by the statement, we consider a **catch** statement, which can only be obtained by the reduction of the first rule below. The **catch** statement itself reduces to **skip**. These two rules model all executions where no exception is raised.

$$\frac{\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}\ \|\ S_1\ (\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})}{\sigma\ \|\ \sigma} \tag{6.52}$$

$$\frac{\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}\ \|\ \texttt{skip}}{\sigma\ \|\ \sigma} \tag{6.53}$$

Consider now the **raise** statement. This statement is either written explicitly in the program, or is obtained by a reduction rule in case of an error. For instance, an **if** statement reduces to a **raise** statement if the condition variable is not of type boolean. The effect of the **raise** statement is to skip all statements after it, except a **catch** statement.

$$\frac{\texttt{raise}\ x\ \texttt{end}\ (\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})\ T\ \|\ S_2[X/x]\ T}{\sigma\ \|\ \sigma} \tag{6.54}$$

$$\frac{\texttt{raise}\ x\ \texttt{end}\ S\ T\ \|\ \texttt{raise}\ x\ \texttt{end}\ T}{\sigma\ \|\ \sigma} \quad \text{if } S \text{ is not a } \texttt{catch} \text{ statement} \tag{6.55}$$

This simple model works fine with any number of nested **try** statements, and reflects well that the scope defined by the statement only covers the current threads. An exception in a thread cannot be caught by another thread.

Failed values. Those special values provide a way to transmit exceptions from one thread to another. A failed value y encapsulates an exception x, and is represented in the store by y=*failed*(x). It is created by the operation `FailedValue`.

$$\frac{\texttt{\{FailedValue}\ x\ y\texttt{\}}\ \|\ y\texttt{=}\mathit{failed}(x)}{\sigma\ \|\ \sigma} \tag{6.56}$$

If a statement S needs a failed value, S immediately reduces by raising the exception. The exception is also raised if the statement tries to bind the value.

$$\frac{S \;\|\; \texttt{raise}\ x\ \texttt{end}}{\sigma \;\|\; \sigma} \quad \text{if } y \in \mathit{needed}(S) \text{ and } \sigma \models y{=}\mathit{failed}(x) \tag{6.57}$$

$$\frac{u{=}v \;\|\; \texttt{raise}\ x\ \texttt{end}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \wedge u{=}v \models \mathit{det}(y) \text{ and } \sigma \models y{=}\mathit{failed}(x) \tag{6.58}$$

6.4.3 Read-only views

Oz provides a useful concept for protecting data structures from accidental bindings from the user. This protection allows a user to read a variable without being able to bind it. The idea is to pair two variables x and y by making y a read-only view of x. We write this pairing as $y{=}\mathit{view}(x)$. Such a pair is created by the "bang bang" operator `!!`.

$$\frac{y{=}\texttt{!!}x \;\|\; y{=}z}{\sigma \;\|\; \sigma \wedge z{=}\mathit{view}(x)} \quad \text{where } z \text{ is a fresh variable} \tag{6.59}$$

Once the variable x is determined, being a view of x implies being equal to x. This property is given by the following inference rule. Note that it could be used to drop views from the store, and replace them by equalities: when x is determined, $y{=}\mathit{view}(x)$ is replaced by $y{=}x$.

$$\frac{\sigma \models y{=}\mathit{view}(x) \qquad \sigma \models \mathit{det}(x)}{\sigma \models y{=}x} \tag{6.60}$$

Preventing unification. As read-only views cannot be determined before their variable, we have to strengthen the condition for binding a variable during unification, and make sure that we never bind a read-only view of a variable. The rule (6.36) is rewritten as

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge x{=}t} \quad \text{if} \begin{cases} \sigma \wedge u{=}v \models x{=}t \\ \sigma \not\models \mathit{det}(x), \quad \text{and for all } y,\ \sigma \not\models x{=}\mathit{view}(y) \end{cases} \tag{6.61}$$

Views and by-need synchronization. Read-only views can be used to protect lazy computations, provided that a variable becomes needed when its view is needed. The following inference rule on the store does the job.

$$\frac{\sigma \models y{=}\mathit{view}(x) \qquad \sigma \models \mathit{needed}(y)}{\sigma \models \mathit{needed}(x)} \tag{6.62}$$

Just like a determined variable is needed, one can expect that any attempt to determine a view by unification makes the view needed. Although it does not

exactly fit our definition of needing a variable, we propose the following rule, which makes views needed in case of unification.

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge needed(y)} \quad \text{if} \begin{cases} \sigma \wedge u{=}v \models det(y) \\ \sigma \models y{=}view(x), \quad \text{and } \sigma \not\models needed(y) \end{cases} \tag{6.63}$$

6.4.4 State

All stateful entities can be built on top of cells. The semantics of cells will therefore serve as a reference for all stateful entities with synchronous operations: arrays, dictionaries, etc. Ports can also be built on top of cells. However we will consider a fully asynchronous version of the `Send` operation, which will be given a specific distributed semantics.

A cell is semantically defined as a name associated to a state in the stateful store. If ξ is the name of the cell, the stateful store contains the predicate $\xi{:}x$, where the variable x is the current state of the cell. The creation of a cell consists in creating a name, and adding an initial state for it in the stateful store.

$$\frac{\texttt{\{NewCell}\ x\ c\texttt{\}} \;\|\; c{=}\xi}{\sigma \;\|\; \sigma \wedge \xi{:}x} \quad \text{where } \xi \text{ is a fresh name} \tag{6.64}$$

Just like names, the statement reduces to unifying the cell variable to the name: $c{=}\xi$. This separates the creation of the cell from the binding of the variable c.

Synchronous operations. All synchronous operations can be modeled as a cell *exchange* operation. This operation possibly changes the state of the entity, and returns its former state. The semantics of the operation is given by the following reduction rule.

$$\frac{x{=}c\texttt{:=}y \;\|\; x{=}w}{\sigma \wedge \xi{:}w \;\|\; \sigma \wedge \xi{:}y} \quad \text{if } \sigma \models c{=}\xi \tag{6.65}$$

Asynchronous operations. The operation `Send` on port will serve as a reference for asynchronous operations. Its specificity is that the statement reduction and the state update are not necessarily made together. The statement reduction corresponds to the message begin sent, and the state update to the message being received.

Let us first propose a definition of ports on top of cells. The port is defined as a cell that contains a part of the stream of received messages. This reference to is used to extend the stream as new messages arrive. The stream of messages does not reveal the tail itself, but a read-only view instead. This guarantees that only the port abstraction can add messages to the stream.

```
proc {NewPort S P}
   T in P={NewCell T} S=!!T
end

proc {Send P X}
   T in X|!!T = P := T
end
```

The semantics of `NewPort` is derived from its code. However the semantics given by this definition of `Send` is not satisfactory. This definition is actually synchronous. It works perfectly in a centralized setting. It has the observable property that all messages sent from a given thread are received in the order they were sent. In other words, each thread imposes a partial order on the reception of its own messages. We call this property the *sender ordering*.

Let us propose a fully asynchronous definition of `Send`. The definition below allow messages to arrive in any order. We will use this definition as a reference.

```
proc {Send P X}
   thread T in X|!!T = P := T end
end
```

Let us provide semantic rules that reflect well the semantics of the asynchronous `Send`. The thread created is modeled as a special thread $n \Leftarrow x$, which represents the message being sent. This special thread reduces upon message reception, which adds the message to the message stream.

$$\frac{\texttt{\{Send}\ p\ x\texttt{\}}\ T \;\|\; T, \xi \Leftarrow x}{\sigma \qquad\quad \sigma} \quad \text{if } \sigma \models p{=}\xi \tag{6.66}$$

$$\frac{\xi \Leftarrow x}{\sigma \wedge \xi{:}t \;\|\; \sigma \wedge \xi{:}t' \wedge t{=}x | s \wedge s{=}\mathit{view}(t')} \quad \text{if } s,\ t' \text{ are fresh variables} \tag{6.67}$$

Note that the real semantics of `Send` satisfies the sender ordering property. This property is clearly not satisfied by our semantic rules. It would require to model one message queue per port and per thread. We have chosen to keep the semantics simple, as we believe that this improved model is not significant for the operation itself.

7

Distributed semantics

The operational semantics we give in this chapter *refines* the centralized semantics given in the former chapter. The refinement is defined in the following way: every distributed configuration $\mathcal{D}$ maps to a centralized configuration $\mathcal{C}$, and every distributed reduction $\mathcal{D} \xrightarrow{d} \mathcal{D}'$ maps to a valid centralized reduction $\mathcal{C} \xrightarrow{c} \mathcal{C}'$. The identity reduction, where $\mathcal{C}{=}\mathcal{C}'$, is considered valid. This kind of property is usually visualized by a commutative diagram like

$$\begin{array}{ll} \mathcal{C} \xrightarrow{c} \mathcal{C}' & \text{(centralized)} \\ \uparrow \quad \uparrow & \\ \mathcal{D} \xrightarrow{d} \mathcal{D}' & \text{(distributed)} \end{array}$$

The semantics should reflect some aspects of the distribution, like partial failures and network latency. Those are necessary to reason about the program. The semantics should also reflect the distribution strategy of entities. Not all entities are distributed the same way. For instance, stateful entities allow at least three schemes (centralized, mobile, and replicated). Each strategy should be clearly identifiable in the semantics, but they all must map to the same centralized semantics.

That being said, the semantics should abstract as much as possible the details which are not relevant for the programmer. For instance, we do not describe how failures are detected in practice, but we give conditions that failure detectors must satisfy. Neither do we specify how communication takes place over the network, how data are serialized, or how Oz names are guaranteed unique across machines. Those issues are supposed to be solved for the programmer. What the semantics give are the elements that the programmer must be aware of, like network and site failures, and the elements that are under the programmer's control, like the distribution strategy and failure modes of language entities.

Sections 7.1 and 7.2 define store extensions that reflect the sites, the network, and how entities are distributed among sites. The semantics of `Annotate` is also given there. Sections 7.3 and 7.4 give the distributed semantics for the declarative and nondeclarative parts of the kernel language, respectively. Section 7.5 gives the semantics of the fault stream and the operations `Kill` and `Break`. Section 7.6 gives the mapping from distributed to centralized semantics.

7.1 Reflecting network and site behavior

The basic principle of a distributed semantics is to incorporate some information about the network and sites in the system. The semantics implicitly provides a formal model of the program environment. The advantage is to reflect site and network behavior at the programming language level. So that the programmer can explain or predict the effect of environment changes on his or her program.

7.1.1 Locality

The very first consequence of distributing a program is to introduce a notion of locality. Each site in the system has only a partial view of the whole store. Though the network transparency aims at abstracting this fact, it is essential for reflecting performance and failure issues. We consider that a distributed configuration is like a centralized configuration, where each thread and each store element is tagged with a site identifier. Consider for instance

$$\frac{(y{=}x{+}1\ z{=}y{*}2)_a, (w{=}z{>}100)_b}{(x{=}42)_a \wedge (x{=}42)_c \wedge \ldots}$$

The first thread runs on site a, while the second thread runs on site b. The store contains at least the constraint $x{=}42$, which is present on both sites a and c. Dropping the site index gives an equivalent centralized store.

The local configuration of a site a in the system is simply the restriction of the distributed configuration to the elements indexed by a, that we denote $\mathcal{T}|_a/\sigma|_a$. The operator $|_a$ ("at a") is defined by the following equations. Both β and γ denote predicates, but γ has either a subscript different from a or no subscript (like the predicate $a{\leftrightarrow}b$ defined in the next section).

$$\begin{array}{rlrl} (\mathcal{T},\mathcal{T}')|_a &= \mathcal{T}|_a, \mathcal{T}'|_a & \qquad (\sigma\wedge\sigma')|_a &= \sigma|_a \wedge \sigma'|_a \\ T_a|_a &= T & \beta_a|_a &= \beta \\ T_b|_a &= \emptyset & \gamma|_a &= \top \end{array} \tag{7.1}$$

7.1.2 Network failures

Let us first enrich the store with information about network links. A network link between sites a and b is operational when the predicate $a{\leftrightarrow}b$ is present in

the store. We consider for the sake of simplicity that those links are bidirectional.

$$\frac{\sigma \models a{\leftrightarrow}b}{\sigma \models b{\leftrightarrow}a} \tag{7.2}$$

The rules below temporarily cut network links, and restores them. We consider that those rules are triggered by the system itself. They define valid state transitions for the model of the environment.

$$\frac{}{\sigma \wedge a{\leftrightarrow}b \parallel \sigma} \qquad \frac{}{\sigma \parallel \sigma \wedge a{\leftrightarrow}b} \quad \text{if } \sigma \not\models a{\leftrightarrow}b \tag{7.3}$$

7.1.3 Site failures

The notion of locality above is expressed in terms of site. We now model site failures in the semantics. Remember that a site failure is of the kind *crash-stop*. Its effect is simply to drop the part of the configuration that is specific to that site. It is described by the global reduction rule

$$\frac{\mathcal{T} \parallel \mathcal{T} \downarrow a}{\sigma \parallel \sigma \downarrow a} \tag{7.4}$$

where the operator $\downarrow a$ ("down a") is defined by the following equations, with the same convention as above for γ.

$$\begin{array}{rr} (\mathcal{T}, \mathcal{T}') \downarrow a = (\mathcal{T} \downarrow a), (\mathcal{T}' \downarrow a) & (\sigma \wedge \sigma') \downarrow a = (\sigma' \downarrow a) \wedge (\sigma' \downarrow a) \\ T_a \downarrow a = \emptyset & \beta_a \downarrow a = \top \\ T_b \downarrow a = T_b & \gamma \downarrow a = \gamma \end{array} \tag{7.5}$$

Site failures have no synchronous effect on other sites. Therefore the operators $\downarrow$ and $|$ have the following properties. The first states that a site failure removes everything that is specific to the site, and the second states that other sites are not affected by the failure.

$$(\mathcal{C} \downarrow a)|_a = \top \tag{7.6}$$

$$(\mathcal{C} \downarrow a)|_b = \mathcal{C}|_b \tag{7.7}$$

7.2 Reflecting entity behavior

In order to handle distributed entities, we introduce three extra ingredients that reflect parts of their behavior. Those elements are mostly independent from the type of entity.

7.2.1 Entity failures

In order to reflect entity failures, we introduce the predicate $\mathsf{alive}(e)$ in the predicate store. This predicate is put in the store at the creation of e, and its absence means the permanent failure of e. It may occur at most once per entity in the whole store, and is localized on a given site (its coordination site). Note that it applies to variables as entities, and that $\mathsf{alive}(x)$ is not equivalent to $\mathsf{alive}(\xi)$, even if $x{=}\xi$ in the store. The principle of substitution by equals does not apply to the predicate alive.

Remember that each site maintains a current fault state for each entity in the system. We assume that on every site a, the store entails one equality like $(\mathsf{fstate}(e){=}s)_a$, which states that a considers entity e to be in fault state s. The precise definition of how the store entails that fact and modifies it, is given in Section 7.5. The principle of substitution by equals does not apply to fstate.

An entity e is *correct* if and only if the distributed store contains $\mathsf{alive}(e)_a$ for some site a. Removing the predicate automatically makes e permanently failed. For a site b to perform an operation on e, we will require e to be correct, accessible, and not locally failed on b. Note that this condition is necessary but not sufficient. In order to abstract a bit this condition, we introduce the predicate $\mathsf{correct}$ on b, which is defined by the following equation.

$$\mathsf{correct}(e,a)_b \equiv \mathsf{alive}(e)_a \wedge a{\leftrightarrow}b \wedge (\mathsf{fstate}(e){=}\mathsf{ok})_b \tag{7.8}$$

Let us comment a bit on each of the conditions required by $\mathsf{correct}(e,a)_b$.

- The failure of site a causes the predicate $\mathsf{alive}(e)_a$ to be dropped from the store; this effectively prevents any further operation on e that would require it to be correct. This property is enough to model the blocking behavior of operations on failed entities.

- The second condition states that site b must be able to communicate with site a. This is because making a consistent update of the entity generally requires some synchronization with the coordination site of the entity.

- The third condition states that b considers e to be in fault state ok. This means that b must be consistent with itself. This is necessary, as b may consider e to be locally failed.

7.2.2 Entity annotations

Some entities have several alternatives for their distributed semantics. Which alternative is used depends on an entity's *annotation*. Every entity annotation is visible in the store as a predicate, like $\mathsf{stationary}(e)$. Note that the predicate is not localized to a site, which means that any site referring to the entity should know its annotations.

Let us consider protocol annotations. We define the predicates stationary, migratory, replicated, variable, reply, immediate, eager, lazy. They are idempotent and mutually inconsistent for a given entity. For instance, we have

$$\begin{aligned} \mathsf{variable}(x) \wedge \mathsf{variable}(x) &\equiv \mathsf{variable}(x) \\ \mathsf{stationary}(\xi) \wedge \mathsf{replicated}(\xi) &\equiv \bot \end{aligned}$$

Access architecture annotations are defined in a similar way. Reference consistency protocol annotations as well, except that an annotation specifies a subset of the provided protocols. Note also that annotations may be inconsistent if they are applied to the wrong type of entity. A cell cannot be annotated with variable, for instance.

We also use the predicate $\mathsf{annot}(e, v)$ as an alternative notation to state that entity e is annotated with v. Each predicate mentioned above corresponds to exactly one value v:

$$\begin{aligned} \mathsf{annot}(e, \texttt{stationary}) &\equiv \mathsf{stationary}(e) \\ \mathsf{annot}(e, \texttt{migratory}) &\equiv \mathsf{migratory}(e) \\ &\vdots \\ \mathsf{annot}(e, \texttt{lazy}) &\equiv \mathsf{lazy}(e) \end{aligned}$$

Setting annotations. Here we define how annotations are set on entities. The first rule tells the annotation of the entity to the store. There is a similar rule, which we don't mention here, that raises an exception if the annotation is inconsistent.

$$\frac{(\{\texttt{Annotate}\ e\ t\})_a \;\|\; (\textbf{skip})_a}{\sigma \;\|\; \sigma \wedge \mathsf{annot}(e, v)} \quad \text{if} \begin{cases} \sigma \models \mathsf{alive}(e)_a \\ \sigma|_a \models t{=}v \text{ for a value } v \\ \sigma \wedge \mathsf{annot}(e, v) \text{ is consistent} \end{cases} \tag{7.9}$$

The second rule defines the effect of a *default* annotation: if the entity is shared by more than one site and no annotation was specified, a default one is picked at the home site of the entity.

$$\frac{}{\sigma \;\|\; \sigma \wedge \mathsf{annot}(e, v)} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } e \\ \sigma \models \mathsf{alive}(e)_a \\ v \text{ is default for } e \text{ on } a \\ \sigma \wedge \mathsf{annot}(e, v) \text{ is consistent} \end{cases} \tag{7.10}$$

The condition "$\sigma|_b$ refers to e" means that e occurs in a predicate or an equality in the store $\sigma|_b$.

7.3 Declarative kernel language

We now give the distributed semantics of language statements.

7.3.1 Purely local reductions

The rules that do not modify the store require very small adaptation to the distributed case. Basically the threads must be localized on a site, and the condition must be evaluated with the local store $\sigma|_a$, where a is the site of the reduced thread.

This is the case for the sequential and concurrent composition. Notice that the **thread** statement creates a thread on the site where the statement reduces. The conditional statements and procedure application are also extended in this way.

7.3.2 Variable introduction and binding

The rules that introduce and bind variables require an extra adaptation. The creation of a variable x on a site a automatically introduces the predicate $\mathsf{alive}(x)_a$ in the store:

$$\frac{(\mathbf{local}\ X\ \mathbf{in}\ S\ \mathbf{end})_a \;\|\; (S[X/x])_a}{\sigma \;\|\; \sigma \wedge \mathsf{alive}(x)_a} \quad x \text{ fresh variable} \tag{7.11}$$

This predicate is necessary for binding the variable, as shown in the rules below. The first rule binds x on its coordination site first, while the second rule is responsible for propagating the basic constraint from the coordination site to the other sites. Note that the binding $x{=}t$ proposed by site b depends on its local store $\sigma|_b$.

$$\frac{(u{=}v)_b \;\|\; (u{=}v)_b}{\sigma \;\|\; \sigma \wedge (x{=}t)_a} \quad \text{if} \begin{cases} \sigma|_b \wedge u{=}v \models x{=}t \\ \sigma|_a \not\models det(x) \\ \sigma \models \mathsf{correct}(x,a)_b \end{cases} \tag{7.12}$$

$$\frac{\quad\|\quad}{\sigma \wedge (x{=}t)_a \;\|\; \sigma \wedge (x{=}t)_a \wedge (x{=}t)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } x \\ \sigma|_b \not\models x{=}t \\ \sigma \models \mathsf{correct}(x,a)_b \end{cases} \tag{7.13}$$

Notice how the latter rule propagates references to t on all sites b that refer to the variable x.

7.3.3 Procedure creation and copying

Procedures do not require much adaptation, since they are values. However, we should model the copying of the value from site to site. The value is copied at

most once per site, which keeps the local stores consistent. In the rules below, P is the abstraction $\lambda X_1 \ldots X_n.S$.

$$\frac{(\textbf{proc}\ \{p\ X_1 \ldots X_n\}\ S\ \textbf{end})_a \quad\|\quad (p{=}\xi)_a}{\sigma \quad\|\quad \sigma \wedge (\xi{:}P)_a} \quad \xi \text{ fresh name} \tag{7.14}$$

$$\frac{\qquad\qquad\|\qquad\qquad}{\sigma \wedge (\xi{:}P)_a \quad\|\quad \sigma \wedge (\xi{:}P)_a \wedge (\xi{:}P)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } \xi \\ \sigma \models \mathsf{eager}(\xi) \wedge a{\leftrightarrow}b \\ \sigma \not\models (\xi{:}P)_b \end{cases} \tag{7.15}$$

The latter rule propagates the abstraction P on all sites that refer to ξ. As a consequence, all the references in the procedure body S are also propagated on those sites.

Lazy copying. If the procedure is annotated with `lazy`, the copy of the abstraction should be done lazily. The reduction rule for copying is similar to the one above, except that it should be reducible only when it is needed on site b, i.e., when a thread on b tries to call it.

$$\frac{S_b \quad\|\quad S_b}{\sigma \wedge (\xi{:}P)_a \quad\|\quad \sigma \wedge (\xi{:}P)_a \wedge (\xi{:}P)_b} \quad \text{if} \begin{cases} p \in \mathit{needed}(S), \quad \sigma|_b \models p{=}\xi \\ \sigma \models \mathsf{lazy}(\xi) \wedge a{\leftrightarrow}b \\ \sigma \not\models (\xi{:}P)_b \end{cases} \tag{7.16}$$

7.3.4 By-need synchronization

This mechanism is easy to extend in the distributed case, since making variable x needed consists in telling the constraint $\mathit{needed}(x)$ everywhere in the system. The two existing rules are extended as purely local rules, such that $\mathit{needed}(x)$ is told and asked locally. The additional rules below propagate the predicate $\mathit{needed}(x)$ to all sites via the coordination site, provided the variable is correct.

$$\frac{\qquad\|\qquad}{\sigma \quad\|\quad \sigma \wedge \mathit{needed}(x)_a} \quad \text{if} \begin{cases} \sigma \models \mathsf{correct}(x,a)_b \wedge \mathit{needed}(x)_b \\ \sigma \not\models \mathit{needed}(x)_a \end{cases} \tag{7.17}$$

$$\frac{\qquad\|\qquad}{\sigma \quad\|\quad \sigma \wedge \mathit{needed}(x)_b} \quad \text{if} \begin{cases} \sigma \models \mathsf{correct}(x,a)_b \wedge \mathit{needed}(x)_a \\ \sigma \not\models \mathit{needed}(x)_b \end{cases} \tag{7.18}$$

7.4 Nondeclarative extensions

We now complete the distributed semantics of Oz by extending the semantics of nondeclarative language features to the distributed case. Like in the centralized case, those features admit a semantics that is mostly compositional with respect to the declarative part of the language.

7.4.1 Exception handling and read-only views

The exception mechanism interacts with thread execution. Its reduction rules are extended to purely local reductions. Failed values are also reduced locally, and are copied from site to site just like ordinary values.

Read-only views are also handled locally. The basic constraint $x{=}\mathit{view}(y)$ is handled like an ordinary binding, and copied on all sites that refer to x. The binding rule (7.12) is extended with the extra condition

$$\text{for all } y,\ \sigma|_a \not\models x{=}\mathit{view}(y).$$

7.4.2 State

Stateful entities are somewhat richer when it comes to their distribution. As we have seen already, several strategies are possible for maintaining their state. We consider three strategies here, and each has its own semantics: stationary state, migratory state, and replicated state. The properties of those strategies have been discussed in Chapter 3. Our concern in this chapter is that all variants are refinements of the centralized semantics.

Let us first extend the cell creation semantics. The new reduction rule is pretty straightforward: we locate the state on the creation site, together with the predicate $\mathsf{alive}(\xi)$. The cell is distributed once two sites at least refer to the name ξ.

$$\frac{(\{\texttt{NewCell}\ x\ c\})_a \quad\Big\|\quad (c{=}\xi)_a}{\sigma \quad\Big\|\quad \sigma \wedge \mathsf{alive}(\xi)_a \wedge (\xi{:}x)_a} \quad n \text{ fresh name} \tag{7.19}$$

Before we go into the details of the state operations, we have to describe how the state "becomes" distributed among sites. In the case of stationary or migratory state, nothing special is needed. The state is already present on the right site. Replicated state needs some extra support for distributing the state. All we need is one rule that copies the state from its home site a to every other site b that refers to the entity.

$$\frac{}{\sigma \wedge (\xi{:}x)_a \quad\Big\|\quad \sigma \wedge (\xi{:}x)_a \wedge (\xi{:}x)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } \xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{replicated}(\xi) \\ \sigma \not\models (\xi{:}x)_b \end{cases} \tag{7.20}$$

Synchronous operations. We now give three semantic rules for the cell exchange operation, each rule reflecting the cell's possible distribution strategy. The first rule shows a stationary cell: the state is located at site a. The operation is performed at a.

The second rule shows a migratory cell. The operation reduces once the state move to b, coming from another site b'. The semantics does not tell how b and b' are chosen, this is left to the actual protocol. But there is an interesting

case when $b = b'$: the state remains on b and can be updated without any network operation. This is the "caching" behavior of the migratory state.

The third rule shows a cell whose state is replicated on several sites. In the rule, the symbol $*$ denotes the set of sites that have a copy of the cell's state. One can see that updating the state is costly: it requires the home site of the cell to communicate with all the state replicas. However, if an operation does not change the state, it can be performed on the local copy of the state.

$$\frac{(x{=}c:{=}y)_b \parallel (x{=}w)_b}{\sigma \wedge (\xi{:}w)_a \parallel \sigma \wedge (\xi{:}y)_a} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{stationary}(\xi) \end{cases} \tag{7.21}$$

$$\frac{(x{=}c:{=}y)_b \parallel (x{=}w)_b}{\sigma \wedge (\xi{:}w)_{b'} \parallel \sigma \wedge (\xi{:}y)_b} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{migratory}(\xi) \\ \sigma \models b'{\leftrightarrow}b \end{cases} \tag{7.22}$$

$$\frac{(x{=}c:{=}y)_b \parallel (x{=}w)_b}{\sigma \wedge (\xi{:}w)_* \parallel \sigma \wedge (\xi{:}y)_*} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{replicated}(\xi) \\ \sigma \models a{\leftrightarrow}* \end{cases} \tag{7.23}$$

All those rules have an interesting special case. If site b has the state and performs a read operation, none of the other sites is affected, and the state can be read locally. The only condition is that the local fault state of the entity must be ok. For this case all rules can be simplified to

$$\frac{(x{=}@c)_b \parallel (x{=}w)_b}{\sigma \wedge (\xi{:}w)_b \parallel \sigma \wedge (\xi{:}w)_b} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models (\mathsf{fstate}(\xi){=}\mathsf{ok})_b \end{cases} \tag{7.24}$$

Asynchronous operations. Just like in the centralized semantics, the Send operation reduces immediately by sending a message $p{\Leftarrow}x$.

$$\frac{(\{\mathsf{Send}\ p\ x\}\ T)_b \parallel T_b, (\xi{\Leftarrow}x)_b}{\sigma \qquad\qquad \sigma} \quad \text{if } \sigma|_b \models p{=}\xi \tag{7.25}$$

The first rule below shows the case of the stationary port. The message is received by the coordination site of the entity. Once delivered, the operation is performed, and the binding of the stream is visible globally. The second rule considers a mobile port. In that case, the message is kept locally until the state comes at the site; the operation is then performed locally. The latter rule considers a replicated port. All the copies of the state are atomically changed.

$$\frac{(\xi{\Leftarrow}x)_b \parallel}{\sigma \wedge (\xi{:}t)_a \parallel \begin{array}{l} \sigma \wedge (t{=}x|t')_a \wedge (\xi{:}t')_a \wedge \\ \mathsf{alive}(t')_a \wedge (t'{=}\mathit{view}())_a \end{array}} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{stationary}(\xi) \end{cases} \tag{7.26}$$

$$\frac{(\xi{\Leftarrow}x)_{b'} \;\|}{\sigma \wedge (\xi{:}t)_b \;\|\; \sigma \wedge (t{=}x|t')_b \wedge (\xi{:}t')_{b'} \wedge \mathsf{alive}(t')_{b'} \wedge (t'{=}\mathit{view}())_{b'}} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{migratory}(\xi) \\ \sigma \models \mathsf{correct}(t, b)_{b'} \end{cases} \tag{7.27}$$

$$\frac{(\xi{\Leftarrow}x)_{b} \;\|}{\sigma \wedge (\xi{:}t)_* \;\|\; \sigma \wedge (t{=}x|t')_a \wedge (\xi{:}t')_* \wedge \mathsf{alive}(t')_a \wedge (t'{=}\mathit{view}())_a} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{replicated}(\xi) \\ \sigma \models a{\leftrightarrow}* \end{cases} \tag{7.28}$$

State failure. An important property of cells is that they fail once their state is lost. In other words, if the store σ has no occurrence of a state predicate $\xi{:}z$ anywhere, the cell ξ must fail. In both the stationary and replicated protocols, this situation follows from the failure of the coordinator site of the cell. But in the migratory protocol, we have to add a specific rule that removes the predicate $\mathsf{alive}(\xi)_a$ from the store:

$$\frac{\|}{\sigma \wedge \mathsf{alive}(\xi)_a \;\|\; \sigma} \quad \text{if } \sigma \not\models (\xi{:}z)_b \text{ for all } z \text{ and } b \tag{7.29}$$

7.5 Failure handling

7.5.1 Failure detectors

We provide a generic rule that reflects how the system may update the fault stream of an entity on a site. Upon creation, every entity e in the system has a fault stream on each site a, which is described in the store as the predicate $(\mathsf{fs}(e){=}s|t)_a$, where s is the current fault state of e, and t is the tail of the stream. The latter is a read-only view, but for the sake of simplicity we will treat it as a plain logic variable. The programmer can access it by calling `GetFaultStream`:

$$\frac{(\{\texttt{GetFaultStream}\ e\ x\})_a \;\|\; (x{=}s|t)_a}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models (\mathsf{fs}(e){=}s|t)_a \tag{7.30}$$

The fault stream of e on a is updated by the rule

$$\frac{\|}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \;\|\; \sigma \wedge (\mathsf{fs}(e){=}t)_a \wedge \mathsf{alive}(t')_a \wedge (t{=}s'|t')_a} \quad \text{if} \begin{cases} t' \text{ fresh variable} \\ \mathit{cond}(s, s') \end{cases} \tag{7.31}$$

where the condition $cond(s, s')$ is defined below. The site h is the coordination site of e (its home site). Note that s in (7.34) must be different from `permFail`.

$$cond(\texttt{ok}, \texttt{tempFail}) \quad \text{iff} \quad \sigma \not\models \mathsf{alive}(e)_h \quad \text{or} \quad \sigma \not\models a{\leftrightarrow}h \tag{7.32}$$

$$cond(\texttt{tempFail}, \texttt{ok}) \quad \text{iff} \quad \sigma \models \mathsf{alive}(e)_h \quad \text{and} \quad \sigma \models a{\leftrightarrow}h \tag{7.33}$$

$$cond(s, \texttt{permFail}) \quad \text{iff} \quad \sigma \not\models \mathsf{alive}(e)_h \quad \text{and} \quad \sigma \models a{\leftrightarrow}h \tag{7.34}$$

$$cond(s, s') \quad \text{is false otherwise} \tag{7.35}$$

As you can see in the first condition, a temporary failure for an entity e can be reported on a site a when either the entity actually failed, or the network link between a and h is down. The third condition states that detecting the permanent failure of an entity requires site a to be able to reach the home site of the entity. This condition is not fulfilled in general if the site h has crashed. However, it can happen in certain cases, for instance if sites a and h are on the same local area network (LAN). The operating system may report the crash of the process corresponding to site h.

The current fault state of an entity on a given site, as it is defined in Section 7.2 on page 95, is derived from its fault stream on that site. We define it with the following inference rule.

$$\frac{\sigma \models (\mathsf{fs}(e){=}s|t)_a}{\sigma \models (\mathsf{fstate}(e){=}s)_a} \tag{7.36}$$

Fault stream of variables As stated in Section 4.2.3, the fault streams of unified variables are merged. In order to define which one is bound to the other, we assume that all variables in the system are ordered by a relation $\prec$. This order is used by the system, but not directly available to the programmer itself. So the semantic rules below give the possible ways to merge. The last rule finalizes a variable's fault stream when the variable is determined.

$$\frac{\sigma \wedge (\mathsf{fs}(x){=}s|t)_a \wedge (\mathsf{fs}(y){=}s'|t')_a}{\sigma \wedge t{=}t' \wedge (\mathsf{fs}(y){=}s'|t')_a} \quad \text{if } \sigma|_a \models x{=}y \wedge x{\prec}y \wedge s{=}s' \tag{7.37}$$

$$\frac{\sigma \wedge (\mathsf{fs}(x){=}s|t)_a \wedge (\mathsf{fs}(y){=}s'|t')_a}{\sigma \wedge t{=}s'|t' \wedge (\mathsf{fs}(y){=}s'|t')_a} \quad \text{if } \sigma|_a \models x{=}y \wedge x{\prec}y \wedge s{\neq}s' \tag{7.38}$$

$$\frac{\sigma \wedge (\mathsf{fs}(x){=}s|t)_a}{\sigma \wedge t{=}\texttt{nil}} \quad \text{if } \sigma|_a \models det(x) \tag{7.39}$$

Creation and finalization of the fault stream. Assuming that every site maintains a fault stream for every entity in the system may be misleading. Indeed, this does not take into account the fact that a site may forget some information about an entity (see the discussion of Section 4.4 on page 54). So we propose the following two rules for creating and finalizing the fault stream

of an entity e on a site a. The concept of *liveness* of an entity is the usual one used by garbage collectors. For the sake of conciseness, we skip its formal definition.

$$\frac{\mathcal{T} \;\|\; \mathcal{T}}{\sigma \;\|\; \sigma \wedge (\mathsf{fs}(e){=}\mathtt{ok}|t)_a} \quad \text{if} \begin{cases} e \text{ is alive on } \mathcal{T}|_a/\sigma|_a \\ \sigma \not\models (\mathsf{fs}(e){=}\ldots)_a \\ t \text{ is a fresh variable} \end{cases} \tag{7.40}$$

$$\frac{\mathcal{T} \;\|\; \mathcal{T}}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \;\|\; \sigma \wedge t{=}\mathtt{nil}} \quad \text{if } e \text{ is not alive in } \mathcal{T}|_a/\sigma|_a \tag{7.41}$$

7.5.2 Making entities fail

In this section, we define the operations `Kill` and `Break`.

Global failure. The operation `Kill` should make its argument fail, i.e., it should remove the predicate $\mathsf{alive}(x)_a$ from the store, where x is the argument of the call. As the operation is asynchronous, we use a "kill" message $x{\Leftarrow}\dagger$ that is similar to the messages used in Section 7.4.2.

$$\frac{(\{\mathtt{Kill}\ x\}\ T)_b \;\|\; T_b, (x{\Leftarrow}\dagger)_b}{\sigma \;\|\; \sigma} \tag{7.42}$$

$$\frac{(x{\Leftarrow}\dagger)_b \;\|\;}{\sigma \wedge \mathsf{alive}(x)_a \;\|\; \sigma} \quad \text{if } \sigma \models a{\leftrightarrow}b \tag{7.43}$$

Notice that the latter rule states explicitly that communication with the coordinator site of x is necessary to make x permanently failed.

Local failure. The procedure `Break` is pretty easy to define. Its effect is to change the fault state of the entity to `localFail`, unless the fault state already has that value or `permFail`.

$$\frac{(\{\mathtt{Break}\ x\})_a \;\|\; (\mathbf{skip})_a}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \;\|\; \begin{array}{c}\sigma \wedge (\mathsf{fs}(e){=}t)_a \wedge \mathsf{alive}(t')_a \\ \wedge\ (t{=}\mathtt{localFail}|t')_a\end{array}} \quad \begin{array}{c}\text{if } \sigma \models s{=}\mathtt{ok} \text{ or} \\ \sigma \models s{=}\mathtt{tempFail}\end{array} \tag{7.44}$$

$$\frac{(\{\mathtt{Break}\ x\})_a \;\|\; (\mathbf{skip})_a}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \;\|\; \sigma \wedge (\mathsf{fs}(e){=}s|t)_a} \quad \begin{array}{c}\text{if } \sigma \models s{=}\mathtt{localFail} \\ \text{or } \sigma \models s{=}\mathtt{permFail}\end{array} \tag{7.45}$$

If site b has locally killed an entity e, the predicate $\mathsf{correct}(e, a)_b$ will never be entailed by the store. Hence, all operations that require e to be correct on b block forever.

7.6 Mapping distributed to centralized configurations

The definition we gave of a refinement on page 93 states that every distributed configuration maps to a centralized configuration. This section defines precisely that mapping.

7.6.1 The mapping

The mapping itself is pretty easy to define. Basically we collect the configurations of all sites in the system. We define it in terms of the operator $|_a$. The disjoint union and conjunction operators range on the set of all sites in the system.

$$\mathit{centralized}\left(\frac{\mathcal{T}}{\sigma}\right) = \frac{\uplus_a \mathcal{T}|_a}{\wedge_a \sigma|_a} \tag{7.46}$$

There is only a small issue with the conjunction of the local stores. They are always consistent with each other, except for the entities' fault streams, which can be in different states. However, we can consider that a centralized configuration possibly has several fault streams for a given entity. This looks like the centralized system maintains many failure detectors for each entity, which can have different views.

7.6.2 Network transparency

With this definition we can now formulate a theorem which relates the distributed and centralized semantics of the language Oz. As this property translates the network transparency at the semantic level, we call it the *Network transparency theorem*. The theorem can be proven by induction on every distributed reduction rule.

Theorem (Network transparency)**.** The distributed semantics of the language is a refinement of its centralized semantics. In other words, for every pair of distributed configurations $\mathcal{D}$ and $\mathcal{D}'$, if $\mathcal{D} \to \mathcal{D}'$ is a valid distributed reduction, then $\mathit{centralized}(\mathcal{D}) \to \mathit{centralized}(\mathcal{D}')$ is a valid centralized reduction.

8

Implementation

This chapter describes the new implementation of the distribution of Oz, based on the Distribution Subsystem (DSS). This work has been achieved by several developers, among which Erik Klintskog, Zacharias El Banna, Boris Mejías, and myself. In order to make a clear distinction between the former and new implementations, we will refer to them as "Mozart" and "Mozart/DSS", respectively.

In Section 8.1, we explain the architecture, and some principles underlying Mozart/DSS. We show there that the implementation splits up into three distinct layers. The topmost layer is the virtual machine, which has been modified as little as possible in order to take distribution into account. Section 8.2 describes the bottom layer, which provides all the distributed protocols. Section 8.3 describes the middle layer, also known as the Glue, that interfaces the virtual machine to the DSS layer.

8.1 Architecture of Mozart/DSS

The platform Mozart/DSS is a new implementation of the distribution of Oz, based on the virtual machine of Mozart and the library DSS. The latter provides abstractions for distributing programming language entities. The general architecture for a distributed entity is depicted in Figure 8.1 on the following page. The diagrams shows the fundamental components implementing an entity that is shared among three sites. The components are divided up into sites, separated by bold vertical lines, and into implementation layers, separated by dashed horizontal lines.

All language entities, local or distributed, are stored in virtual machine heaps. A distributed entity can be seen as a set of local entities connected together via the network, and cooperating in order to provide the illusion of a single global entity to the programmer. Note that an entity in the heap might

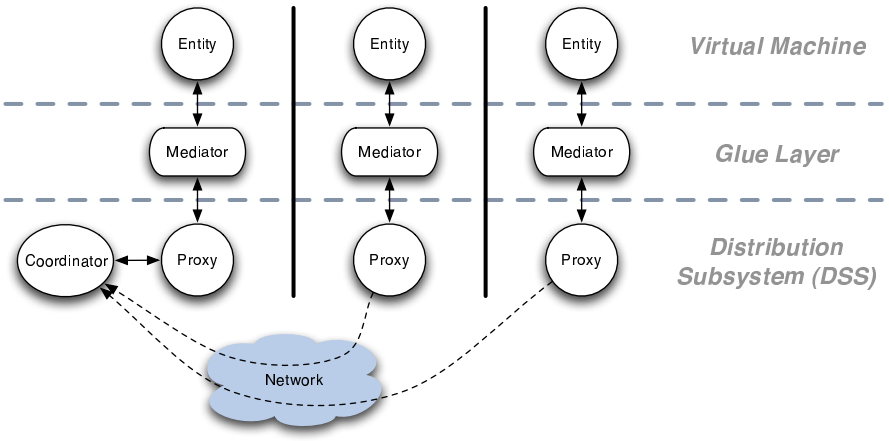


Figure 8.1: The three layers that implement the distribution in Mozart/DSS

be only a *stub*, i.e., the state of the entity is not available locally. In that case, it is necessary for that site to cooperate with other sites in order to complete a language operation.

A distributed entity has a special hook that connects it to a *proxy* in the DSS library. The proxies of a given entity are connected together with a *coordinator* via network links. The coordinator and proxies form the *coordination network* of the entity, which has a unique global identity. It is used to identify the language entity across sites boundaries. The coordination network implements the *access architecture* of the entity, which we introduced in Section 3.3.4.

The role of each layer. Each layer in the implementation plays a specific role. The virtual machine layer implements the entity's centralized semantics. The DSS layer provides global naming for entities, a general serialization mechanism for user data, a set of selectable protocols implementing generic entity operations, a distributed garbage collector with several protocols available, and failure detectors for sites and entities.

The Glue layer implements the distributed semantics of entity operations by mapping them to DSS entity operations. It implements the failure semantics of entities, and makes both the local and distributed garbage collectors cooperate. It also provides network communication channels for the DSS.

The author's contributions. A prototype of the Glue layer was given to us by Erik Klintskog. We revised its design, and implemented it entirely, with a little help from our colleage Boris Mejías. Together with Boris we modified the Mozart marshaler to serialize and deserialize entities using the DSS. The new language features, like annotations, the fault stream, and the operations

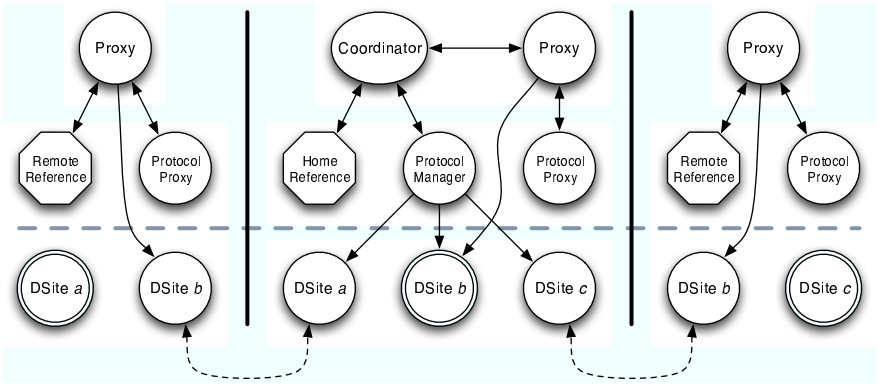


Figure 8.2: Architecture of the DSS, shown for one entity shared among three sites a, b, and c

`Kill` and `Break`, were implemented solely by the author. We rewrote all the entity protocols in the DSS such that they could handle partial failures. We also extended the DSS interface to handle and reflect entity failures.

We made a few contributions to the virtual machine, too. Together with Fred Spiessens, we implemented our new design of the by-need synchronization mechanism. We also had to adapt the virtual machine in order to handle the new distribution of procedures and object, among others. Finally, we improved the unification's implementation to make it more incremental.

8.2 The Distribution Subsystem

The DSS library provides a set of protocols for distributing programming language entities [Kli05]. It is itself split into a *protocol layer*, and a *messaging layer*. The protocol layer brings together all the protocols to manage distributed entities. Protocols are partitioned in three classes, that handle orthogonal aspects of an entity's distribution.

- *Coordination protocols* provide an entity's unique identity, and maintain its coordination network. They let proxies and their coordinators reach each other by message passing.
- *Entity protocols*, or *consistency protocols*, implement generic entity operations. They also handle partial failures of entities.
- *Reference protocols* implement distributed garbage collection policies for shared language entities.

Figure 8.2 shows the main DSS components for one entity shared among three sites a, b, and c. The sites are separated by the vertical bold lines, and

the horizontal dashed gray line splits up both layers. One can see that each site has a proxy for the entity, and site b owns its coordinator; those components implement the coordination protocol of the entity. Each proxy is connected to a *protocol proxy*, while the coordinator has a *protocol manager*; protocol proxies and manager implement the consistency protocol of the entity. Finally, the coordinator owns a *home reference*, and the remote proxies have a *remote reference*; those components implement the distributed garbage collection protocol of the entity.

Some of those components, like a proxy with its protocol proxy, may call each other directly, but they interact more generally via the messaging layer. The latter provides a channel-based communication mechanism (reliable and ordered message passing) between sites. Each site is abstracted by a *DSite* component, which hides the communication channel, and reflects the fault status of the given site. The DSS on each site maintains a set of known sites, as it is shown in Figure 8.2 on the preceding page. Each site also has a representation for itself, drawn with double lines in the figure.

Each component in the protocol layer can be addressed with its type (proxy, coordinator, protocol proxy, protocol manager, or reference), the global identity of its coordination network, and a DSite. For instance, the protocol proxy on site c can easily send a message to its protocol manager via its proxy, which knows the global identity and the DSite of its coordinator. The protocol manager on site b can send a message to its protocol proxy on site a, with its own reference to the DSite of a, and the global identity of its coordinator. This facility greatly simplifies the implementation of the protocols.

By design, the coordination network provides a mean for each proxy to send messages to its coordinator. In case the coordinator is stationary, each proxy just has to know the coordinator's site. All the DSS protocols are built on top of this architecture. By default the protocol manager does not know its proxies, so in some cases it maintains a list of DSite references corresponding to its proxies. This list is built either explicitly by making proxies register to their manager, or implicitly by collecting message origins. The latter case uses the fact that a message is always delivered together with the DSite representing the sender's site.

8.2.1 Protocols for mutables

Those protocols implement three operations, namely *read*, *write*, and *send*. The operations read and write are synchronous, and may return a result. The send operation is similar to an asynchronous version of a write, and does not return any value. There are essentially four protocols available in this category: the stationary state protocol, the migratory state protocol, the pilgrim protocol, and the invalidation protocol. The author made significant contributions to the last three ones.

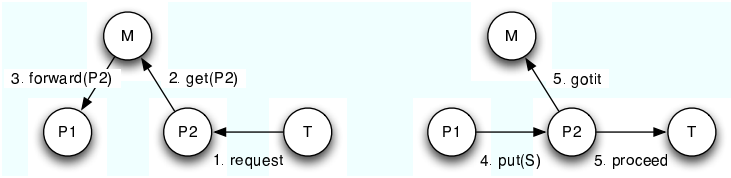


Figure 8.3: Basic migratory state protocol

The stationary state protocol

This protocol is the simplest of all. Remote proxies send their operation requests to their protocol manager, which performs the operation. If the operation is not a send, it returns a result message once the operation on the entity has completed. The result message allows the proxy to resume the corresponding suspended operation on its site.

The migratory state protocol

This protocol was first described in [HVS97, VHB+97, HVBS98], then extended in [VBHC99] to make it handle permanent failures. The author proposed a formalization of the extended protocol, and proved it correct in [BVCK00]. Mozart used that protocol for distributed cells and objects.

The protocol uses a token which is passed between the proxies in the coordination network. The proxy holding the token has sole access to the state of the distributed language entity, and the entity's state is passed together with the token. The migration of the token is shown in Figure 8.3. The protocol manager M builds a forwarding chain with all the proxies requesting an operation. When a proxy P2 needs the state of the entity to perform an operation for thread T, it sends a message `get(P2)` to M. The latter then sends a message `forward(P2)` to the last proxy in the forwarding chain. This message will make P1 forward the state token to P2, so that P2 becomes the last proxy in the forwarding chain. When P2 receives the state token, it sends a message `gotit` to its manager. This message allows M to maintain a list of the proxies that could hold the state token.

Bypassing failed proxies. This simple extension to the basic protocol allows a proxy to avoid sending the state token to a failed proxy. This situation is depicted in Figure 8.4. The proxy P1 has detected that its successor P2 in the chain is permanently failed. In order to find the next successor, it notifies its manager. The manager M can then send a new message `forward()`, because M owns a representation of the forwarding chain.

State loss detection. The state token may be lost either if a proxy holds it and crashes, or it has been sent over the network in a message `put`, and the

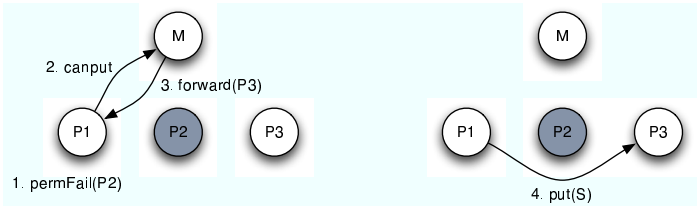


Figure 8.4: Bypassing a failed proxy

message is lost because of a site failure (of the sender or the receiver). When the manager detects that a proxy in the chain has permanently failed, it runs an inquiry protocol, which determines whether the state token has been lost. The manager asks each proxy where the state token is. The proxy can answer `beforeMe`, `atMe`, or `afterMe`. If the manager finds two proxies that answer `afterMe` and `beforeMe`, all proxies between them have crashed, and there is nothing in the network, then the entity state is lost. The permanent failure of the entity is notified to all proxies.

The pilgrim protocol

This mobile state protocol is inspired by the work in [GLT97]. It can be seen as a variant of the migratory token protocol, where the proxies accessing the token form a ring instead of a chain. Each proxy in the ring has a successor, to which it forwards the token. A proxy remains in the ring unless it has not performed any entity operation for a certain period of time. The proxies interact with the manager only to enter or leave the ring. This greatly reduces the interaction with the manager when a set of proxies regularly access the token.

The author made a significant contribution to make that mobile state protocol handle failures. The original implementation, as provided by [Kli05], had no simple way to detect whether the state token was lost. Moreover, proxy insertions and removals in the ring were serialized at the manager. This implied a strong protocol invariant which was relied upon for garbage collection, because it allowed proxies to know whether they were inside the ring, and consequently whether they had to be kept alive. Proxies inside the ring should not be removed by their respective garbage collectors, since their removal would create a gap in which the state token can be lost. But determining efficiently when a proxy is no longer accessible from the ring can be tricky, in particular when ring proxies crash.

In order to remove any dependency of the manager on its proxies, we have simplified the proxy insertion and removal in the ring. The result is shown in Figure 8.5. Dashed arrows represent the successor relation in the ring. When the manager receives a request from a proxy P to enter the ring, it chooses two consecutive proxies P1 and P2 in the ring. It then sends a message to P1 to make its successor P, and another message to P to make its successor P2, so

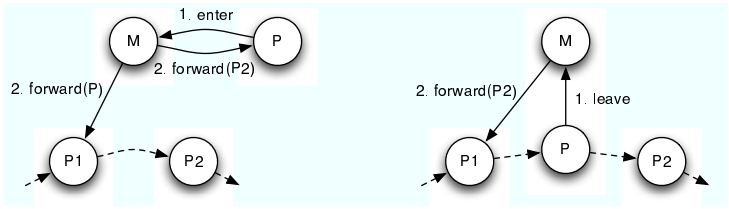


Figure 8.5: Pilgrim: entering and leaving the ring

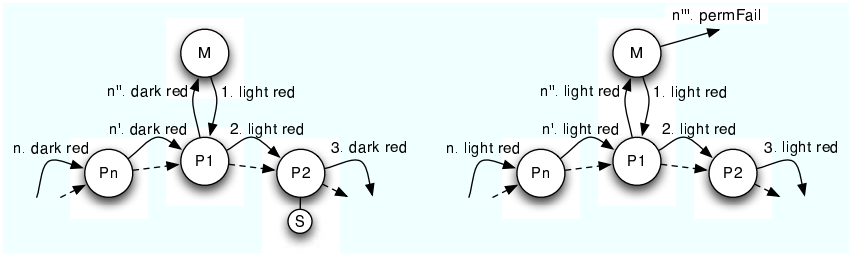


Figure 8.6: Pilgrim: ring coloring

that P is eventually between P1 and P2 in the ring. Proxy removal is even simpler: its ring predecessor is sent a new successor. The simplified protocol allows the manager to remove any suspect proxy from the ring without that proxy's cooperation. To compensate for the apparent sloppiness of the protocol, we have added an orthogonal protocol that both detects the loss of the state token, and solves the garbage collection issue.

Ring coloring. The idea of the coloring protocol is to mark the proxies that are inside the ring, following the ring structure. The proxies that have not been marked during the process are guaranteed to be unreachable from the ring, and can therefore be safely removed. Moreover, the protocol is able to detect whether the coloring token or a newly marked proxy has encountered the state token. At the end of the coloring, the manager checks this information, and notifies all proxies if the state token is no longer present.

More than two colors are necessary to make the protocol robust. The coloring can be interrupted at any moment by a failed proxy, and several color changes can be performed concurrently. Eventually all proxies will change to the most recent color.

Figure 8.6 shows how the coloring protocol works. The protocol attaches a color to every proxy and the state token, and the color is either "light" or "dark". A proxy always attaches its own color to the state token. When the manager initiates a color change, it sends a message to one of the proxies in the ring with a light color (red in the figure). The proxy creates a color token, and

passes it around the ring. The color token changes the color of every proxy it encounters. If the color token meets the state token, its color is darkened, and the coloring continues. The first proxy also darkens its color when it receives the state token. When that proxy receives the color token, it knows that the state token has been lost if and only if the token's color and its own color are equal and light. The final color is sent back to the manager. Note that the proxies never accept a state token with a less recent color, except the proxy that initiated the coloring.

A color change is triggered each time a proxy fails, or when a proxy wants to determine whether it is still reachable from the ring. The proxy is guaranteed to be unreachable if its color has not been changed by the process. The manager keeps track of the proxies that left the ring, and forwards them the new color after the coloring. A proxy that has a different color knows that it is unreachable from the ring.

The invalidation protocol

This protocol, inspired by protocols presented in [Lam79], manages an entity whose state is replicated on its proxies. It implements the annotation `replicated`. It maintains two types of tokens, multiple read tokens and a single write token. Holding a read token allows a proxy to read its local copy of the state of the entity. To perform a write operation, the manager asks proxies to release their read token, and *invalidate* their copy of the state. Once all read tokens have been collected, the write token is used to update the state, then read tokens are redistributed to proxies with the new state. The proxies delay read operations until they receive a read token.

In its first formulation, the manager was giving the write token to the proxy that requested the write operation [Kli05]. The new state was then sent to the manager, which redistributed it with read tokens. However, that protocol was sensitive to proxy failures. The author has chosen to simplify its failure modes by performing all write operations on the protocol manager. This makes the protocol insensitive to proxy failures. It also improves performances, since the manager no longer has to send the write token to a proxy.

8.2.2 Protocols for immutables

Those protocols should only offer a *read* operation. The stationary protocol is valid for immutable entities, as well as three others, namely the immediate protocol, and the eager and lazy replication protocols. Their implementation in the DSS was done by Per Sahlin [Sah04], then slightly extended by the author to handle partial failures.

The immediate protocol is not really a protocol, since a full representation of the entity is serialized when a reference is passed between sites. The eager and lazy replication protocols are pretty simple: each proxy can ask its manager a copy of the entity's state. Once the state is installed, all read operations are

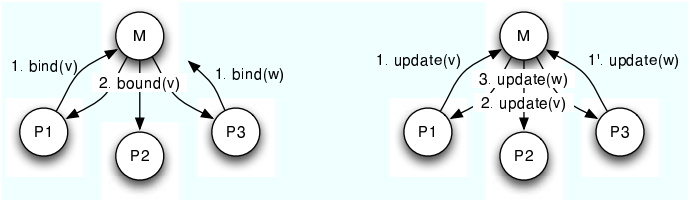


Figure 8.7: Transient protocol: bind and update operations

performed locally. In the eager protocol, a proxy requests the state right after its creation. In the lazy protocol, the proxy delays the state request until the first read operation. Both the eager and lazy protocols guarantee a unique copy of the entity's state. Indeed, if a reference to the entity is sent to a site, that site will identify the reference to the entity's proxy. If the proxy had already requested the entity state, it will not request it again.

8.2.3 Protocols for transients

This protocol implements single assignment variables, with two operations: *bind* and *update*. The protocol has first been published in [HVB+99]. The DSS implements this protocol extended with an incremental update operation. Upon creation, the transient entity is unbound. Its transient state may be updated as many times as desired, until the entity is bound. The binding is unique and final; all subsequent updates and bindings will fail. By-need synchronization in Mozart/DSS is implemented as an update (see Section 8.3.4).

Bind. In order to guarantee the unicity of the binding, the protocol manager plays the role of an arbiter of binding attempts. The protocol is depicted in Figure 8.7: proxies send binding requests to their manager; the latter accepts the first request, and forwards the binding to all proxies. All subsequent binding and update requests are ignored. When proxies receive the binding, they install the final state in their entity, and check their former binding attempts to decide whether they have succeeded.

As the manager broadcasts the binding to all its proxies, they must register to their manager, unless the coordination network provides a way to reach them all. The registration is done at proxy creation, when it is deserialized, via an explicit registration message sent to the manager. If the entity is bound at the message reception, the manager replies with the entity's binding. Note that the registration can be optimized if the entity reference was sent by the home proxy: the manager may automatically register the destination site. This *autoregistration* mechanism saves a registration message, and can improve the throughput of the protocol in case of stream communication.

Update. State updates are handled in a similar way, but they do not put an end to the entity. All updates are serialized by the manager, which forwards them to all known proxies, so that the updates are applied in the same order on all proxies (see the right drawing in Figure 8.7). When a proxy registers, the manager may send back an update that *summarizes* all former updates. This guarantees that the proxy does not miss past updates, provided that the entity's state can reflect all past updates. This summary update is a contribution of the author. Its absence creates a race condition between proxy registration and updates.

The transient remote protocol. This variant of the transient protocol, chosen by the annotation `reply`, delegates the arbiter role to a proxy. That proxy can directly bind the entity, and forward the binding to the manager, which broadcasts it to the other known proxies. The manager forwards all the other updates and binding requests to that proxy, which serializes them. The protocol is optimal if there is only one remote proxy, and that proxy binds the entity. Indeed, the proxy is autoregistered because the entity reference must have been sent from the manager's site, and the only message actually sent is the forwarded binding.

The manager is responsible for choosing the arbiter proxy. The simplest rule is to choose the first proxy registered outside the manager's site. If that proxy is deleted, it sends a deregistration message to the manager, which reassigns the arbiter role to its home proxy.

8.2.4 Handling failures

All entities supported by the DSS may fail. Failures have two origins: the environment and the programmer, and both kinds must be detected and reflected to the user. To implement that, each coordination proxy maintains a failure state for its entity. Each time that state changes, the proxy notifies its corresponding mediator in the Glue layer. The state may be changed by the proxy itself, or its protocol proxy.

Failures due to the environment are detected via DSites. We take for granted that DSites have their own failure detection mechanism, which reflects their corresponding site's fault state. Consider a DSite representing site b on a site a. Once the DSite changes its fault state, this change is notified to all coordination and protocol components on site a. Every component checks whether it affects its entity. If so, it changes its failure state accordingly, and notifies the mediator of its entity. How an entity is affected depends on the entity's coordination architecture and protocol. For instance, the mobile state protocols will probe the proxies that may hold the entity's state, in case one of those proxies is reported as failed.

Failure are not only reported locally, but also to a global scale. A generic protocol supports this global reporting. Once an entity is diagnosed as permanently failed by a protocol manager, the latter broadcasts a message `PERMFAIL`

to all its known proxies. Those proxies will then update their failure state, and report the change to their own mediator. Note that this generic protocol is sometimes adapted to avoid inconsistencies. For instance, the transient protocol never makes an entity permanently failed after its binding.

Kill. In order to let a program make an entity fail, all protocols support an operation called *kill*. The same generic protocol is used to implement that operation. To perform a kill, a protocol proxy simply sends a message `PERMFAIL` to its manager, which propagates the failure globally as we explained in the former paragraph.

8.2.5 Distributed garbage collection

The garbage collector provided by the DSS maintains a status for each coordination proxy in the system. The proxies of a given entity have different status, depending on the references and the entity's protocol. A given proxy can be in one out of four states:

- `PRIMARY`: the entity is kept alive by remote references. The virtual machine must keep the entity, because other sites depend on it. This status means that the coordinator of the entity is on the current site.
- `WEAK`: the entity is kept alive because for protocol needs. This typically happens when the current site is the only one to hold the entity's state. That status is generally not definitive: the DSS can be instructed to move away from that status.
- `LOCALIZE`: no remote reference keeps this entity alive. This usually means that all the proxies of the entity are gone except this one. The entity can be localized or deleted, depending on whether it is kept alive locally.
- `NONE`: the liveness of the entity entirely depends on local information. If the entity is alive, the proxy should be kept. Otherwise, both can be removed.

Note that proxies are considered alive by the DSS until they are deleted explicitly by the upper layer. In fact all the components at the interface of the DSS library must be deleted explicitly. The DSS uses them as roots for its own internal garbage collector.

Section 8.3.6 on page 123 explains how these states are used by the local garbage collectors. The distributed garbage collector itself is implemented by the *reference* components shown in Figure 8.2 on page 109. Several protocols are available, and they can be combined together. The most important protocols are a weighted reference counting algorithm, and a time lease algorithm. More details on the algorithms are given in Erik Klintskog's works [Kli05, KNBH01].

8.3 The language interface

The Glue layer implements the language interface to the distribution library. As the distribution of Oz comes from sharing entities, the most important component of this layer is the *mediator*, that interfaces an entity to its proxy, and vice-versa. Every distributable entity has a mediator, which contains the entity's annotations, its fault state, and its memory status. For the sake of performance, the mediator of a local entity is created lazily, once the entity is annotated, broken, or serialized for a remote site.

An entity is distributed if and only if it has a proxy in the DSS layer, otherwise it is purely local to its site. In general, distributed entities have a pointer to their mediator, while local entities have an indirect access to their mediator via a table. The mediator itself has pointers to both its entity and proxy, and the latter has a pointer to the mediator. This provides both efficient access for distributed entities, and small overhead for purely local entities.

8.3.1 Distributed operations in general

Performing a language operation on a distributed entity involves several components in the three layers of Mozart/DSS. Those components are depicted in Figure 8.8 on the next page, with the horizontal dashed lines separating the implementation layers. Each one has a specific role during the distributed execution of the operation. Note that entity failures are ignored here; they are explained in the next section.

A language operation on a distributed entity is always delegated to the Glue layer, which accesses the entity's mediator, then the corresponding DSS proxy, and invokes the latter with a generic operation. The proxy forwards the call to its protocol proxy, which implements the protocol chosen for the entity. The protocol proxy prescribes to either perform the operation locally, as if the entity was purely local, or suspend and resume it later. The decision entirely depends on the protocol.

Let us illustrate the decision taken in the case of the protocol `migratory` (mobile state). If the entity's state is on the site that attempts the operation, it will be performed locally. This is fine, since the state is stored in the virtual machine's representation of the entity. If the entity's state is not present, the operation is suspended, and the distributed protocol is used to complete the operation. Once performed, it is resumed, together with the thread that attempted it.

Suspension and resumption. If the operation has to be suspended, the Glue creates an object that represents the suspended operation. That object must be able to resume the operation whenever the protocol says so. Resumption may be done in two ways: either

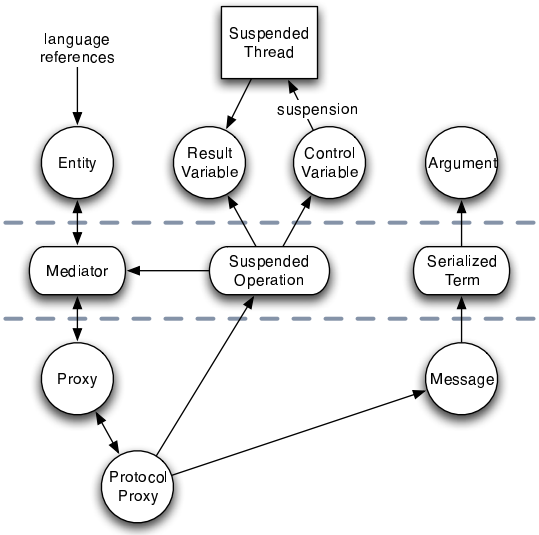


Figure 8.8: The components involved into the distributed execution of a language operation

- the protocol installs a copy of the entity's state in the local entity, and the operation is performed locally, or

- the operation has been performed remotely, and the operation resumes by delivering the results.

The suspended operation typically suspends the thread that attempted the operation on a control variable. Once the operation completes, the thread is woken up by binding the variable. The control variable also permits to raise an exception in the thread, or resume the operation with another statement. The suspended operation can also deliver an output from a remote execution through a result variable.

When an operation is performed on a remote site, the entity's protocol proxy on that site invokes the corresponding mediator in order to perform a virtual machine operation.

Passing values. Protocol messages may include Oz values. Those values can be the input or output of an operation that is performed remotely, or a value that represents the entity's state. They are encapsulated in a Glue component that takes care of their (de)serialization. The DSS library defines an interface for a suspendable marshaler. It means that values are generally not serialized as a whole, but the serialization is done until a buffer is almost full. This technique generally results in a smaller memory footprint.

8.3.2 Distributed immutables

The role of an immutable entity's proxy is to either provide a copy of the entity's contents (protocols `immediate`, `eager`, and `lazy`), or to provide remote access to the entity (protocol `stationary`). We call the first kind of entities *copiable*, because their distribution protocol consists in copying their contents between sites.

Copiable entities do not always require to query the Glue layer to perform an operation. Indeed, once the contents of the entity is available on a given site, all operations on that site will be performed locally. The overhead of distribution can be reduced to nothing if the virtual machine performs this optimization.

Another aspect of copiable entities is that they can survive the execution of a program: they can be stored in a file, and reused later, possibly by another program. This can be an issue if the entity's identity is provided by its coordination network, because the latter is no longer functional once the program stops. For this reason, we provided the entity with a global identifier that does not refer to a live DSS component. As a consequence, it is possible to remove the entity's proxy once it has been copied. This can be done for instance by the garbage collector.

8.3.3 Remote invocations and thread migration

Stationary objects and procedures are never copied between sites, only their reference is transmitted between sites. The operation they have in common is the call (also called invocation in the case of objects). For this operation, an object can be seen as a special case of a procedure. Therefore the discussion will only mention procedures, and calls to stationary procedures.

Assume that a thread on a site a attempts to execute the call statement `{`$p\ x_1 \ldots x_n$`}`, where p is a stationary procedure on a site b. If the protocol proxy of p has to perform the call remotely, it asks the Glue layer to transmit arguments for the remote operation, i.e., in our case x_1, ..., x_n. The Glue layer on site b then simply calls p again with the given arguments on a new thread, which will execute p because it is now on its site. As the procedure call may exit with an exception, site b creates a variable z, and pushes the following statement on the thread:

```
z = try {p x1 ... xn} unit
    catch E then {FailedValue E} end
```

The variable z is returned to site a, which replaces the original call to p by `{Wait` z`}`. This automatically synchronizes the thread, and transmits an exception if needed. This is illustrated in Figure 8.9 on the facing page, for a procedure `P` with two arguments.

This simple solution works in most cases, but it has a subtle issue: the calling thread, and the one that actually executes the procedure have different

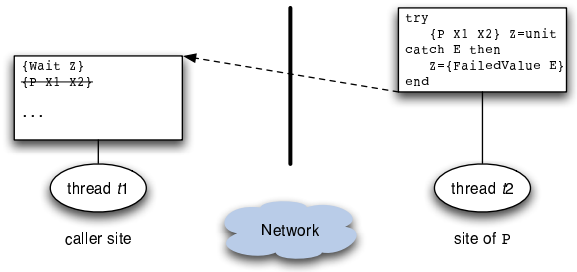


Figure 8.9: A remote procedure call

thread identifiers. This is a problem if these threads lock critical sections with *reentrant locks*. Those locks allow a thread to enter many times in a critical section, and use the thread identifier to distinguish between threads. For the remote procedure call to be *really* transparent, both threads should have the same identifier, just like if the thread has *migrated* between sites. Otherwise, a deadlock may occur. In the example of Figure 8.9, we should have $t1 = t2$.

To realize this, the identifier t of the caller thread is sent together with the procedure's arguments. On site b, the procedure is executed on a thread with identifier t. If no such thread exists on b, one is created. If such a thread exists, then by design it must be suspended on another remote procedure call; the topmost statement must be a call to `Wait` as above. Pushing a new statement on that thread is safe, because after that statement is reduced, the thread will re-suspend thanks to the `Wait` statement.

8.3.4 Unification and by-need synchronization

While the unification operation belongs to the virtual machine, its implementation deserves a special attention. The reason is that a single unification may involve several concurrent distributed variable bindings. Performing those distributed bindings sequentially may have a significant impact on the operation's performance. This impact may be reduced to a minimum if those bindings are truly performed in parallel. The author has modified the existing implementation of the unification in Mozart, which was interrupted by every distributed binding it had to perform.

The unification of two terms basically traverses two directed graphs, and binds the encountered variable nodes in order to make both graphs equivalent. If a variable is local to a site, its binding is done immediately. But a distributed variable may require to invoke its protocol. In that case, the binding is suspended until the protocol terminates the operation, and returns its result. Suspending the whole unification at that point is correct but inefficient, since all involved distributed bindings will be performed in sequence.

To make a better implementation of the unification's semantics, the original

algorithm is modified as follows. When a variable binding suspends, it is put aside in a "suspended set". Note that bindings of read-only variables are also put in that set. The bindings in the suspended set are considered valid until the algorithm terminates. If the algorithm terminates without failing, and the suspended set is nonempty, say $\{x_1{=}v_1, \ldots, x_n{=}v_n\}$ with $n > 0$, the unification resumes as unifying two tuples with the remaining bindings, i.e.,

$$x_1\#\ldots\#x_n = v_1\#\ldots\#v_n.$$

Moreover, the current thread is suspended on the variables x_1, ..., x_n. The thread will be woken up as soon as one of those variables is bound (possibly by a distributed binding), and the unification will make progress. If all the variables x_i are distributed, then all the bindings $x_i{=}v_i$ will proceed concurrently.

Unifying distributed variables. The unification of two distributed variables requires a tiebreaker to decide which variable is bound to the other. This is necessary for avoiding "binding cycles" in the system. Indeed, if two sites attempt to perform $x{=}y$ and decide differently, x may be bound to y and vice-versa. This is problematic, since both transients are bound, and can therefore no longer be bound to anything else: the variables are unbound forever. The tiebreaker is borrowed from the DSS, which provides an arbitrary total order between all its distributed entities. The order is guaranteed to be the same on all sites.

By-need synchronization. As we have seen in Chapter 7, the semantics of by-need evaluation simply requires to propagate the need of a variable on all sites. The operation *update* provided by the transient protocols perfectly fulfills the job. Once a variable is made needed, an update operation is performed, which will make all representatives of that variable needed. Note that making a variable needed never blocks, since that update can be considered asynchronous.

8.3.5 Fault stream and annotations

The mediator of a language entity manages most aspects of to the distribution of that entity. Some of those aspects, like the entity's fault stream, are visible as language entities. Others, like the entity's annotations, are stored directly in the mediator. Figure 8.10 on the facing page shows the extra language entities used by an entity's mediator.

Fault stream and blocking threads. First, the mediator keeps track of the entity's fault state. It is updated whenever a new fault state is reported by the entity's proxy, or enforced by the user (`localFail`). The mediator manages the entity's fault stream by keeping a reference to its tail, a read-only variable. The stream is extended each time the fault state changes. The mediator also

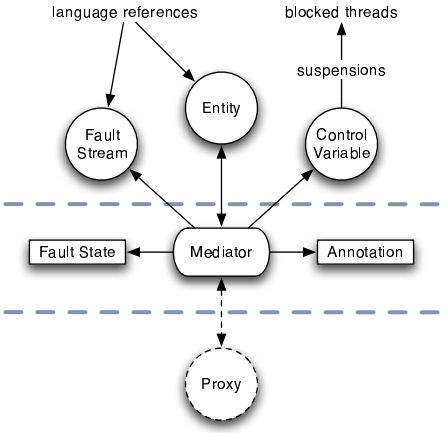


Figure 8.10: The entities managed by an entity's mediator

manages a control variable. If a thread attempts an operation on the entity while its fault state is not `ok`, the thread suspends on that variable. Whenever the fault state becomes `ok`, that control variable is bound to **`unit`**; this automatically wakes up all blocked threads, which will retry their operation.

Note that the memory footprint of the entity pretty small in practice. Both the fault stream and the control variable are created lazily, once the program needs them. Another optimization comes from the fact that the control variable is only effective with the fault state `tempFail`. Indeed, a thread blocking because of the fault states `localFail` and `permFail` will never resume. Therefore the control variable does not need to be kept alive by the mediator in that case, since it will never be bound.

Annotations and proxy. The entity's annotations are also stored in the mediator. They are used to create a DSS proxy for the entity. The creation and removal of the proxy are both managed by the mediator. The proxy creation (called entity *globalization*) is triggered when the entity is serialized, while its removal (called entity *localization*) is prescribed by the garbage collector when the entity is no longer referenced outside its original site.

8.3.6 Garbage collection

The memory management of Mozart/DSS involves both the virtual machine's garbage collector and the DSS's distributed garbage collector. We will call them the local and DSS garbage collectors, respectively. The latter has already been described in Section 8.2.5 on page 117. This section focuses on the cooperation between the implementation layers.

The basic principle is that a live entity keeps its mediator alive, and a live

mediator keeps all its entities (the entity, its fault stream and control variable) alive. We combine that principle with the information coming from both the virtual machine and the DSS. When the Glue layer decides to remove or localize a distributed entity, it deletes the entity's proxy. The following paragraph explains how the decision is taken.

Putting it all together. The cooperation between the garbage collectors is quite generic. First, both the virtual machine and the DSS should provide correct information about what must be kept in memory. Second, some decisions, like the correct handling of the `WEAK` state above, depend on whether an entity is kept alive by local computations only. Because some of those entities have to be kept anyway, the process requires two passes of the local garbage collector. The main steps of the garbage collection process are the following.

1. Distributed entities in state `PRIMARY` are taken as roots for the local garbage collector.
2. The local garbage collector is run, which recursively marks entities from the roots. We can now determine which entities are marked by local computations.
3. The distributed entities in state `WEAK` are checked. For each such entity, if it has not been marked yet, mark it and instruct its proxy to move away from that state.
4. Local garbage collection is performed again. Now all entities that must be kept in memory are marked.
5. The distributed entities in state `NONE` are kept only if they are marked locally; otherwise they are deleted together with their mediator and proxy. The distributed entities in state `LOCALIZE` are localized (their proxy is removed) if they are marked locally; otherwise they are deleted together with their mediator and proxy.
6. The DSS performs its own internal garbage collection. This has no effect at all on the virtual machine's memory heap.

These steps give the broad idea for collecting the entities that must be kept in memory. However some important details are missing, in particular at step 1. The rest of the section identifies the missing roots for the local garbage collection, with a detailed explanation for each. We analyze (in order) the cases of distributed variables, fault streams, threads blocked on a failure, and the components involved in a distributed operation.

Distributed variables. The process described above does the right thing for most distributed entities. However, let us analyze what happens to threads that suspend on distributed variables. Recall that when a variable is alive, its

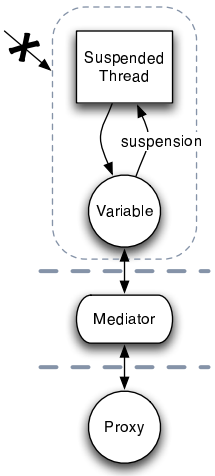


Figure 8.11: A distributed variable with suspensions only

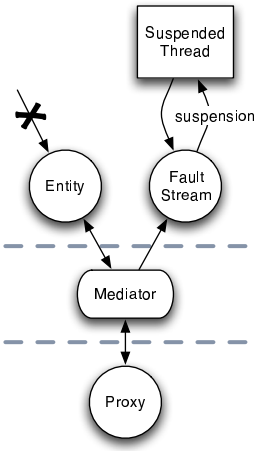


Figure 8.12: A distributed entity with monitoring threads only

suspensions are kept alive. If no live entity refers to any of them, the variable and its suspensions are considered dead. The following situation, depicted in Figure 8.11, might be problematic: a thread suspends on a distributed variable, on a site that does not hold the variable's coordinator. If nothing else keeps that variable alive, it might be considered dead, together with its associated suspended threads. Should the Glue layer keep this variable alive?

Our answer is: yes, distributed variables with local suspensions should be considered as roots for the local garbage collector. The reason is pretty simple. Suspending on a distributed variable is a common idiom, where one site waits for the result of a computation performed on another site. The programmer rarely considers the blocked thread as possibly dead. On the contrary: that thread is often used to keep the continuation of a local computation alive. Silently removing the variable and its suspensions would be an error.

Note, however, that the garbage collector is unable to find out whether there exists a thread in the system that can bind the variable. If no thread binds the variable, all suspensions are dead. The suspensions can also be considered dead if the variable is permanently failed, locally (`localFail`) or globally (`permFail`). If the program is able to detect a dead variable, it can help the local garbage collectors by making the variable fail. We extend the step 1 above with:

1.1. Distributed variables with local suspensions are taken as roots for the local garbage collector, unless they are permanently failed.

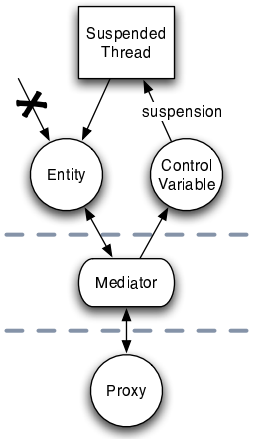


Figure 8.13: A failed entity with blocked threads

Fault streams. While an entity is alive, its fault stream is alive. This is a consequence of the basic rule we mentioned above: the entity keeps its mediator alive, which itself keeps the entity's fault stream alive. Now consider the situation depicted in Figure 8.12 on the previous page. The site has no reference to the entity, but it still monitors it, with a thread suspended on the entity's fault stream.

The garbage collection process must implement the policy given in Section 4.4 on page 54. First, the entity may be removed from memory, since it is not referred to. Second, if the entity is removed, its fault stream must be closed with its tail bound to `nil`. Binding the stream tail should wake up the monitoring threads suspended on it. In order to be effective, it requires both the stream tail and its suspensions to be alive. Therefore, the fault stream of an entity must be kept alive until it is closed. The implementation is very simple, we add the step:

1.2. The fault streams of entities are taken as roots for the local garbage collector.

Blocked threads. We are now interested in the threads that block on a failed entity. Technically those threads are suspended on the control variable used by the entity's mediator to resume from temporary failures. The situation is illustrated in Figure 8.13. Let us recall the principle stated in Section 4.4 on page 54. If the failure is temporary, those threads must be kept alive, otherwise they would not resume. If the failure is permanent, they are not kept alive by the entity itself. We can easily ensure the liveness of those threads by adding the following step:

1.3. The control variables used by the mediators of temporary failed entities

are taken as roots for the local garbage collector.

Distributed operations. Those operations require to handle two extra data structures: components representing suspended operations and terms being serialized (see Figure 8.8 on page 119). Both are freed explicitly by the DSS; meanwhile, they must maintain their associated virtual machine entities alive. In other words, both suspended operations and terms being serialized, can be considered as roots for the local garbage collector. So we add the step:

1.4. Entities referred to by suspended operations and serialized terms are taken as roots for the local garbage collector.

9 Evaluation

This chapter gives an evaluation of our work. Both the language definition and its implementation are evaluated.

9.1 Ease of programming

In Chapter 5 we have shown a few abstractions built with our distribution model. From these examples, we can already say that both the customization of a distributed application and their handling of partial failure are not difficult. The examples show that nontrivial abstractions are coded relatively easily, and without too much code.

9.2 Performance

In this section we compare the performances of Mozart/DSS and Mozart. We will see that their performances are similar, with Mozart/DSS being a bit slower than Mozart. But the new protocols made available by Mozart/DSS permit to take better advantage of the distribution of Oz than Mozart. These performance comparisons complete the experiments done by Erik Klintskog on an early version of Mozart/DSS in [KBBH03]. These experiments showed that the Distribution Subsystem (DSS) incurred around 12% overhead on the total time for a client to perform a given number of requests to a server, when compared to Mozart. The same experiment also showed that Mozart/DSS was about twice as slow as an equivalent C++ program, optimized for the experiment, and using raw sockets for communication.

Issues. The execution of the performance tests was considerably delayed by bugs we discovered in the implementation. Not all those bugs could be fixed.

```
proc {Server N Len}
   S ServerPort={NewPort S}
in
   {Offer ServerPort ...}   % make ServerPort available
   for I in 1..N   get(X) in S do
      L={List.number I+1 I+Len 1}   % list of Len integers
   in
      X=L
   end
end

proc {Client N}
   ServerPort={Take ...}   % connect to the server's port
in
   for I in 1..N do X in
      {Send ServerPort get(X)}
      {Wait X}
   end
end
```

Snippet 9.1: Mozart/DSS vs. Mozart: server and client

Some of them were found in code that we did not write ourselves, notably in the DSS, which made the debugging task quite difficult. So the experiments we show here are the ones that could be run without triggering the remaining bugs.

9.2.1 Mozart/DSS vs. Mozart

This section compares the performance of the distribution layers in both platforms Mozart and Mozart/DSS. We consider a simple client-server program, and compute the CPU time spent in the distribution layer. Note that we do *not* measure network delays in this case. The client sends `N=100000` requests of the form `get(X)` to the server, and the server replies by binding `X` to a list of `Len` elements. The number `N` of requests has been chosen large enough in order to trigger the garbage collector, so that all the components of the distribution layer are involved in the test. We made experiments for `Len=10` and `Len=1000` to compare between small and big messages.

Snippet 9.1 above gives the code for the server and client. The server offers a port for the client to communicate. The procedure `Offer` creates a ticket, and makes it available (in a file, or on a web site). The procedure `Take` retrieves the ticket, and connects to it. The client sends `N` requests, and waits for the reply, so that requests are sequential. For the sake of simplicity, we made the server wait for `N` requests, then exit.

		Mozart	Mozart/DSS	
`Len=10`	centralized	0.260	0.292	
	distributed	25.500	29.304	
	difference	25.240	29.012	(+16%)
`Len=1000`	centralized	6.354	6.378	
	distributed	53.080	80.910	
	difference	46.726	74.532	(+60%)

Table 9.1: Comparison of total CPU times between platforms (network delays not included in measurements)

Results. Table 9.1 gives the results of our experiments. The numbers are the total CPU times spent by client and server in different situations. Network delays are not taken into account. Times are measures in seconds, and averaged over 5 executions. The programs were run on a computer with an Intel Core Duo 2 GHz processor. The centralized case corresponds to the client and server being run on a single site. In the distributed case, they are created on distinct sites, and their CPU time are added. The difference between the two reflects the global overhead in time of the distribution for that program.

The obvious observation is that Mozart/DSS is slower than Mozart, but the ratio between the two is reasonable. The slower performance of Mozart/DSS can be partly explained by its higher degree of flexibility, which requires a few more indirections when performing an entity operation. Another observation is that the size of shared data also has an impact on the relative performance of both platforms: the marshaling process is a bit more involved in Mozart/DSS, because the existing Mozart marshaler is wrapped to comply to the generic API defined by the DSS.

9.2.2 Comparing protocols

We show how we can dramatically change the network behavior of a simple program by changing how a distributed entity is annotated. The program used in the experiment is a simplistic chat application: peers join a group, and broadcast messages within that group. Each peer provides a port on which other peers send messages. The group itself is handled by a small server, which provides a cell with the list of ports. Message broadcast is done by sending the message to all the ports in the list. A peer joins the group by connecting to the server, and adding its port to the list in the cell.

Note that the distributed cell is not fault-tolerant. The purpose of the program is only to show performance variations within a program, just by changing distribution parameters. In this case, we will compare two protocols for distributing the cell.

Snippet 9.2 on page 133 gives the code for the group server and the group peers. The server is instantiated with a particular protocol for the cell. A peer

protocol	*total time*
migratory	63.906
replicated	11.775
no distribution (all peers on one site)	10.754

Table 9.2: Comparison of total time to complete (network delays included)

is created with a nickname. It first subscribes to the group. It then waits for one message to arrive on its port, then sends 1000 messages, with a pause of 10 milliseconds between two messages, unsubscribes and exits. Waiting for a message permits to synchronize peers, so they all start sending messages at the same time. Typically the last connected peer starts the process by calling `{Broadcast start}`.

The code to pay attention to is the procedure `Broadcast`. Each call to `Broadcast` reads the contents of the cell `Group`, and the procedure is called many times. Therefore the efficiency of the peer is largely influenced by how fast it can read `Group`.

Results. Table 9.2 gives the results of our experiments. The experiment consists in n peers broadcasting messages, with one of them also playing the role of the group server. The value $n = 3$ was sufficient to emphasize differences between protocols. The last connected peer broadcasts a dummy message to synchronize all peers. We measured the total time for one of the peers to run completely, i.e., the elapsed time between its startup and termination. In this case, we *do* measure network delays.

The experiment was run on Monday 5th November 2007, between 16:00 and 16:30. The times are measured in seconds and averaged over 5 runs. The peers were located on three machines: `calc6.info.ucl.ac.be` (everlab cluster at UCL), `planet8.cs.huji.ac.il` (everlab cluster at the Hebrew University of Jerusalem, Israel), and my own laptop Apple MacBook Pro connected at the UCL network. The server was located together with the peer on the first machine. The times were measured on the peer on the second machine.

As one can see, the migratory protocol, which is the default for distributing cells, is not efficient for that application. On the contrary, the replicated protocol, where each site has a copy of the current state of the cell, is almost as efficient as if all peers were on the same site. The reason is that the cell is read more often than it is updated, and the replicated protocol requires few network communication in that case. The choice of protocol has a noticeable impact on the performance of that example.

```
proc {Server Protocol}
   Group={NewCell nil}
in
   {Annotate Group Protocol}     % annotate cell
   {Offer Group ...}             % make Group available
end

proc {Peer NickName}
   Group={Take ...}       % connect to the cell Group
   proc {Subscribe P}     % add P to the group
      T in T=Group:=P|T
   end
   proc {Unsubscribe P}   % remove P from the group
      L T in
      L=Group:=T
      T={List.subtract L P}
   end
   proc {Broadcast M}     % send M to all ports in the group
      for P in @Group do
         {Send P M}
      end
   end
   S P={NewPort S}        % this peer's port
in
   thread
      for X in S do {Show X} end     % show all messages
   end

   {Subscribe P}
   {Wait S}                          % wait for start signal
   for I in 1..1000 do
      {Broadcast NickName#I}
      {Delay 10}
   end
   {Unsubscribe P}
end
```

Snippet 9.2: Comparing protocols: server and peer

10 Conclusion

10.1 Achievements

This work extends former studies on network transparency in distributed programming languages, by showing that this approach to distribution is practical regarding both efficiency and failure handling. We have given a few guidelines on how to structure a distributed program, and how to reason about its network behavior. We have extended the language Oz with annotations, that make the programmer able to customize the distribution of a program by choosing between distribution protocols for entities. The resulting program is a valid centralized program, and all distributed executions of the program are valid in a centralized setting.

We have also redesigned failure handling in Oz, made it simpler and more modular. Failure handling is based exclusively on asynchronous failure detection. We have introduced the concept of a fault stream to monitor an entity, and showed that this concept is sufficient to implement complex failure handling algorithms. The design of the fault stream also provides an effective post-mortem finalization mechanism, which was missing in the language. We have introduced new language operations to make entities fail. Those operations give the programmer more control to handle partial failures, for example by propagating the failure to a group of related entities.

On the implementation side, we have completed Erik Klintskog's work by making all protocols of the Distribution Subsystem (DSS) handle partial failure. Finally we have reimplemented the distribution of the platform Mozart on top of the DSS. The new implementation, Mozart/DSS, implements our distribution model for the language. We have been able to test it, and validate its effectiveness. However, the implementation is a prototype, and still suffers from quite a few bugs.

10.2 Future directions

Decentralized applications. This work should be a solid foundation to make applications fully decentralized. Some existing work, using structured overlay networks, have proved to be useful in this domain. Boris Mejías and Donatien Grolaux have already started to reimplement the library P2PS, which implements a structured overlay network in Oz [MCV05]. The new distribution model seems to be promising for that implementation.

A better virtual machine. The implementation of the virtual machine of Mozart is far from being simple. It is written in the language C++, but does not make use of object polymorphism, for instance. It is therefore difficult to maintain and to extend. It lacks modularity.

This lack of modularity is visible in the Glue layer: every language operation must be *explicitly* mapped to a DSS operation. For instance, the Oz cell has two *write* operations: `Exchange` and `Assign`. For each one, we had to provide a mapping to a DSS *write* operation, together with specific callbacks to perform those operations remotely or resume them. A better option would be to define only one *write* operation on cells, and both `Exchange` and `Assign` could be expressed in terms of that operation. All *write* operations in the virtual machine could be given the same distribution support, and polymorphism would allow to use the entity's *write* operation as a callback directly.

The DSS roughly defines five entity operations: *read*, *write*, *send*, *bind*, and *update*. Every language operations in Oz can be expressed as an instance of one of these operations. A mapping of the five generic operations would provide distribution support for all language operations. Of course this support has to be carefully designed, in order to avoid bad performance of elementary language operations, such as the sum operation.

Protocols written in Oz. In the DSS library, all entity protocols are written in C++. The existing protocols have limitations, for instance they have no recovery mechanism in case of failure. An Oz programmer cannot define its own protocol for objects, unless it modifies the underlying library and recompiles the platform. The recommended strategy is to define an Oz abstraction that encapsulates the recovery mechanism, and has its own protocol defined with ports and variables. This reduces the overall usefulness of the underlying library, since few protocols are really crucial.

Together with my colleague Yves Jaradin, we have sketched the design of a virtual machine that can be extended in the language itself. The idea is to give the possibility for an entity to delegate an operation to another entity, for instance a port. A distributed cell could be implemented by coordinating a group of cells on different sites, such that they give the illusion of being a unique cell. For each cell in the group, basic operations are delegated to a local agent, which can send messages (Oz values) to the other sites. The global

identity of the cell can be provided by an Oz name. The virtual machine should support serialization, and the possibility to send values to other sites.

The design has several advantages. First, it provides a framework to experiment with complex entity protocols at the language level. The latter protocol could use a user-defined overlay network to implement the communication between the sites that coordinate for an entity. Second, it gives the possibility to *dynamically upgrade* a protocol. Indeed, the code of the protocol is defined in the language as a value (a procedure, a class, or a functor), which can be send to a network of sites. A new protocol for cells can implemented and deployed without recompiling the virtual machine platform.

Security. Oz is relatively close to the language E, as we can easily encapsulate values and restrict their access via lexical scope. We can define *capabilities* in Oz in a quite reliable way. However, the implementation is more permissive than the language, in particular when distribution is in the game. The author has the impression that, with a reasonable effort, the distribution layer of Mozart/DSS can be made more secure. Some work was already done by Zacharias El-Banna and Erik Klintskog to make the DSS more secure [EKB05].

Besides that, Mozart/DSS also provides some tools at the language level. One can force all mutable entities to be stationary by default, for instance. This prevents a third-party site from screwing up an entity by cheating with its protocol.

A

Summary of the model

This chapter summarizes the distribution model, and all the language extensions introduced in this thesis.

A.1 Program structure

A program is distributed by letting several *sites* share language entities. The latter are stateless or stateful *data*. The basic operations on those entities behave as in the centralized case, modulo some network latency.

The program is deployed over the network by connecting several centralized programs with shared entities. A reference to an entity `X` can be converted from and to an atom `T` with the following functions. The atom is sent between sites by other means (e-mail, web site, etc.)

`{Connection.offer E}` returns an atom `T`.

`{Connection.take T}` returns the entity `E` from which the atom `T` was created with the former function.

Annotations. Entities can be *annotated* with protocol descriptions. The annotation states which kind of protocol should be used to distributed the entity. An entity's annotation cannot be modified.

`{Annotate E A}` annotates entity `E` with value `A`. Raises an exception if the value `A` is not valid for `E`.

Possible values for `A` are:

- access architecture: `access(stationary)`, `access(migratory)`
- entity protocols: `stationary`, `migratory`, `pilgrim`, `replicated`, `variable`, `reply`, `immediate`, `eager`, `lazy`

- garbage collection: `persistent`, `refcount`, `lease`

A.2 Failure handling

Entity failures. An entity fails by crashing: it stops being functional (forever). It can also fail locally on a given site: the entity is crashed on that site, but it can be correct on other sites. Operations on failed entities simply block.

Entity fault states and fault stream. The language provides failure detectors for entities. Those detectors define the following *local fault states* for the entity:

- `ok`: no failure detected
- `tempFail`: entity suspected of having failed, may go back to `ok`. Language operations block on the entity, and resume if the fault state goes back to `ok`.
- `localFail`: entity has failed locally
- `permFail`: entity has failed globally

Those states are notified in the *fault stream* of the entity. This stream reflects the sequence of fault states of the entity. It is accessed with the following function.

`{GetFaultStream E}` returns the fault stream of entity `E`.

The fault streams of two variables are merged when those variables are unified. The tail of the fault stream is bound to `nil` once the entity is no longer in memory. This can be used to program *post-mortem finalization*.

Making entities fail. Two operations are provided:

`{Kill E}` makes `E` fail globally. The operation is asynchronous, and is not guaranteed to succeed.

`{Break E}` makes `E` fail locally. The fault state of `E` becomes `localFail` immediately (unless it was already failed).

Bibliography

[AM03] Mostafa Al-Metwally. *Design and Implementation of a Fault-Tolerant Transactional Object Store*. PhD thesis, Al-Azhar University, Cairo, Egypt, December 2003.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[AWWV96] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[Bra05] Per Brand. *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2005.

[BVCK00] Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. Path redundancy in a mobile-state protocol as a primitive for language-based fault tolerance. Research Report RR2000-01, Université catholique de Louvain, Département INGI, 2000.

[Car95] Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, January 1995.

[Col04] Raphaël Collet. Laziness and declarative concurrency. 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity PostJava'04, 2004.

[CV06] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In C. Dony et al., editor, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 121–140. Springer-Verlag, 2006.

[EKB05] Zacharias El Banna, Erik Klintskog, and Per Brand. Making the distribution subsystem secure. Technical Report T2004:14, Swedish Institute of Computer Science, Kista, Sweden, January 2005.

[GGV04] Donatien Grolaux, Kevin Glynn, and Peter Van Roy. A fault tolerant abstraction for transparent distributed programming. In *Second International Mozart/Oz Conference (MOZ 2004)*, Charleroi, Belgium, October 2004. Springer-Verlag LNCS volume 3389.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at `http://java.sun.com` (June 2007).

[GLT97] H. Guyennet, J.-C. Lapayre, and M. Tréhel. Distributed shared memory layer for cooperative work applications. In *22nd Annual Conference on Computer Networks, LCN'97*, pages 72–78, Minneapolis, USA, November 1997. IEEE Computer Society and TC Computer Communications.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.

[HVB+99] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[HVBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HVS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[ISO98] ISO/IEC. Open distributed processing - reference model: Overview, 1998.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Univ. of Washington, Seattle, Wash., 1988.

[KBBH03] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a middleware library for distribution of language entities. In *Asian Computing Science Conference*, pages 243–259, 2003.

[Kli05] Erik Klintskog. *Generic Distribution Support for Programming Systems*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2005.

[KNBH01] Erik Klintskog, Anna Neiderud, Per Brand, and Seif Haridi. Fractional weighted reference counting. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 486–490. Springer, 2001.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1979.

[MCGV05] Valentin Mesaros, Raphaël Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Research Report RR2005-01, Université catholique de Louvain, Département INGI, 2005.

[MCV05] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for Mozart. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[Moz99] The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org` (June 2007).

[Sah04] Per Sahlin. Efficient distribution of immutable data structures in the distributed subsystem middleware library. Master's thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2004.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[SCV03] Alfred Spiessens, Raphaël Collet, and Peter Van Roy. Declarative laziness in a concurrent constraint language. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03, 2003.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Sun97] Sun Microsystems. *The Remote Method Invocation Specification*, 1997. Available at `http://java.sun.com` (June 2007).

[Van06] Peter Van Roy. Convergence in language design: A case of lightning striking four times in the same place. In *Functional and Logic Programming, 8th International Symposium (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2006.

[VBHC99] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. *Lecture Notes in Computer Science*, 1686:32–46, 1999.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, 2004.

[VHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM TOPLAS*, 19(5):804–851, September 1997.

[VHB99] Peter Van Roy, Seif Haridi, and Per Brand. *Distributed Programming in Mozart – A Tutorial Introduction*, 1999. In Mozart documentation, available at `http://www.mozart-oz.org` (June 2007).

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, CA, November 1994.

A.3 Range queries on structured overlay networks

Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 31 (2008) 280–291
computer communications
www.elsevier.com/locate/comcom

Range queries on structured overlay networks

Thorsten Schütt *, Florian Schintke, Alexander Reinefeld

Zuse Institute Berlin, Takustraße 7, 14195 Berlin-Dahlem, Germany

Available online 31 August 2007

Abstract

The efficient handling of range queries in peer-to-peer systems is still an open issue. Several approaches exist, but their lookup schemes are either too expensive (space-filling curves) or their queries lack expressiveness (topology-driven data distribution).

We present two structured overlay networks that support arbitrary range queries. The first one, named Chord#, has been derived from Chord by substituting Chord's hashing function by a key-order preserving function. It has a logarithmic routing performance and it supports range queries, which is not possible with Chord. Its O(1) pointer update algorithm can be applied to any peer-to-peer routing protocol with exponentially increasing pointers. We present a formal proof of the logarithmic routing performance and show empirical results that demonstrate the superiority of Chord# over Chord in systems with high churn rates.

We then extend our routing scheme to multiple dimensions, resulting in SONAR, a *Structured Overlay Network with Arbitrary Range queries.* SONAR covers multi-dimensional data spaces and, in contrast to other approaches, SONAR's range queries are not restricted to rectangular shapes but may have arbitrary shapes. Empirical results with a data set of two million objects show the logarithmic routing performance in a geospatial domain.

Keywords: Structured overlay networks; Routing in P2P networks; Consistent hashing; Multi-dimensional range queries

1. Introduction

Efficient data lookup is at the heart of peer-to-peer computing: given a key, find the node that stores the associated object. Many structured overlay protocols like CAN [23], Chord [30], Kademlia [20], Pastry [26], P-Grid [1], or Tapestry [31] use *consistent hashing* [16] to store (*key*, *value*)-pairs in a distributed hash table (DHT). The hashing distributes the keys uniformly. A data lookup needs $O(\log N)$ communication hops with high probability in networks of N nodes.

The basis of DHTs [2] is the *key space* which defines the names of items storable in the DHT, e.g. bitstrings of length 160. A partitioning scheme splits the ownership of the key space uniformly among participating nodes, which form a ring structure by maintaining links to neighboring nodes in the *node space*. The ring structure allows to find the owner of any given key.

A typical use of this system for storing and retrieving files might work as follows. Suppose the key space contains keys in the range $[0, 2^{160})$. To store a file with a given filename in the DHT, the SHA1 hash of the filename is computed, producing a 160-bit *key*, and a message `put(key,value)` is sent to an arbitrary participating node, where *value* is the content of the file. The message is forwarded from node to node through the overlay network until it reaches the node responsible for *key* as specified by the key space partitioning, where the pair $(key, value)$ is stored. Any other client can now retrieve the contents of the file by again hashing its filename to produce *key* and asking any node to find the value associated with *key* with a message `get(key)`. The message will again be routed through the overlay to the node responsible for *key*, which will reply with the stored $(key, value)$ pair.

The described hashing uniformly distributes skewed data items over nodes and thereby supports the logarithmic

* Corresponding author. Tel.: +49 3084185361.
E-mail addresses: schuett@zib.de (T. Schütt), schintke@zib.de (F. Schintke), reinefeld@zib.de (A. Reinefeld).

0140-3664/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2007.08.027

routing, it also incurs a major drawback: none of the mentioned P2P protocols is able to handle queries with partial keywords, wildcards, or ranges, because the hashing spreads lexicographically adjacent identifiers over all nodes.

1.1. Schemes for range queries on structured overlays

Two different approaches for range queries on structured overlays have been proposed in the literature: space-filling curves on DHTs and key-order preserving mapping functions for various network topologies.

The approaches based on space-filling curves [5,10,14] incur higher maintenance costs, because they are built on top of a DHT and therefore require one additional mapping step. Even worse, space-filling curves cover the key-space by discrete non-overlapping patches, which makes it difficult to support large multi-dimensional range queries covering several patches in a single lookup. Depending on the patch size, such queries require several independent lookups (Ref. Fig. 4).

The second type of structured overlays map adjacent key ranges to contiguous ranges in the node space. These methods are key-order-preserving and therefore capable of supporting arbitrary range queries. Since the key distribution is not known *a priori*, one approach, Mercury [9], approximates a density function of the keys to ensure logarithmic routing. The associated maintenance and storage overhead is not negligible, despite a mediocre accuracy. MURK [14] goes one step further. It is similar to our approach in that it also splits the data space into hypercuboids that are managed by separate nodes. But in contrast to our approach MURK uses a heuristic based on skip graphs whereas our SONAR builds on an extension of Chord#'s ring topology. More detailed information on related work can be found in Section 4.

1.2. Structured overlays without consistent hashing

In this paper, we argue that it is neither necessary nor beneficial to use DHTs in structured overlays. We first introduce a scheme that matches the key space directly to the node space and is thereby able to support range queries. Thereafter we generalize the scheme to multi-dimensional data spaces.

1.2.1. Chord#

Section 2 presents Chord#. It has been derived from Chord by eliminating the hashing function and introducing an explicit load balancing mechanism. Chord# has the same maintenance cost as Chord, but is superior in a number of aspects: it has a proven upper-bound lookup cost of $\log_b N$ for arbitrary b, it has a constant-time finger update algorithm and it supports range queries. It does so by routing in the node space rather than the key space. We describe the finger placement algorithm (Section 2.1), prove its logarithmic lookup performance (Section 2.1.2 and Appendix A), and demonstrate its improvements over Chord in highly dynamical systems (Section 2.3) with simulations.

1.2.2. SONAR

Section 3 extends the principle idea of Chord# to multiple dimensions. The resulting algorithm, SONAR, performs efficient multi-attribute queries. While other systems [9,14] also support multi-dimensional range queries, to the best of our knowledge there exist no other approach that is equally deterministic in its finger placement. We present details on SONAR's finger placement (Section 3.2), routing strategy (Section 3.4), range query execution (Section 3.5), and demonstrate its practical lookup performance in networks storing two million objects (Section 3.6).

Section 4 discusses related work and Section 5 presents a summary and conclusion.

2. Chord#

Since its introduction in the year 2001, Chord [30] was deployed in many practical P2P systems, making it one of the best-understood overlay protocols. Chord's lookup mechanism is provably robust in the face of node failures and re-joins. It uses consistent hashing to map the keys and node IDs uniformly onto the identifier space. In the following, we introduce our Chord# algorithm by deriving it step by step from Chord as illustrated in the four parts of Fig. 1.

Let us assume a key space of size 2^8 and a node space of size 2^4. Chord organizes the nodes $n_0,\ldots,n_{15}$ in a logical ring, each of them being responsible for a subset of the keys $0,\ldots,2^8-1$. Each node maintains a *finger table* that contains the addresses of the peers halfway, quarter-way, 1/8-way, 1/16-way,..., around the ring. When a node (e.g. n_0) receives a query, it forwards it to the node in its finger table with the highest ID not exceeding `hash(key)`. This halves the distance in each step, resulting in $O(\log N)$ hops in networks with N nodes, because the hashing ensures a uniform distribution of the keys and nodes with a high probability [30] (Fig. 1a).

Because this scheme does not support range queries, we eliminate the hashing of the keys. All keys are now sorted in lexicographical order, but unfortunately the nodes that are responsible for popular keys (e.g. 'E') will become overloaded – both in terms of storage space and query load. Hence, this approach is impractical, even though its routing performance is still logarithmic (Fig. 1b).

When we additionally eliminate the hashing of the *node IDs*, the nodes can be placed at any suitable place in the ring to achieve a better load distribution. We must introduce an explicit load balancing scheme [17] that dynamically removes keys from overloaded nodes. Unfortunately, without adjusting the fingers in the finger table, much more hops are needed to retrieve a given key. The

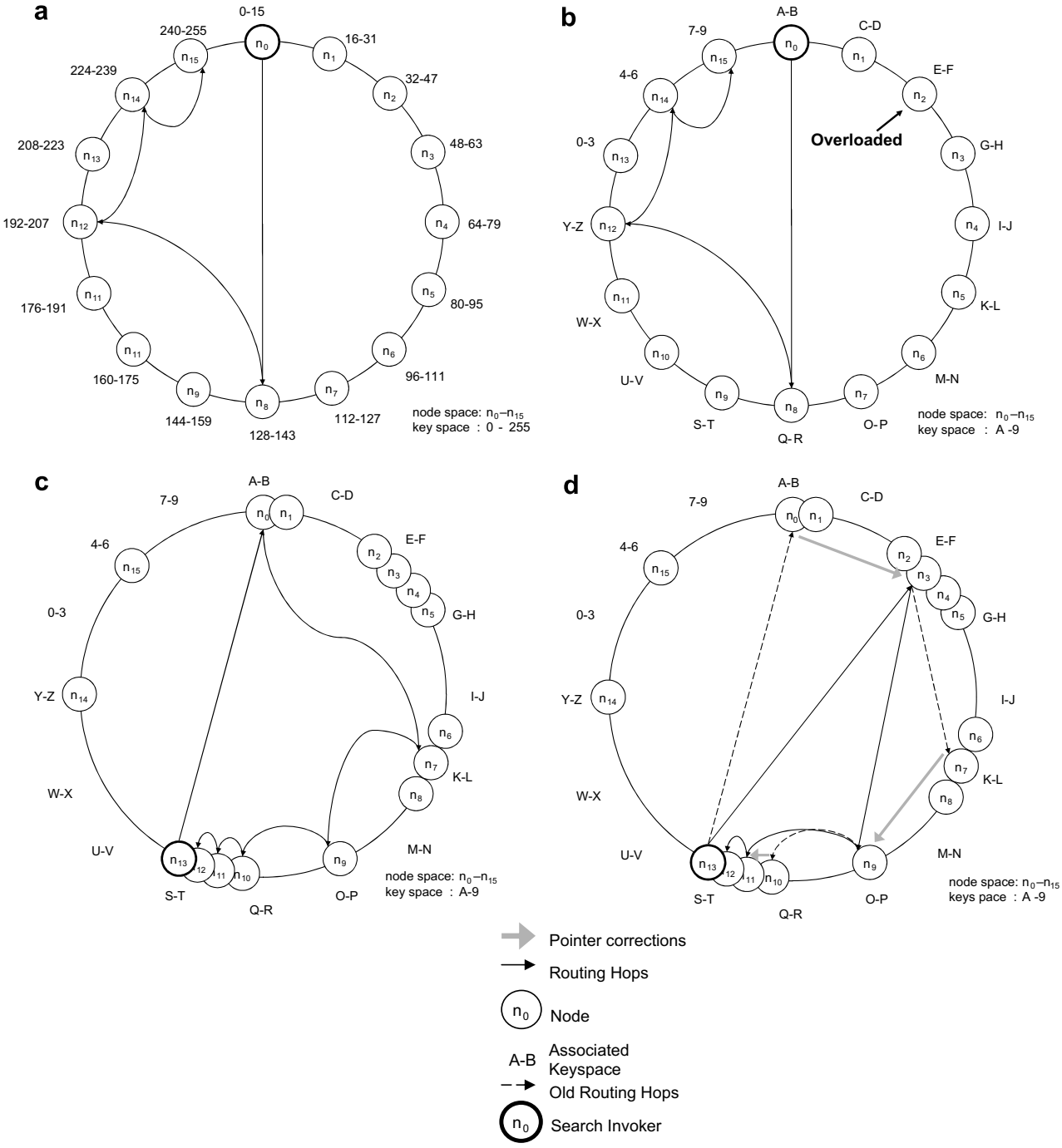


Fig. 1. Transforming Chord into Chord#.

lookup started in node n_{13} for key 'R', for example, needs six instead of four hops. In general, the routing degrades to O(N) (Fig. 1c).

In the final step, we introduce a new finger placement algorithm that dynamically adjusts the fingers in the routing table. The lookup performance is now again O($\log N$) – just as in Chord. But in contrast to Chord this new variant does the routing in the node space rather than the key space, and it supports complex queries – all with logarithmic routing effort (Fig. 1d).

We now present the new finger placement algorithm and discuss its logarithmic performance. A formal proof can be found in Appendix A.

2.1. Finger placement

In the above transformation we substituted Chord's hash function by a key-order preserving function. When doing so, the keys are no longer uniformly distributed over the nodes but they follow some unknown density function.

To still obtain logarithmic routing, we must ensure that the fingers in the routing tables cross an exponential number of *nodes* in the ring. This can be achieved as follows: to calculate the longest finger (i.e., the finger pointing half-way around the ring) we ask the node referred to by our second longest entry (i.e., quarter-way around) for its second longest entry (again quarter-way). For calculating our second longest finger, we follow our third-longest finger, and so on.

In general, to calculate the ith finger in its finger table, a node asks the remote node, to which its $(i-1)$th finger refers to, for its $(i-1)$th finger. The fingers at level i are set to the fingers' pointers in the next lower level $i-1$. At the lowest level, the finger refers to the direct successor.

$$\text{finger}_i = \begin{cases} \text{successor} & : i = 0 \\ \text{finger}_{i-1} \rightarrow \text{getFinger}(i-1) & : i \neq 0 \end{cases}$$

2.1.1. Logarithmic finger table size

The finger table of Chord$^{\#}$ has a maximum of $\lceil \log N \rceil$ entries. This can be seen by observing its construction. Chord$^{\#}$ first enters the shortest finger (direct successor of the node) and recursively doubles the distance in the node space with each further entry. Formally: a node n inserts an additional finger_i into its routing table as long as the following equations holds true:

$$\text{finger}_{i-1} < \text{finger}_i < n$$

This construction process guarantees that – in contrast to Chord – no two entries point to the same node. Since the fingers in the table point to nodes at exponentially increasing distance, it becomes apparent that the routing table has a total of $\lceil \log N \rceil$ entries.

2.1.2. Logarithmic routing performance

Like the original Chord, Chord$^{\#}$ has a routing performance of $O(\log N)$ hops. Unlike Chord, the logarithmic routing performance is not only proven 'with high probability', but it constitutes a guaranteed upper bound. In the following, we outline the basic idea; the formal proof can be found in Appendix A.

Let the key space be $0 \ldots 2^{m-1}$ and let $\oplus$ be an addition modulo 2^m. In the original Chord, the ith finger (finger_i) in the routing table of node n refers to the node responsible for the key f_i with

$$f_i = (n.key \oplus 2^{i-1}) \quad \text{for} \quad 1 \leqslant i \leqslant m$$

The original Chord calculates finger_i by sending a query to the node responsible for f_i. This requires $O(\log N)$ communication hops for each single entry in the routing table, which sums up to $O(\log^2 N)$ hops per routing table.

Chord$^{\#}$'s finger placement algorithm is derived by reformulating the above equation as

$$f_i = (n.key \oplus 2^{i-2}) \oplus 2^{i-2}$$

Having split the right hand side into two terms, the recursive structure becomes apparent and it is clear that the whole calculation can be done in just 1 hop. The first term represents the $(i-1)$th finger and the second term the $(i-1)$th finger on the node pointed to by finger_{i-1}.

Routing in the node space allows us to remove the hashing function and to arrange the keys in lexicographical order among the nodes so that no node is overloaded. This new finger placement has two advantages over Chord's algorithm: first, it works with any type of keys as long as a total order over the keys exists, and second, finger updates are cheaper than in Chord, because they need just one hop instead of a full search. This is because Chord$^{\#}$ uses the better informed remote information for adjusting the fingers in its finger table by recursive finger references.

2.2. Fewer hops with $\log_b$ routing

The routing performance of Chord$^{\#}$ can be further improved at the cost of additional storage space. The idea for this enhancement comes from DKS [3], which inserts extra fingers into the routing table to allow for higher-order search schemes than simple binary search. By this means, the $\log_2 N$ routing performance can be reduced to $\log_b N$ with arbitrary bases b.

In Chord$^{\#}$, the longest finger, finger_{m-1}, with $m = \lceil \log N \rceil$, and the position of the current node n split the ring into two intervals: $[n, \text{finger}_{m-1}]$ and $[\text{finger}_{m-1}, n]$. The intervals have about equal sizes because of the recursive finger update algorithm. The next shorter finger, finger_{m-2}, splits the first half again into two halves $[n, \text{finger}_{m-2}]$ and $[\text{finger}_{m-2}, \text{finger}_{m-1}]$. This splitting is recursively continued until the subsets contain only one key. It becomes obvious that each routing hop cuts the distance to the goal in half, resulting in $O(\log_2 N)$ hops.

By dividing the interval on the ring into b equally sized subsets at each level, each hop reduces the distance to $\frac{1}{b}$th and the overall number of hops per search to $O(\log_b N)$. To implement $\log_b$ routing, we need to extend the routing table by $b-2$ extra columns. The calculation of the fingers is similar to the finger placement algorithm presented in Section 2.1.

Note that with $\log_b$ routing ($b > 2$) there are several alternatives to calculate the long fingers via different intermediate nodes. This allows to eliminate inconsistent fingers in dynamic systems based on local information. Moreover, the update process may be improved by piggybacking information on routing table entries when sending search results. This on-thy-fly correction has only negligible traffic overhead and allows more frequent updates.

2.3. Experimental results

This section presents empirical results of Chord$^{\#}$ in a highly dynamic network. To allow the comparison to other P2P protocols, we followed the experimental set up of Li et al. [19] who simulated a network of 1024 nodes under

heavy churn. The network runs for 6 h and each node fails on average (exponentially distributed) after 1 h with an absent time of 1 h (again exponentially distributed). Hence, only 50% of the nodes are online at any moment. Each alive node issues every 10 minutes a lookup query for a randomly chosen key, where the time intervals are exponentially distributed. Messages have a length of 20 bytes plus 4 bytes for each additional node address contained in the message. The latency between the nodes is given by the King data set [15] which contains real data observed at Internet DNS servers.

Note that the simulation does not account for user data. Only the protocol overhead is measured, and hence the result gives the worst case.

We used the same parameters for testing the algorithm's performance as Li et al. [19], resulting in a total of 480 simulation runs:

(i) *Base* is the branching factor of each finger table entry. Each finger table contains a total of $(\text{base}-1)*\log_{\text{base}}(n)$ fingers. *Values: 2, 8, 16, 32.*
(ii) *Successors* is the number of direct successors stored in each nodes' successor list. *Values: 4, 8, 16, 32.*
(iii) *Successor stabilization interval* denotes the time spent between two updates of the nodes' successor lists. *Values: 30 s, 60 s, 90 s.*
(iv) *Finger update interval* is the time spent between two finger table updates. *Values: 60 s, 300 s, 600 s, 900 s, 1200 s.*
(v) *Latency optimizer* tells whether proximity routing was used for improving the latency. *Values: true, false.*

Fig. 2 shows the average lookup latency versus the maintenance overhead (measured in bytes per node per second). For simplicity, we plotted just the convex hull of the parameter combinations – all inferior data points above the graphs are omitted (more detailed results can be found in [28]). As can be seen, Chord# (dotted graph) strictly outperforms Chord: it has a lower latency with reduced maintenance cost. Much of this favorable performance is attributed to the better finger placement algorithm.

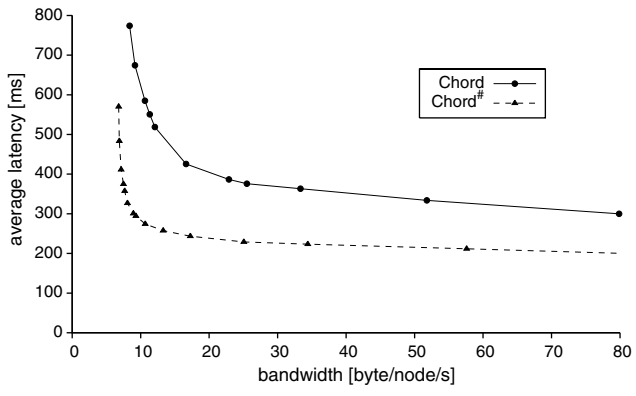


Fig. 2. The convex hull of 480 (resp. 960) experiments with different parameter combinations: Chord# outperforms Chord under churn.

Table 1 lists the number of hops for searching random keys in a network of 1024 nodes. As expected Chord# never needs more than $\log 1024 = 10$ hops while the original Chord requires up to 41 hops. This is because Chord computes the finger placement in the key space, which does not ensure that the distance in the *node space* is halved with each hop.

In summary, the experimental results indicate that Chord# has a lower maintenance overhead (Fig. 2) and requires often fewer lookup hops (Table 1) as compared to Chord. This is true for both, static and dynamic networks. Most of these favorable results are due to Chord#'s recursive finger update algorithm which uses O(1) instead of $O(\log N)$ communication hops to calculate a single finger entry in the routing table. Because of the recursive construction process, one could expect that the longer fingers are more likely to be misplaced – especially in networks with high churn rates. But our experimental results show the contrary: the recursive finger placement is beneficial even under a very high churn.

Table 1
Number of hops to find random keys in 1024 nodes

Hops	Chord	Chord#
0	27	44
1	549	320
2	1924	1433
3	3734	3342
4	4932	4783
5	4377	4930
6	2635	3212
7	1116	1365
8	317	315
9	78	29
10	9	3
12	9	–
13	5	–
14	4	–
15	1	–
16	7	–
17	6	–
18	7	–
19	6	–
20	4	–
21	3	–
22	2	–
23	3	–
24	1	–
25	2	–
26	2	–
27	4	–
28	3	–
29	3	–
30	1	–
31	1	–
33	1	–
37	1	–
38	1	–
41	1	–

Chord# needs a maximum of $\log(1024) = 10$ hops whereas Chord exhibits logarithmic routing performance only with 'high probability', i.e., there are some degenerated cases with considerably more hops.

3. SONAR

In this section, we extend the concept of Chord# to multiple dimensions. The resulting algorithm, named SONAR for Structured Overlay Network with Arbitrary Range Queries, is capable of handling non-rectangular range queries over multiple dimensions with a logarithmic number of routing hops. Non-rectangular range queries are important, for example, for geoinformation systems, where objects are sought that lie within a given distance from a specified position. Other applications include Internet games with thousands or millions of online-players concurrently interacting in a virtual space or grid resource management systems [12,25].

3.1. d-Dimensional data space

SONAR operates on a virtual d-dimensional Cartesian coordinate space with d attribute domains. Keys are represented by attribute vectors of length d. Like in MURK [14,21], the total key space is dynamically partitioned among the nodes such that each node is responsible for approximately the same amount of keys. Explicit load balancing allows to add or remove nodes when the number of objects increases or shrinks or when additional storage space becomes available.

Similarly to CAN [23], each SONAR node participates in d dimensions and has direct neighbors in all directions. The torus is made up of hypercuboids, each of them managed by a single node. Taken together, all hypercuboids cover the complete key space.

During runtime the system balances the key load in such a way that each hypercuboid contains about the same number of keys, and hence each node has to handle a similar amount of data. As a consequence, the hypercuboids have different sizes and a node usually has more than one direct neighbor per direction. Compared to Chord# or CAN, this slightly complicates the routing, because there are usually several options for selecting a 'direct neighbor' – we chose the one that is adjacent to the center of the node (see the small ticks in Fig. 3).

3.2. Building the routing table

Fig. 3 illustrates a routing table in a two-dimensional data space. The keys are specified by attribute vectors (x, y) and the hypercuboids (here: rectangular boxes), which are managed by the nodes, cover the complete key space. Their different area is due to the key distribution: at runtime, the load balancing scheme ensures that each box holds about the same number of keys.

SONAR maintains two data structures, a neighbor list and a routing table. The neighbor list contains links to all neighbors of a node. The node depicted by the grey box in Fig. 3, for example, has ten neighbors.

The routing table comprises d subtables, one for each dimension. Each subtable s with $1 \leqslant s \leqslant d$ contains fingers

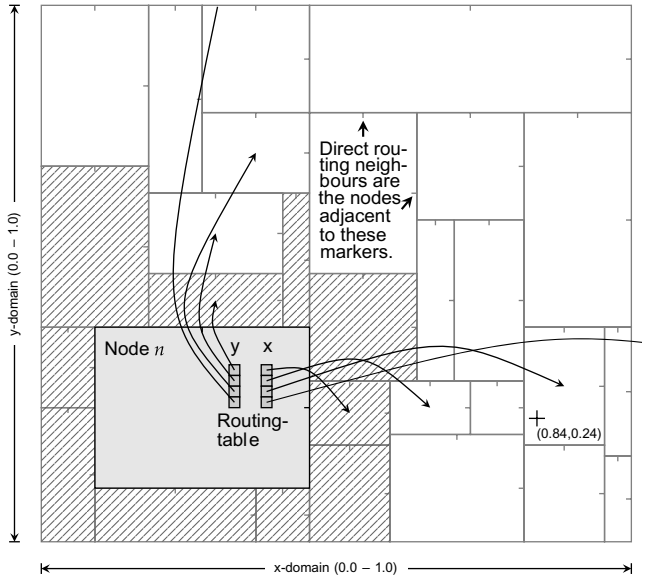


Fig. 3. Routing fingers of a SONAR node in a two-dimensional data space.

that point to remote nodes in exponentially increasing distances. We use the same recursive finger placement algorithm as in Chord# (Section 2.1.2) for inserting fingers into each subtable s of node n: starting with the neighbor that is adjacent to the center of node n in routing direction s, we insert an additional finger_i into the subtable as long as the following equations holds true:

$$\text{finger}_{i-1} < \text{finger}_i < n$$

This construction process ensures that each subtable s contains $\log N_s$ entries, where N_s is the number of nodes in dimension s. Taken together, all subtables s of a node n will have $\log N$ entries – just as in Chord#. Note that this is a self-regulating process [4]: the exact number of entries per dimension depends on the actual data distribution of the application. All that can be said, is that the total number of entries is $\log N$.

3.3. Routing performance

Assuming that the d-dimensional torus is filled uniformly by $N_1 * N_2 * \ldots * N_d = N$ nodes, the routing performance is $\log N_1 + \log N_2 + \cdots + \log N_d = \log N$. So, even though SONAR supports multi-dimensional range queries, it exhibits a similar logarithmic routing performance as Chord#. Our simulations presented in Section 3.6 confirm this observation even for highly skewed, real world data distributions.

3.4. Routing in d dimensions

Chord# uses greedy routing: it always selects the finger that is nearest to the target and forwards the request to this node. In SONAR the situation is more complex, because in

each routing step there are d dimensions to chose from. For regular grids the order of dimensions does not make a difference for the number of routing hops. Here it is best to choose the dimension for the next hop based on network latency, called proximity routing. When the grid is irregular (as in Fig. 3), it is important to select the 'best' dimension in each routing hop – either by distance (greedy routing), by the volume of the managed data space, or simply at random.

3.5. Non-rectangular range queries

The one-dimensional range queries supported by Chord# are defined by a lower and an upper bound: the query returns all keys between those bounds. Extending this approach to d dimensions results in *rectangular* range queries. In contrast to other approaches, SONAR also efficiently supports *non-rectangular* range queries.

Fig. 5 shows a circular range query in two dimensions, defined by a center and a radius. In this example, we assume that a person in the governmental district of Berlin is searching for a hotel. The center of the circle is the location of the person and the radius is chosen to find hotels in 'walking distance'. In a first step, SONAR routes the query to the node responsible for the center of the circle, taking $O(\log N)$ hops. Thereafter, the query is broadcasted to the neighbors which partially cover the circle, which is proportional to the size of the range query (a single hop for each neighbor). The query is performed on the local data and the results are returned to the requesting node.

Other structured overlays, which support multi-dimensional range queries, like [5,14,27,29], use space filling curves to map the multi-dimensional space to one dimension. As shown in Fig. 4 z-curves can be used for this. Different parts of the z-curve are assigned to different nodes. Even for simple range queries several non-contiguous parts of the curve may be responsible for the queried range. For each part a complete lookup with $O(\log N)$ hops is necessary (eight in the given example).

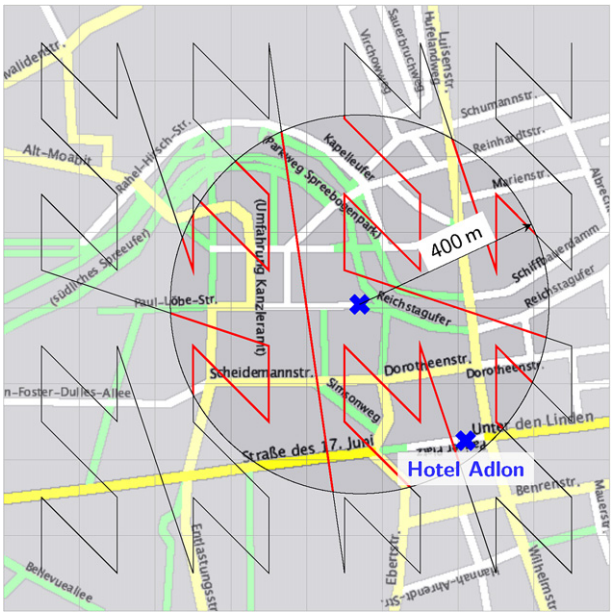


Fig. 4. 2-Dimensional range query using z-curves.

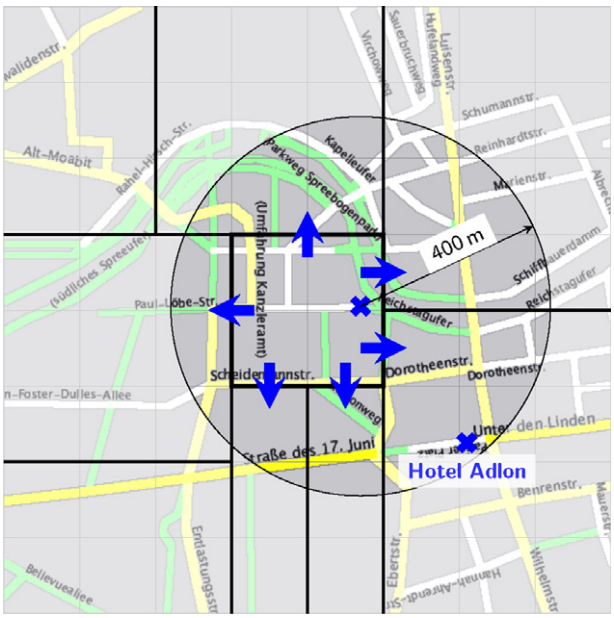


Fig. 5. 2-Dimensional range query by SONAR.

3.6. Performance results

For our experiments we selected a large data set of a traveling salesman problem with the 1,904,711 largest cities worldwide [6]. Their GPS locations follow a Zipf distribution [32], which is a common distribution pattern of many other application domains. In a preprocessing step we partitioned the globe into non-overlapping rectangular patches so that each patch contains about the same amount of cities. We did this by recursively splitting the patches along alternate sides until the number of cities in the area dropped below a given threshold. We mapped the coordinates onto a doughnut-shaped torus (Fig. 6) rather than a sphere, because the poles of a sphere would become a routing bottleneck and the rings for the western, respectively, eastern hemisphere (southwards vs. northwards) would be in opposite directions.

Fig. 7 gives an overview of the results for networks of 128–131,072 nodes. It shows the routing performance (average hop number) and the routing table size in x- and y-direction. As expected, all data points increase logarithmically with the network size.

Most interesting are the results on the routing table sizes: although the nodes autonomously determine the table sizes solely on their local knowledge, the result perfectly matches the theoretical expectation of $\log_2 N$ entries. In the two-dimensional case (depicted here), each routing table contains one subtable in x-direction, which is a bit

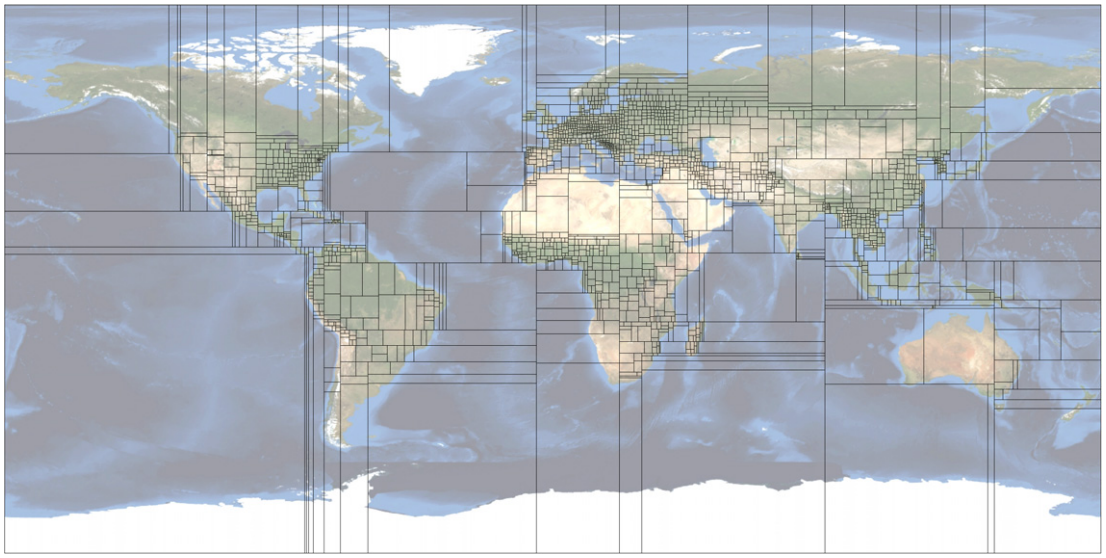


Fig. 6. SONAR overlay network with 1.9 million keys (city coordinates) over 2048 nodes. Each rectangle represents one node.

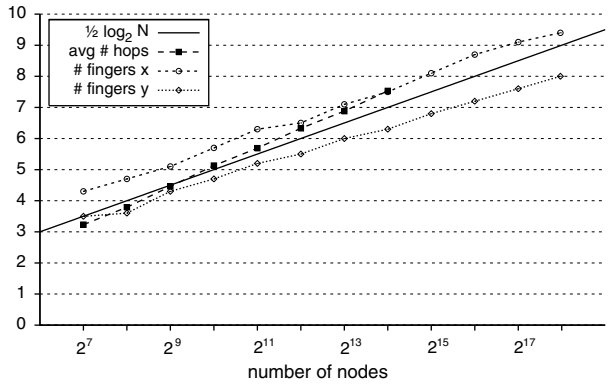


Fig. 7. Lookup performance (avg. #hops) and size of routing tables (#fingers) for various network sizes.

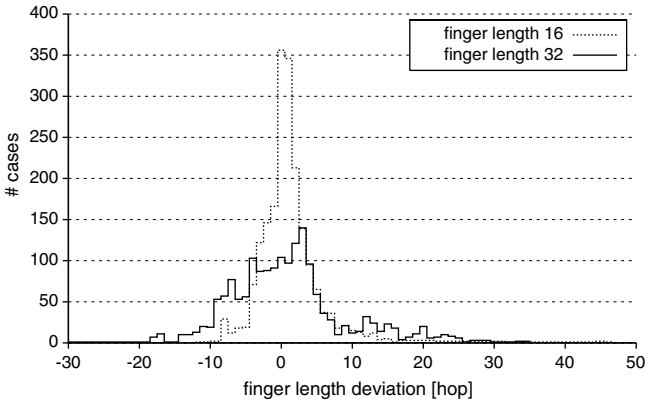


Fig. 8. Deviation of finger lengths due to local knowledge in a torus of 1024 nodes. The measured lengths are centered around their expected length of 16 resp. 32.

larger, and one subtable in y-direction, which is a bit smaller. The deviations are due to the given key distribution of this domain. Together, both subtables contain $\log_2 N$ entries.

Fig. 7 also shows the average number of routing hops. The slope of this graph is slightly above the expected value ($0.5\log_2 N$) because the routing table entries are calculated on local knowledge only, which may result in longer lookup chains. To verify this hypothesis, we checked the accuracy of the finger lengths. Fig. 8 shows the length deviations for fingers of length 16 and 32 in a torus of 2048 nodes. This data was obtained by comparing the entries in the routing table to the actual distance computed with global knowledge (based on a breadth-first search in all directions along the neighbors). The results seem to indicate that the longer fingers of size 32 (straight line) more often overestimate the actual length.

The observed inaccuracy of the finger lengths may also be attributed to the specific data distribution in our experiment [24]. From Fig. 6 it is evident that the rectangles of the whole data space have very different sizes. This becomes more obvious when plotting the number of rectangles versus their size: Fig. 9 shows an almost perfect Zipf distribution, which is common for a large number of applications [11].

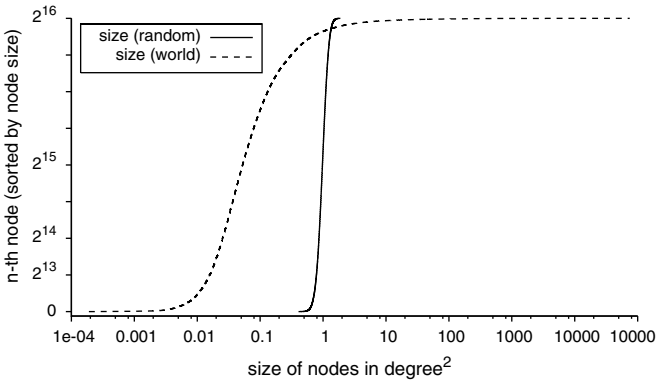


Fig. 9. Distribution of node sizes.

Assuming that large rectangles have more incoming and outgoing fingers than the smaller ones, we plotted the in-degree of all nodes, see Fig. 10. If all keys would be uniformly distributed over the key space, and consequently all nodes would be of the same size, one would expect an in-degree of $\log_2 2048 = 11$ for each node. Fig. 10 indicates that the in-degree is slightly smaller for most nodes, because there are a few (presumably large) nodes with an extremely high in-degree of up to 100.

Due to the different node sizes the rings of the torus are skewed, i.e., they do not end in the beginning node, but may form spirals or may even join in a common node.[1] For the routing process, this does not cause harm, because a data lookup never makes a full round. Only for performance reasons we need to address the problem of uneven in-degrees: some nodes may have to handle more routing requests than others which may affect the routing performance. One solution to this problem would be to include the in-degree into the load metric. The load balancing scheme would then autonomously balance the in-degrees by splitting larger nodes and thereby reducing the in-degree. Alternatively, we could provide more flexibility in the selection of neighbors by allowing also neighbors that are not adjacent to the center of the node for the routing.

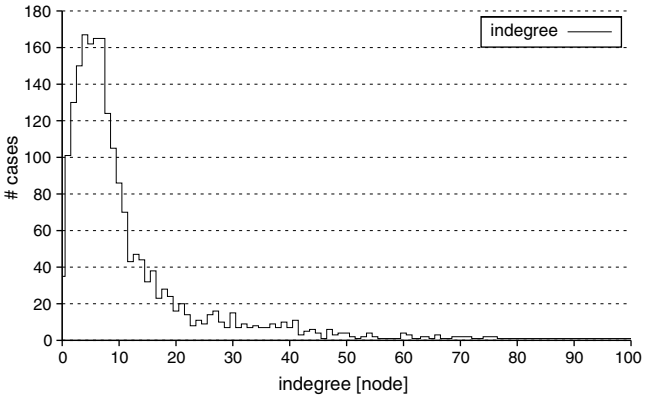


Fig. 10. In-degree of nodes in a torus with 2048 nodes. The expected value is $\log_2 2048 = 11$.

4. Related work

The first structured overlay networks like Chord [30] or CAN [23], published in 2001, allow to organize unreliable peers into a stable, regular structure, where each node is responsible for a part (ring segment in Chord, rectangle in CAN). Due to consistent hashing these system allow to distribute the nodes and keys equally along the system and to route with $O(\log N)$ resp. $O(\sqrt[d]{N})$ performance. Both handle one-dimensional keys but do not support efficient range queries, because adjacent keys are not mapped to adjacent nodes in the overlay.

4.1. One-dimensional range queries

To allow efficient range queries, structured overlays without hashing had to be developed, which put adjacent keys to nodes adjacent in the overlay. So, with one logarithmic lookup for the start of the range, the range query can be performed by that node and the nodes adjacent in the logical structure of the overlay. One major challenge for such systems is the uneven distribution of keys to nodes and the finger maintenance for efficient routing. SkipGraphs [7], a distributed implementation of skip lists [22], for example, use probabilistically balanced trees and thereby allow efficient range queries with $O(\log N)$ performance.

Ganesan et al. [13] further improved SkipGraphs with an emphasis on load-balancing. Their load-balancing scheme maps ordered sets of keys on nodes with formally proven performance bounds, similar to [17]. For routing, they deploy a SkipGraph on top of the nodes, guaranteeing the routing performance of $O(\log N)$ with high probability.

Mercury [9] does not use consistent hashing and therefore has to deal with load imbalance. It determines the density function (see Section A) with random walk sampling, which generates additional traffic for maintaining the finger table.

In contrast to Mercury, Chord# [28] never needs to compute the density function and therefore has significantly less overhead. It is the recursive construction of the routing table in the node space using the successor information, which allows Chord# to handle load-imbalance while preserving an $O(\log N)$ routing performance.

4.2. Multi-dimensional keys and range queries

There exist several systems, that support multi-dimensional keys *and* range queries. They can be split into two groups:

4.2.1. Space filling curves

Several systems [5,14,27] have been proposed that use space-filling curves to map multi-dimensional to one-dimensional keys. Space-filling curves are locality preserving, but they provide less efficient range queries than the space partitioning schemes described below. This is because a single range query may cover several parts which have to be queried separately (Fig. 4).

Chawathe et al. [10] implemented a 2-dimensional range query system on top of OpenDHT using z-curves for linearization. In contrast to many other publications, they report performance results from a real-world application. Due to the layered structure the query latency is only 2–3 s for 24–30 nodes.

4.2.2. Space partitioning schemes

Another approach is the partitioning and direct mapping of the key space to the nodes. SONAR belongs to this

[1] In fact, the overlay built by SONAR is only a torus when the keys are uniformly distributed. In all practical instances, SONAR builds a graph that only slightly resembles a torus.

group of systems. The main differentiating factor between such systems is the indexing and routing structure.

SWAM [8] employs a Voronoi-based space partitioning scheme and uses a small-world graph overlay [18] with routing tables of size O(1). The overlay does not rely on a regular partitioning like a kd-tree, but it must sample the network to place its fingers.

Multi-attribute range queries were also addressed by Mercury [9], but their implementation uses a large number of replicas per item to achieve logarithmic routing performance. Following this scheme, we could employ Chord# for the same purpose, but SONAR supports multi-dimensional range queries with considerably less storage overhead.

Ganesan et al. [14] proposed two systems for multi-dimensional range queries in P2P systems – SCRAP and MURK. SCRAP follows the traditional approach of using space-filling curves to map multi-dimensional data down to one dimension. Each range-query can then be mapped to several range queries on the one dimensional mapping.

MURK is more similar to our approach as it also divides the data space into hypercuboids with each partition assigned to one node. In contrast to SONAR, MURK implements a heuristic approach based on skip graphs.

5. Conclusion

We presented two structured overlay protocols that do not use consistent hashing and are therefore able to support range queries.

Our Chord# provides a richer query expressiveness with the same logarithmic routing complexity as Chord. Its finger update algorithm needs just one communication hop per routing table entry instead of $O(\log N)$ hops as in Chord. As shown in our experimental results (Fig. 2), this greatly reduces the maintenance overhead and it is beneficial in dynamic environments with a high churn rate.

The second part of the paper generalizes the concepts of Chord# to multi-dimensional data. The resulting algorithm, named SONAR for Structured Overlay Network with Arbitrary Range Queries, is capable of handling non-rectangular range queries over multiple dimensions with a logarithmic number of routing hops. Non-rectangular range queries are necessary for geo-information systems, where objects are sought that lie within a given distance from a specified position, or in Internet games with millions of online-players interacting in a virtual game space.

Acknowledgements

We thank the anonymous reviewers and the guest editors for their valuable comments. Our thanks go also to Slaven Rezić for his street map of Berlin (used in Figs. 4 and 5) and to NASA's Earth Observatory for the topographic images from the 'Blue Marble next generation' project in Fig. 6. Part of this research was supported by the EU projects SELFMAN and XtreemOS.

Appendix A. Proof of the logarithmic routing performance of Chord#

Before proving the routing performance of Chord# to be $O(\log_2 N)$, we briefly motivate our line of argumentation. Let the key space be $0 \ldots 2^{m-1}$. In Chord, the ith finger (finger_i) in the finger table of node n refers to the node responsible for the key f_i with[2]

$$f_i = (n.key \oplus 2^{i-1}) \quad \text{for} \quad 1 \leqslant i \leqslant m \tag{A.1}$$

This procedure needs $O(\log N)$ hops for each entry. It can be rewritten as

$$f_i = (n.key \oplus 2^{i-2}) \oplus 2^{i-2}$$

Having split the right hand side into two terms, the recursive structure becomes apparent and it is clear that the whole calculation can be done in just 1 hop. The first term represents the $(i-1)$th finger and the second term the $(i-1)$th finger on the node pointed to by finger $i-1$.

For proving the correctness, we describe the node distribution by the density function $d(x)$. It gives for each point x in the key space the reciprocal of the width of the corresponding interval. For a Chord ring with N nodes and a key space size of $K = 2^m$ the density function can be approximated by $d(x) = \frac{N}{2^m}$ (the reciprocal of $\frac{K}{N}$ and $K = 2^m$) because it is based on consistent hashing.

Theorem 1 (*Consistent hashing* [16]). *For any set of N nodes and K keys, with high probability*:

(i) *Each node is responsible for at most* $(1+\epsilon)\frac{K}{N}$ *keys.*
(ii) *node* $(N+1)$ *joins or leaves the network, responsibility for* $O(\frac{K}{N})$ *keys changes hands* (*and only to or from the joining or leaving node*).

The most interesting property of $d(x)$ is the integral over subsets of the key space:

Lemma 1. *The integral over* $d(x)$ *equals the number of nodes in the corresponding range. Hence, the integral over the whole key space is*:

$$\int_{\text{keyspace}} d(x)\mathrm{d}x = N.$$

Proof. We first investigate the integral of an interval from a_i to a_{i+1}, where a_i and a_{i+1} are the left and the right end of the key range owned by a single node.

$$\int_{a_i}^{a_{i+1}} d(x)\ \mathrm{d}x \stackrel{?}{=} 1.$$

Because a_i and a_{i+1} mark the begin and the end of an interval served by one node, d is constant for the whole range.

[2] We assume calculations to be done in a ring using $(\mathrm{mod}\, 2^m)$.

The width of this interval is $a_{i+1} - a_i$ and therefore according to its definition $d(x) = \frac{1}{a_{i+1}-a_i}$. Because we chose a_i and a_{i+1} to span exactly one interval the result is 1, as expected.

The integral over the whole key space therefore equals the sum of all intervals, which is N:

$$\int_{\text{keyspace}} d(x)\ \mathrm{d}x = \sum_{i=0}^{N-1} \int_{a_i}^{a_{i+1}} d(x)\ \mathrm{d}x = N \qquad \square$$

Note that Lemma 1 could also be used to estimate the amount of nodes $\tilde{N}$ in the system, having an approximation of $d(x)$ called $\tilde{d}(x)$. Each node could compare $\frac{1}{2}\log(\tilde{N})$ to the observed average routing performance in order to estimate and improve its local approximation $\tilde{d}(x)$.

A.1. Finger placement in chord

Both, Chord and Chord$^{\#}$ use logarithmically placed fingers, so that searching is done in $O(\log N)$. Chord, in contrast to our scheme, computes the placement of its fingers in the key space. This ensures that with each hop the distance in the key space to the searched key is halved, but it does not ensure that the distance in the node space is also halved. So, a search may need more than $O(\log N)$ network hops. According to Theorem 1, the search in the node space still takes $O(\log N)$ steps *with high probability*. In regions with less than average sized intervals ($d(x) \gg \frac{N}{K}$) the routing performance degrades.

Chord places the fingers finger_i in a node n with the following scheme:

$$f_i = (n.key \oplus 2^{i-1}), \quad 1 \leqslant i \leqslant m \tag{A.2}$$

Using our integral approach from Lemma 1 and the density function $d(x)$, we develop an equivalent finger placement algorithm as follows. First, we take a look at the longest finger finger_{m-1}. It points to the node responsible for $n + 2^{m-1}$ when the key space has a size of 2^m. This corresponds to the opposite side of n in the Chord ring. With a total of N nodes this finger links to the $\frac{N}{2}$th node to the right with *high probability* due to the consistent hashing theorem.

With Lemma 1 key f_{m-1}, which is stored on the $\frac{N}{2}$th node to the right, can be predicted.

$$\int_n^{f_{m-1}} d(x)\mathrm{d}x = \frac{N}{2}$$

Other fingers to the $\frac{N}{4}$th, …, $\frac{N}{2^i}$th node are calculated accordingly.

As a result we can now formulate the following more flexible finger placement algorithm:

Theorem 2 (Chord finger placement)**.** *For Chord, the following two finger placement algorithms are equivalent*:

(i) $f_i = (n.key \oplus 2^{i-1}), \quad 1 \leqslant i \leqslant m$
(ii) $\int_n^{f_i} d(x)\,dx = \frac{2^{i-1}}{2^m}N, \quad 1 \leqslant i \leqslant m$

Proof. To prove the equivalence, we set $d(x) = \frac{N}{2^m}$ according to Theorem 1.

$$\int_n^{f_i} d(x)\,\mathrm{d}x = \frac{2^{i-1}}{2^m}N$$
$$\int_n^{f_i} \frac{N}{2^m}\,\mathrm{d}x = \frac{2^{i-1}}{2^m}N$$
$$\frac{N}{2^m}(f_i \ominus n) = \frac{2^{i-1}}{2^m}N$$
$$f_i = n.key \oplus 2^{i-1} \qquad \square$$

The equivalence of Chord's two finger placement algorithms will be used in the following section to prove the correctness of Chord$^{\#}$'s algorithm.

A.2. Finger placement in Chord$^{\#}$

Theorem 3 (*Chord$^{\#}$ finger placement*)**.**

$$\text{finger}_i = \begin{cases} \text{successor} & : i = 0 \\ \text{finger}_{i-1} \rightarrow \ \text{getFinger}(i-1) & : i \neq 0 \end{cases}$$

Proof. We first analyze Chord's finger placement (Ref. Theorem 2) in more detail.

$$\int_n^{f_i} d(x)\ \mathrm{d}x = \frac{2^{i-1}}{2^m}N, \quad 1 \leqslant i \leqslant m \tag{A.3}$$

First we split the integral into two equal parts by introducing an arbitrary point X between n (the key of the local node) and f_i (the key of finger_i):

$$\int_n^{X} d(x)\,\mathrm{d}x = \frac{2^{i-2}}{2^m}N \tag{A.4}$$
$$\int_X^{f_i} d(x)\,\mathrm{d}x = \frac{2^{i-2}}{2^m}N \tag{A.5}$$

In Eqs. (A.4) and (A.5), the only unknown is X. Comparing Eq. (A.4) to Theorem 2, we see that X is f_{i-1}.

In summary, to calculate finger_i we go to the node addressed by finger_{i-1} in our finger table (Eq. (A.4)), which crosses half of the nodes to finger_i. From this node the $(i-1)$th entry in the finger table is retrieved, which refers to finger_i according to Eq. (A.5). So, Eq. (A.3) is equivalent to

$$\text{finger}_i = \text{finger}_{i-1} \rightarrow \ \text{getFinger}(i-1)$$

Instead of approximating $d(x)$ for the whole range between n and f_i, we split the integral into two parts and treat them separately. The integral from n to f_{i-1} is equivalent to the calculation of finger_{i-1} and the remaining equation is equivalent to the calculation of the $(i-1)$th finger of the node at finger_{i-1}. We thereby proved the correctness of the pointer placement algorithm in Theorem 3. $\square$

With this new routing algorithm, the cost for updating the complete finger table has been reduced from $O(\log^2 N)$ in Chord to $O(\log N)$ in Chord$^{\#}$.

References

[1] K. Aberer, P-Grid: a self-organizing access structure for P2P information systems, CoopIS (2001).
[2] K. Aberer, L. Onana Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, M. Hauswirth, The essence of P2P: a reference architecture for overlay networks, IEEE P2P (2005).
[3] L. Alima, S. El-Ansary, P. Brand, S. Haridi, DKS(N,k,f): a family of low-communication, scalable and fault-tolerant infrastructures for P2P applications, GP2PC (2003).
[4] A. Andrzejak, A. Reinefeld, F. Schintke, T. Schtt, On adaptability in grid systems, in: V. Getov, D. Laforenza, A. Reinefeld (Eds.), Future Generation Grids, 2006, pp. 29–46.
[5] A. Andrzejak, Z. Xu, Scalable, efficient range queries for Grid information services, IEEE P2P (2002).
[6] D. Applegate, R. Bixby, V. Chvatal, W. Cook, Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems, Mathematical Programming, Series B, 97 <http://www.tsp.gatech.edu/world>, 2003.
[7] J. Aspnes, G. Shah, Skip graphs, SODA (2003).
[8] F. Banaei-Kashani, C. Shahabi, SWAM: a family of access methods for similarity-search in peer-to-peer data networks, CIKM (2004).
[9] A. Bharambe, M. Agrawal, S. Seshan, Mercury: supporting scalable multi-attribute range queries, ACM SIGCOMM (2004).
[10] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, J. Hellerstein, A case study in building layered DHT applications, ACM SIGCOMM (2005).
[11] P.J. Denning, Network laws, CACM 47 (11) (2004).
[12] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998).
[13] P. Ganesan, M. Bawa, H. Garcia-Molina, Online balancing of range-partitioned data with applications to peer-to-peer systems, VLDB (2004).
[14] P. Ganesan, B. Yang, H. Garcia-Molina, One torus to rule them all: multidimensional queries in P2P systems, WebDB (2004).
[15] K.P. Gummadi, S. Saroiu, S.D. Gribble, King: estimating latency between arbitrary internet end hosts, in: Proceedings of the 2nd Usenix/ACM SIGCOMM Internet Measurement Workshop, 2002.
[16] D. Karger, E. Lehman, T. Leighton, R. Panigrah, M. Levine, D. Lewin, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. 29th Annual ACM Symposium on Theory of Computing, 1997.
[17] D. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, IPTPS (2004).
[18] J. Kleinberg, The small-world phenomenon: an algorithmic perspective, in: Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000.
[19] J. Li, J. Stribling, R. Morris, M.F. Kaashoek, T.M. Gil, A performance vs. cost framework for evaluating DHT design tradeoffs under churn, Infocom (2005).
[20] P. Maymounkov, D. Maziéres, Kademlia: a peer-to-peer information system based on the XOR metric, IPTPS (2002).
[21] M. Naor, U. Wieder, Novel architectures for P2P Applications: the continuous-discrete approach, ACM SPAA (2003).
[22] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM (1990).
[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, ACM SIGCOMM (2001).
[24] A. Reinefeld, F. Schintke, T. Schütt, P2P routing of range queries in skewed multidimensional data sets, ZIB-Report 07-23 (2007).
[25] A. Reinefeld, F. Schintke, T. Schütt, Scalable and self-optimizing data grids, in: Yuen Chung Kwong (Ed.), Annual Review of Scalable Computing, vol. 6, 2004, pp. 30–60 (Chapter 2).
[26] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems, Middleware (2001).
[27] C. Schmidt, M. Parashar, Enabling flexible queries with guarantees in P2P systems, IEEE Internet Computing 19–26 (2004).
[28] T. Schütt, F. Schintke, A. Reinefeld, Structured overlay without consistent hashing: empirical results, GP2PC (2006).
[29] Y. Shu, B. Chin Ooi, K. Tan, A. Zhou, Supporting multi-dimensional range queries in peer-to-peer systems, IEEE P2P (2005).
[30] I. Stoica, R. Morris, M.F. Kaashoek, D. Karger, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for Internet application, ACM SIGCOMM (2001).
[31] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, J. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004).
[32] G. Zipf, Relative frequency as a determinant of phonetic change, reprinted from Harvard Studies in Classical Philiology, 1929.

Thorsten Schütt is a Ph.D. candidate with the Zuse Institute Berlin (ZIB). He got his Diploma with distinction in 2002 from the Technical University Berlin. Since then he works as a research staff member in the Computer Science Research Department at ZIB and participates in several EU projects. His research interests include distributed data management, scalable grid systems and P2P computing.

Florian Schintke is a Ph.D. candidate with the Zuse Institute Berlin (ZIB). He graduated in 2000 with distinction from the Technical University, Berlin. Since then, he works as a research staff member in the Computer Science Research Department at ZIB. He participates in several European and German research projects on Grid and Peer-to-Peer computing. He is a member of the German Computer Science Society and the IEEE Technical Committee on Scalable Computing. His research interests are in the areas of distributed data management, scalable systems and autonomic computing.

Alexander Reinefeld heads the computer science Department of the Zuse Institute Berlin and is a professor for parallel and distributed systems at the Humboldt-University Berlin. He received the Diploma degree (M.Sc) in 1982 and the Ph.D. in 1987, both from the University Hamburg. In 1984, he was awarded a Ph.D. scholarship by the DAAD and in 1987 a Sir Izaak Walton Killam Post Doctoral Fellowship of the University of Alberta in Edmonton/Canada. His research interests include grid and peer-to-peer computing, distributed data management, and algorithms for innovative HPC systems such as hardware accelerators. He organized international conferences (CCGrid2002, GGF 2004) and he participates in several editorial and advisory boards. He is a member of ACM, IEEE Computer Society and Gesellschaft fuer Informatik.

A.4 Sloppy Management of Structured P2P Services

Sloppy Management of Structured P2P Services

Paolo Costa[1] Guillaume Pierre[1] Alexander Reinefeld[2]
Thorsten Schütt[2] Maarten van Steen[1]
[1] Vrije Universiteit Amsterdam
[2] Zuse Institute Berlin

1. INTRODUCTION

While most structured peer-to-peer services are conceptually very simple, their implementation is not. Even though similar observations can be made about any type of software, it can be shown that large parts of implementation complexity derives from the way management tasks such as error condition handling are dealt with.

The traditional way to manage distributed systems software is to list all possible error conditions such as churn and partial node or network failures, and come up with repair algorithms that take care of maintaining the desired structure despite adversary conditions. However, implementing repair algorithms is cumbersome, and any error can potentially lead to complex liveness bugs [6], not to speak about unwanted cross-interactions of repair schemes.

We observe that most — if not all — structured peer-to-peer systems deploy strategies to sustain temporary structure inconsistencies that derive from error conditions. In this context, our position is that *explicit repair algorithms can and should be avoided in the implementation of structured peer-to-peer services. Instead, we should use continuous background probabilistic algorithms to handle non-functional management tasks such as routing table maintenance, while relying on the original structured algorithms for the functional tasks such as routing messages through a DHT.* We call this form of probabilistic overlay maintenance "sloppy management".

We already demonstrated in two simple situations that such a dual approach based on a structured deterministic functional plane and an unstructured probabilistic non-functional can actually work [2, 8]. As an example we briefly discuss in Section 2 how gossip-based protocols can be used to maintain the routing tables of a Chord DHT. However, if we are to take the sloppy management approach seriously, many other non-trivial management tasks should be handled by the probabilistic plane. We discuss them in Section 3, and finally conclude in Section 4.

2. EXAMPLE

Teaching the algorithms from Chord to graduate students can be done in no more than three slides, if one is willing to ignore all management issues. The first slide introduces the circular ID space and the way nodes and data items are hashed into IDs; the second slide presents routing tables and shows which fingers should be stored where; the last slide shows how messages can be routed to a key in $\log N$ hops. As every peer-to-peer expert knows, the real difficulty is to build and maintain the routing tables over time in a decentralized fashion.

In [8] we showed how very simple gossip-based probabilistic algorithms can be used to efficiently maintain Chord routing tables, without using any explicit repair algorithm. The probabilistic routing table management algorithm is decomposed in two separate gossip-based overlays. At the bottom level, nodes periodically gossip with each other to create an ever-changing random network within themselves. It is important to note that gossipping takes place at a fixed periodicity, even if no structure inconsistency appears in the Chord routing tables.

The second management overlay uses similar periodic gossipping, but this time it does not aim at building a random network. Instead, it selects links that are good candidates for being used in the Chord finger table. Each node periodically exchanges its current list of fingers with one peer from either of the two gossipping overlays. Under steady conditions, routing tables are kept unaltered. However, if failures occur, the system can easily replace failed nodes by acquiring new identities from its neighbors. Note that this does not require any particular action from the periodic gossip: all error conditions such as churned nodes and partial network failures are simply ignored.

One can show that this algorithm converges extremely fast and can maintain the Chord structure over time, even under massive churn.

Gossip-based management algorithms have a cost, though: gossipping takes place periodically irrespective of the appearance of structure inconsistencies to repair, which means that some continuous background network traffic is created. This traffic can however be kept low, in the order of less than one message per second and per node on average, which is of the same order (if not lower) to the typical overhead of *alive* messages or periodic stabilization protocols executed by structured systems [1].

3. RESEARCH PLAN

As discussed in the above example, epidemic protocols have proved to efficiently maintain topological properties of several overlay networks [8, 5]. Nevertheless, it is not yet clear to what extent this can be generalized to any possible structure. For instance, while maintaining Chord finger tables is relatively easy, maintaining more complex structures such as [7, 4] may be harder.

The most prominent research challenge, however, is to extend this approach to the management of other non-functional properties, beyond topological constraints. Ideally, one would only need to specify the non-functional invariant that must be maintained (e.g., routing tables should be consistent, system load should be balanced, etc.); the probabilistic layer would autonomously maintain the desired properties, without the need to define explicit algorithms to detect and repair invariant violations.

One interesting issue of large-scale distributed systems that could be addressed by sloppy management is load balancing in the presence of skewed load distributions. Traditional techniques to avoid 'hotspots' resort to periodically run network-wide protocols in order to compute the load of each node, and reorganize the network if some inequality is discovered. This may however incur significant overhead, thus reducing the overall performance of the system. In addition, defining the optimal frequency of this operation is highly critical. A too short period would result in excessive overhead, while a too long period would poorly tolerate unexpected bursts of activity. Similarly, on-demand protocols, running only when one node considers itself overloaded, would save some bandwith but may be inappropriate to ensure timely recovery.

In contrast, sloppy management would continuously work in the background to balance the load. By periodically gossiping with random nodes in the network, a node could detect if other nodes are less overloaded and could properly repartition the load, *before* becoming a hotspot. Furthermore, potential inconsistencies arising in the process (e.g., a node leaves while the takeover is occurring) could be seamlessly solved, relying on the aforementioned fast convergence of gossip-based protocols.

A more difficult challenge is to autonomously deal with other types of system misbehavior due to (partial) failures or system peculiarities. For instance, most peer-to-peer protocols assume, unrealistically, the absence of firewalls and the possibility to establish connections between any pair of nodes. In reality things are more complicated and specific machinery is required to encompass firewalls and network policies. We believe that epidemic protocols have the potential to detect and work around these anomalies in a fully decentralized and autonomous way [3]. For example, in Chord, if a firewall prevents a node to communicate with one of its fingers, the local gossip-based layer will declare that finger dead (from his perspective) and fill in the finger table with another node, thereby maintaining correct routing.

We imagine that epidemic protocols could be used to autonomously work around other types of misbehavior, such as certain types of malicious attacks. For instance, the *Eclipse* attack consists for one or several attackers of attracting most of the overlay links to them before disconnecting, thus creating unrecoverable network partitions because most links are gone. We imagine that, thanks to the very fast convergence properties of epidemic protocols, innocent nodes could autonomously detect and 'repair' the implied skew of in-degree distributions, thereby defeating the attack. Interestingly, in this case, it would not be necessary to identify the origin of the attack nor the vector by which it intruded into the system; simply, good nodes of the overlay would detect that some property is not as it should be, and take autonomous action to bring the system back to a workable state. Thus, for a healthy system it must only be ensured that the majority of the nodes are at a healthy state.

4. CONCLUSION

Most implementation complexity of large-scale distributed overlays is due to algorithms that aim at repairing the overlay in the presence of many adversary events, such as node churn and partial failures. Instead of devising ad-hoc repair algorithms to take care of each possible issue, we propose an approach where **the programmer simply specifies the non-functional invariants to be maintained**, while a simple probabilistic background task is in charge of maintaining the invariant.

We have successfully applied this technique to a few simple examples such as building and maintaining Chord routing tables over time. This encourages us to envisage more ambitious uses of the same technique, for example to balance the load of an overlay under (ever-changing) skewed load distribution, or even to repair the effects of firewalls on the connectivity between nodes.

Pushing the approach to its extreme, we imagine that we could use similar techniques to handle a range of possible attacks. We however remain extremely careful about that last claim: attackers should be expected to deploy the best available *strategy* to defeat the sloppy management system. We therefore consider that any contribution, no matter how modest, of sloppy management towards improving the security of peer-to-peer overlays, would constitute a convincing proof of their power to handle conventional issues such as churn, load balancing, and partial network failures.

5. REFERENCES

[1] Castro, M., Costa, M., and Rowstron, A. Debunking some myths about structured and unstructured overlays. In *Proc. NSDI* (2005).

[2] Costa, P., Pierre, G., and van Steen, M. Autonomous resource selection for large-scale grid systems. Submitted for publication, 2008.

[3] Drost, N., Ogston, E., van Nieuwpoort, R. V., and Bal, H. E. ARRG: Real-world gossiping. In *Proc. HPDC* (July 2007).

[4] Gupta, A., et al. Meghdoot: content-based publish/subscribe over P2P networks. In *Proc. Middleware* (2004).

[5] Jelasity, M., and Babaoglu, O. T-man: Gossip-based overlay topology management. In *Proc. Intl. Workshop on Engineering Self-Organizing Applications* (Hakodate, Japan, may 2006).

[6] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. NSDI* (Apr. 2007), pp. 243–256.

[7] Schutt, T., Schintke, F., and Reinefeld, A. Structured overlay without consistent hashing: Empirical results. In *Proc. CCGrid* (2006).

[8] Voulgaris, S., and van Steen, M. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *Proc. DSOM* (Oct. 2003).

A.5 The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay Networks

Parallel Processing Letters

THE RELAXED-RING: A FAULT-TOLERANT TOPOLOGY FOR STRUCTURED OVERLAY NETWORKS

BORIS MEJÍAS AND PETER VAN ROY
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

ABSTRACT

Fault-tolerance and lookup consistency are considered crucial properties for building applications on top of structured overlay networks. Many of these networks use the ring topology for the organization or their peers. The network must handle multiple joins, leaves and failures of peers while keeping the connection between every pair of successor-predecessor correct. This property makes the maintenance of the ring very costly and temporarily impossible to achieve, requiring periodic stabilization for fixing the ring. We introduce the relaxed-ring topology that does not rely on a perfect successor-predecessor relationship and it does not need a any periodic maintenance. Leaves and failures are considered as the same type of event providing a fault-tolerant and self-organizing maintenance of the ring. Relaxed-ring's limitations with respect to failure handling are formally identified, providing strong guarantees to develop applications on top of the architecture. Besides permanent failures, the paper analyses temporary failures and false suspicions caused by broken links, which are often ignored.

Keywords: peer-to-peer, network topology, relaxed-ring, self-configuration, fault-tolerance

1. Introduction

Building decentralized applications requires several guarantees from the underlaying peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay networks using a Chord-like ring topology [15] are a popular choice when the application needs efficient routing, lookup consistency and accessibility of resources. According to [7], the ring topology is one of the most resilient to failures, and it is competitive with any other structured overlay network with respect to reaching any other node in a small amount of steps.

The ring topology offers many good properties as we just mentioned, but its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents *temporary inconsistencies* with peers massively joining the network, even in fault-free environments, as we will discuss in section 4. A stabilization protocol must be run periodically to fix these inconsistencies increasing the load of the network. One possible solution is presented by DKS [6], offering an

atomic join/leave algorithm based on a locking mechanism. Even when this approach offers strong guarantees, we consider locks extremely restrictive for a dynamic network based on asynchronous communication. Every lookup request involving the critical range of keys must be suspended in presence of a join/leave event in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted the relevant locks. Given that, crashing peers just leave the network without respecting the protocol of the locking mechanism breaking the guarantees of the system. Another critical problem for performance is presented when a peer crashes while some joining or leaving peer is holding its lock.

The problem with the maintenance of the ring is that routing algorithms and correctly assigning responsibilities over keys, rely on the perfect relationship between predecessor and successor. But, every join, leave or failure, brakes temporary this relationship. In order to solve this problem, existing algorithms require the agreement of three nodes to perform a *join* or *leave* operation. Managing three nodes at the same time can create unexpected problems, mainly because transitivity of communication cannot be assumed. If node a can talk to b, and b can talk to c, it does not mean that a can talk to c. This problem is equivalent to false suspicions of failure. Consider that a, b and c are talking to each other. Suddenly, the connection between a and c is broken. Peer a informs b about this failure, but b sees that c is alive, meaning that a falsely suspected of c. This is why algorithm based on the synchronized agreement of three nodes are not fault-tolerant. A recent work [14] conclude that lookup inconsistencies are mainly caused by false suspicions. Churn does not introduce inconsistencies if periodic stabilization is triggered often enough, but this very costly as it will be shown in Section 4.

The contribution of this work is an algorithm that only needs the agreement of two nodes at each stage of the maintenance of the ring. By working with only two nodes in every step we have arrived to the Relaxed-Ring topology, which allows a ring to be partially open. This approach simplifies the joining algorithm dividing it into two steps involving two peers each. Lookup consistency is guaranteed after every step. The algorithm provides a failure recovery mechanism where only two nodes interact in every step. Graceful leaves are consider a special case of failure, and therefore, they are equivalent events. Because of this, there is no need for a graceful leaving protocol. This is useful for end-users, because they can just shut down their application letting the network to handle their departure. Fault-tolerance is achieved at the level of permanent failures, temporary failures and false suspicions, which results from broken links, which are often ignored.

The Relaxed-Ring adds robustness to the network in presence of churn and failures. It simplifies the arrival and departure of peers by dividing these event into smaller and correct steps. Due to the relaxed topology, the routing performance is shortly degraded. On the other hand, there is no need for periodic maintenance of the ring, because the Relaxed-Ring remains correct after every step, reducing the cost of maintenance. Part of this contribution has been published in [11], where the

Relaxed-Ring was presented from the point of view of its design as a self-managing system. This work is focused on the correctness of the algorithm through analytical results, and an empirical validation with simulations.

The next section gives more details of the related work. Section 3 describes the relaxed-ring architecture and its guarantees. We continue with further analysis of the topology. Evaluation of the relaxed-ring is presented in section 4, ending with conclusions and future work.

2. The problem and related work

Chord is the canonical structured overlay network using ring topology. Its algorithms for ring maintenance handling joins and leaves have been already studied [6] showing problems of temporary inconsistent lookups, where more that one node appears to be the responsible for the same key. Peers need to trigger periodic stabilization in order to fix inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronization of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [8, 9] introduce locks in the algorithms in order to provide atomicity of the join and leave operations, removing the need for a periodic stabilization. Unfortunately, locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free environments, which is not realistic. Another problem with these approaches is that they are not starvation-free, and therefore, it is not possible to guarantee liveness. A better solution using locks is provided by Ghodsi [6], using DKS [2] for its results. This approach is better because it gives a simpler design for a locking mechanism and proves that no deadlock occurs. It also guarantees liveness by proving that the algorithm is starvation-free. Unfortunately, the proofs are given in fault-free environments.

The DKS algorithm for ring maintenance goes already in the right direction because it request the locks of only two peers instead of three (as in [8, 9]). It works as follows. Every peer holds a lock that can be exclusively taken by any peer. The lock grants access to update the pointers of the peer. A joining/leaving peer needs to get its own lock and its successor's lock. Let us consider peer q joining in between p and r. Peer q first has to get its own lock and then the lock of r, its successor candidate. This is sufficient for q to update its predecessor and successor, to update r's predecessor, and to update p's successor pointer. Note that p cannot change its pointers because that would require getting r's lock, which is already taken by q. The situation of q leaving is analogous, with the difference that p show acquires q's lock in order to perform any action. This mechanism guarantees that if the relevant locks are acquired, the join/leave can be performed *atomically.*

One of the problem with the algorithm is that even when it only requires the lock of two peers, it still requires the *atomic* update of the pointers of *three* peers. While this three changes are made, no lookup involving peers p, q or r is allowed. A

more complex problem is that the algorithm relies on peers gracefully leaving the ring, which is neither efficient nor fault-tolerant. The algorithm becomes very slow if a peer holding a relevant lock crashes. How can the other peers continue? The same problem occurs if a locked peer stop responding. Another problems is that a joining peer q that acquires its own lock and r's lock, is not guaranteed to establish communication with p in order to change its successor pointer.

We are not aware of other approaches solving the problem of atomic join/leave with failure recovery, or other approaches targeting the elimination of periodic stabilization.

3. The Relaxed-Ring

The relaxed-ring topology has evolved from the Peer-to-Peer System (P2PS) [5], and it is implemented using the Mozart-Oz programming system [13]. As any overlay network built using ring topology, in our system every peer has a successor, predecessor, and fingers to jump to other parts of the ring in order to provide efficient routing. Ring's key-distribution is formed by integers from 0 to $N-1$ growing clockwise. For the description of the algorithms we will use event-driven notation. When a peer receives a message, the message is triggered as an event in the ring maintenance tier.

Range between keys, such as $(p, q]$ follows the key distribution clockwise, so it is possible that $p > q$, and then the range goes from p to q passing through 0. Parentheses '(' and ')' excludes a key from the range and, '[' and ']' includes it.

As we previously mentioned, one of the problem we have observed in existing ring maintenance algorithms is the need for an agreement between three peers to perform a join/leave action. We provide an algorithm where every step only needs the agreement of two peers, which is guaranteed with a point-to-point communication. In the specific case of a join, instead of having one step involving 3 peers, we have two steps involving 2 peers. Lookup consistency is guaranteed after every step, therefore, the network can still answer lookup requests while simultaneous nodes are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers. We treat leaves and failures as the same event. This is because failure handling already includes graceful leaves as a particular case.

Normally the overlay is a ring with predecessor and successor knowing each other. If a new node joins in between these two peers, it introduces two changes. The first one is to contact the successor. This step already allows the new peer to be part of the network through its successor. The second step, contacting the predecessor, will close the ring again. Following this reasoning, our first invariant is that *every peer is in the same ring as its successor*. Therefore, it is enough for a peer to have connection with its successor to be considered inside the network. Secondly, the responsibility of a peer starts with the key of its predecessor, excluding predecessor's key, and it finishes with its own key. Therefore, a peer does not need to have connection with its predecessor, but it must know its key. These are two

crucial properties that allow us to introduce the relaxation of the ring. When a peer cannot connect to its predecessor, it forms a branch from the *"perfect ring"*. Figure 1 shows a fraction of a relaxed ring where peer t is the root of a branch, and where the connection between peers q and p is broken.

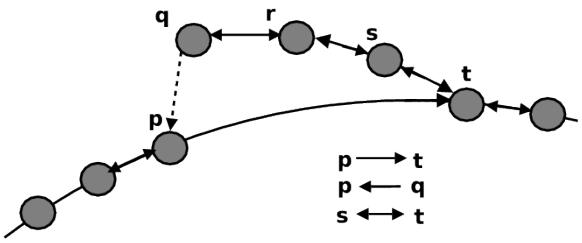


Fig. 1. A branch on the relaxed-ring created because peer q cannot establish communication with p. Peers p and s consider t as successor, but t only considers s as predecessor.

Having the relaxed-ring architecture, we introduce a new principle that modifies the routing mechanism of Chord. The principle is that *a peer p always forwards the lookup request to the possible responsible, even if p is the predecessor of such responsible.* Considering the example in figure 1, p may think that t is the responsible for keys in the interval $(p, t]$, but in fact, there are three other nodes involved in this range. In Chord, p would just reply t as the result of a lookup for key q. In the Relaxed-Ring, the p forwards the message to t. When the message arrives to node t, it is sent backwards to the branch, until it reaches the real responsible. Forwarding the request to the responsible is a conclusion we have already presented in [11], and it has been recently confirmed by Shafaat [14].

Introducing branches into the lookup mechanism modifies the guarantees about proximity offered by Chord. Reaching the root of a branch takes $O(log_k(n))$ hops as in Chord, because the root of the branch belongs to the *core-ring*. Then, the lookup will be delegated a maximum of b hops, where b corresponds to the size of the branch. Then, lookup on the relaxed-ring topology corresponds to $log_k(n) + b$. We will see in section 4 that the average value b is smaller than 1 for large networks.

Before continuing with the description of the algorithms that maintain the relaxed-ring topology, let us define what do we mean by lookup consistency.

Def. *Lookup consistency implies that at any time there is only one responsible for a particular key k, or the responsible is temporary not available.*

Algorithm 1 describes the initial procedure of a node that wants to join the ring. First, it gets its own identifier from a random key-generator. In the implementation, identifiers also represent network references. For simplicity of the description of the algorithms, we will just use the key as identifier and as connection reference. Initially, the node does not have a successor (*succ*), so it does not belong to any ring, and it does not know its predecessor (*pred*), so obviously, it does not have responsibilities.

For resilient purposes, the node uses two sets: a successor list (*succlist*) and an old-predecessor sets (*predlist*). Having an access point, that can be any peer of the ring, the new peer triggers a lookup request for its own key in order to find its best successor candidate. This is quite usual procedure for many Chord-alike systems. When the responsible of the key contacts the new peer, the event *reply_lookup* is triggered in the new peer. This event will generate a joining message that will be discussed in section 3.1.

Algorithm 1 Starting a peer and the lookup algorithm

```
1: procedure init(accesspoint) is
2:     self := getRandomKey()
3:     succ := nil
4:     pred := nil
5:     predlist := ∅
6:     succlist := ∅
7:     send ⟨ lookup | self, self ⟩ to accesspoint
8: end procedure

9: upon event ⟨ lookup | src, key ⟩ do
10:    if (key ∈(pred, self]) then
11:        send ⟨ reply_lookup | self ⟩ to src
12:    else
13:        p := getBetterResponsible(key)
14:        send ⟨ lookup | src, key ⟩ to p
15:    end if
16: end event

17: upon event ⟨ reply_lookup | i ⟩ do
18:    send ⟨ join | self ⟩ to i
19: end event
```

The *lookup* event verifies if the current node is responsible for *key*. If it is not, it picks the best responsible for the key from its routing table, and forwards the request, passing the key and the original source *src*. Choosing the best responsible of a key follows the same mechanism as Chord, with the extra consideration of rooting to the branch when needed, as explained above. One way to decide that a lookup must jump into the branch is by adding a flag to the message called *last*. In the case of Figure 1, when p forwards the messages to t, it sets the flag to *true*. Then, the function *getBetterResponsible* will decide to forward to the predecessor, jumping in to the branch.

3.1. *The join algorithm*

As we have previously mentioned, the relaxed-ring join algorithm is divided in two steps involving two peers each, instead of one step involving three peers as in existing solutions. The whole process is depicted in figure 2, where node q joins in between peers p and r. Following algorithm 1, r replies the lookup to q, and q send the *join* message to r triggering the joining process.

The first step is described in algorithm 2, and following the example, it involves peer q and r. This step consists of two events, *join* and *join_ok*. Since this event may happen simultaneously with other joins or failures, r must verify that it has a successor, respecting the invariant that every peer is in the same ring as its successor. If it is not the case, q will be requested to retry later.

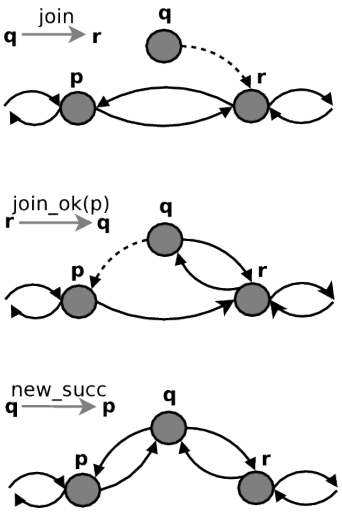


Fig. 2. The join algorithm.

If it is possible to perform the join, peer r verifies that peer q is better predecessor than p. Function *betterPredecessor* checks if the key of the joining peer is in the range of responsibility of the successor candidate. In the example, r verifies that $q \in (p, r]$. If that is the case, p becomes the old predecessor and is added to the *predlist* for resilient purposes. The *pred* pointer is set to the joining peer, and the message *join_ok* is send to it.

It is possible that the responsibility of r has change between the events *reply_lookup* and *join*. In that case, q will be redirected to the corresponding peer with the *goto* message, eventually converging to the responsible of its key.

When the event *join_ok* is triggered in the joining peer q, the *succ* pointer is set to r and *succlist* is initialized. Then, q must set its *pred* pointer to p acquiring its range of responsibility. At this point the joining peer has a valid successor and a range of responsibility, and then, it is considered to be part of the ring, even if p is not yet notified about the existence of q. This is different than all other ring networks we have studied.

Note that before updating the predecessor pointer, peer q must verify that its predecessor pointer is *nil*, or that it belongs to its range of responsibility. This second condition is only used in case of failure recovery and it will be described in section 3.3. In a regular join, *pred* pointer at this stage is always *nil*.

Once q set *pred* to p, it notifies p about its existence with message *new_succ*, triggering the second step of the algorithm.

The second step of the join algorithm basically involves peers p and q, closing the ring as in a regular ring topology. The step is described in algorithm 3. The

Algorithm 2 Join step 1 - adding a new node

```
1: upon event ⟨ join | i ⟩ do
2:     if succ == nil then
3:         send ⟨ try_later | self ⟩ to i
4:     else
5:         if betterPredecessor(i) then
6:             oldp := pred
7:             pred := i
8:             predlist := {oldp} ∪ {predlist}
9:             send ⟨ join_ok | oldp, self, succlist ⟩ to i
10:        else if (i < pred) then
11:            send ⟨ goto | pred ⟩ to i
12:        else
13:            send ⟨ goto | succ ⟩ to i
14:        end if
15:    end if
16: end event

17: upon event ⟨ join_ok | p, s, sl ⟩ do
18:    succ := s
19:    succlist := {s} ∪ sl \ getLast(sl)
20:    if (pred == nil) ∨ (p ∈ (pred, self)) then
21:        pred := p
22:        send ⟨ new_succ | self, succ, succlist ⟩ to pred
23:    end if
24: end event

25: upon event ⟨ goto | j ⟩ do
26:    send ⟨ join | self ⟩ to j
27: end event
```

idea is that when p is notified about the join of q, it updates its successor pointer to q (after verifying that is a correct join), and it updates its successor list with the new information. Functionally, this is enough for closing the ring. An extra event has been added for completeness. Peer p acknowledges its old successor r, about the join of q. When *join_ack* is triggered at peer r, this one can remove p from the resilient *predlist*.

If there is a communication problem between p and q, the event *new_succ* will never be triggered. In that case, the ring ends up having a branch, but it is still able to resolve queries concerning any key in the range $(p, r]$. This is because q has a valid successor and its responsibility is not shared with any other peer. It is important to remark the fact that branches are only introduced in case of communication

Algorithm 3 Join step 2 - Closing the ring

```
1: upon event ⟨ new_succ | s, olds, sl ⟩ do
2:     if (succ == olds) then
3:         oldsucc := succ
4:         succ := s
5:         succlist := {s} ∪ sl \ getLast(sl)
6:         send ⟨ join_ack | self ⟩ to oldsucc
7:         send ⟨ upd_succlist | self, succlist ⟩ to pred
8:     end if
9: end event

10: upon event ⟨ join_ack | op ⟩ do
11:     if (op ∈ predlist) then
12:         predlist := predlist \ {op}
13:     end if
14: end event
```

problems. If q can talk to p and r, the algorithm provides a perfect ring.

No distinction is made concerning the special case of a ring consisting in only one node. In such a case, *succ* and *pred* will point to $self$ and the algorithm works identically. The algorithm works with simultaneous joins, generating temporary or permanent branches, but never introducing inconsistencies. Failures are discussed in section 3.3. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Theorem 1. The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.

Proof.

Let us assume the contrary. There are two peers p and q responsible for key k. If p and q have the same successor is not relevant, because both peers would forward the lookup to the successor, and the successor can resolve the conflict. The problem is when p and q have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$ introducing a inconsistency because of the overlapping or ranges. Let us see now that the algorithm prevents two nodes from having the same predecessor. The join algorithm updates the predecessor pointer upon events *join* and *join_ok*. In the event *join*, the predecessor is set to a new joining peer j. This means that no other peer was having j as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event *join_ok*, the joining peer j initiates its responsibility having a member of the ring as predecessor, say i. The only other peer that had i as predecessor before is the successor of j, say p, which is the peer that triggered the *join_ok* event. This message

is sent only after p has updated its predecessor pointer to j, and thus, modifying its responsibility from $(i, p]$ to $(j, p]$, which does not overlap with j's responsibility $(i, j]$. Therefore, it is impossible that two peers has the same predecessor. □

3.2. *Reducing size of branches*

Let us consider again figure 1. If nodes keeps on joining as predecessors of peer t, the branch will increase its size, even if they could have a good connection with peer p. An improvement on the join algorithm will be that node t sends a *hint* message to node p avoid new joining peer. If p cannot talk to q, it does not mean that it can not talk to r or s. If the p can contact the *hinted* node, it will add it as its successor, making the branch shorter. This hint message will not modify the predecessor pointers of r or s. Peer t uses its *predlist* list for sending hints.

3.3. *Failure Recovery*

In order to provide a robust system that can be used on the Internet, it is unrealistic to assume a fault-free environment or perfect failure detectors, meaning complete and accurate. We assume that every faulty peer will eventually be detected (strongly complete), and that a broken link of communication does not implies that the other peer has crashed (inaccurate). To terminate failure recovery algorithms we assume that eventually any inaccuracy will disappear (eventually strongly accurate). This kind of failure detectors are feasible to implement on the Internet.

When the point-to-point communication layer detects a failure of one of the nodes, the *crash* event is triggered as it is described in algorithm 4. The detected node is removed from the resilient sets *succlist* and *predlist*, and added to a *crashed* set. If the detected peer is the successor, the recovery mechanism is triggered. The *succ* pointer is set to *nil* to avoid other peers joining while recovering from the failure, and the successor candidate is taken from the successors list. The variable *succ_candidate* should be initialized to *nil* in the *init* event of Algorithm 1, but it was not included to avoid confusion at that part of the analysis of the algorithm. The real value is initialized at line 7 of the *crashed* event. The function $getFirst$ returns the peer with the first key found clockwise, and removes it from the set. It returns *nil* if the set is empty. Function *getLast* is analogous. Note that as every crashed peer is immediately removed from the resilient sets, these two functions always return a peer that appears to be alive at this stage. The successor candidate is contacted using the *join* message, triggering the same algorithm as for joining. If the successor candidate also fails, a new candidate will be chosen. This is verified in the if condition.

If a peer p detects that its predecessor *pred* has crashed, it will not trigger the recovery mechanism. It is *pred*'s predecessor who will contact p. In case that no peer contacts p for recovery, p could guess a predecessor candidate from its *predlist*, at the risk of breaking lookup consistency, but closing the ring again. We will not explore this case further in this paper because it does not violate our definition of

consistent lookup. To solve it, it is necessary to set up a time-out to replace the faulty predecessor by the predecessor candidate, but it would always take the risk of a reacting to a false suspicion.

When a link recovers from a temporary failure, the *alive* event is triggered. This can be implemented by using watchers or a fault stream per distributed entity [4]. In this case, it is enough to remove the peer from the *crashed* set. This will terminate any pending recovery algorithm. The faulty peer will trigger by itself the corresponding recovery events with the relevant peers.

Algorithm 4 Failure recovery

```
1: upon event ⟨ crash | p ⟩ do
2:     succlist := succlist \ {p}
3:     predlist := predlist \ {p}
4:     crashed := {p} ∪ crashed
5:     if (p == succ) ∨ (p == succ_candidate) then
6:         succ := nil
7:         succ_candidate := getFirst(succlist)
8:         send ⟨ join | self ⟩ to succ_candidate
9:     end if
10: end event

11: upon event ⟨ alive | p ⟩ do
12:     crashed := crashed \ {p}
13: end event
```

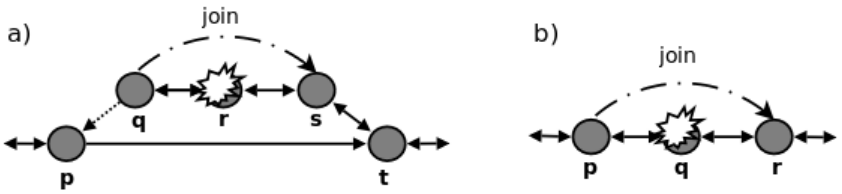


Fig. 3. Failures simple to handle: (a) In a branch, q and s detect that r has crashed. Only q triggers failure recovery. (b) Pers p and r detects q has crashed. Peer p triggers the recovery mechanism.

Figure 3 shows the recovery mechanism triggered by a peer when it detects that its successor has a failure. The figure depicts two equivalent situations. The above one corresponds to a regular crash of a node in a perfect ring. The situation below shows that a crash in a branch is equivalent as long as there is a predecessor that detects the failure.

Having now the knowledge of the *crashed* set, algorithm 5 gives complete definition of the function *betterPredecessor* used in algorithm 2. Since the *join* event is used both for a regular join and for failure recovery, the function will decides if a predecessor candidate is better than the current one if it belongs to its range of responsibility, or if the current *pred* is detected as a faulty peer.

Algorithm 5 Verifying predecessor candidate

```
1: function betterPredecessor(i) is
2:     if (i ∈ (pred, self)) then
3:         return (true)
4:     else
5:         return (pred ∈ crashed)
6:     end if
7: end function
```

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example and check what happens in cases of failures. If q crashes after the event *join*, peer r still has p in its *predlist* for recovery. If q crashes after sending *new_succ* to p, p still has r in its *succlist* for recovery. If p crashes before event *new_succ*, p's predecessor will contact r for recovery, and r will inform this peer about q. If r crashes before *new_succ*, peers p and q will contact simultaneously r's successor for recovery. If q arrives first, everything is in order with respect to the ranges. If p arrives first, there will be two responsible for the ranges $(p, q]$, but one of them, q, is not known by any other peer in the network, and it fact, it does not have a successor, and then, it does not belong to the ring. Then, no inconsistency is introduced in any case of failure. In case of a network partition, these peers will get divided in two or three groups depending on the partition. In such case, they will continue with the recovery algorithm in their own rings. Global consistency is impossible to achieve, but every ring will be consistent in itself.

Since failures are not detected by all peers at the same time, redirection during recovery of failures may end up in a faulty node. The correct version of the *goto* event is described in algorithm 6. If a peer is redirected to a faulty node, it must insist with its successor candidate. Since failure detectors are strongly complete, the algorithm will eventually converge to the correct peer.

Figure 4 shows two simultaneous crashes together with a new peer joining before the peer used for recovery. If the recovery *join* message arrives first, the ring will be fixed before the new peer joins, resulting in a regular join. If the new peer starts the first step of joining before the recovery, it will introduce a temporary branch because of its impossibility of contacting the faulty predecessor. When the recovery *join* message arrive, the recovering peer will contact the new joining peer, fixing the ring and removing the branch.

There are failures more difficult to handle than the ones we have already anal-

Algorithm 6 Modified goto

```
1: upon event ⟨ goto | p ⟩ do
2:     if (p ∉ crashed) then
3:         send ⟨ join | self ⟩ to p
4:     else
5:         send ⟨ join | self ⟩ to succ_candidate
6:     end if
7: end event
```

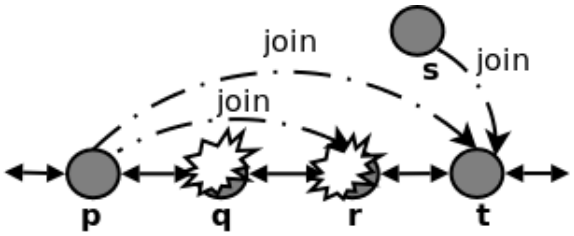


Fig. 4. Multiple failure recovery and simultaneous join. Peer p detects the crash of its successor q. First successor candidate r has also crashed. Peer p contacts t at the same time peer s tries to join the network. Both *join* messages are the same.

ysed. Figure 5 depicts a broken link and the crash of the tail of a branch. In the case of the broken link (inaccuracy), the failure recovery mechanism is triggered, but the successor of the suspected node will not accept the join message. The described algorithm will eventually recover from this situation when the failure detector eventually provides accurate information.

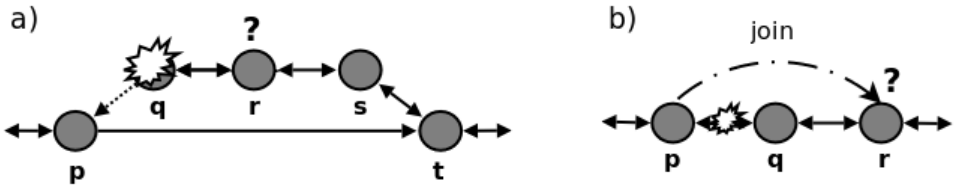


Fig. 5. Failures difficult to handle: (a) failure of the tail of branch, nobody is responsible for range $(p, q]$ (b) broken link generating a false suspicion of p about q.

In the case of the crash of the node at the tail of a branch, there is no predecessor to trigger the recovery mechanism. In this case, the successor could use one of its nodes in the predecessor list to trigger recovery, but that could introduce inconsistencies if the suspected node has not really failed. If the tail of the branch has not really failed but it has a broken link with its successor, then, it becomes

temporary isolated and unreachable to the rest of the network. Having unreachable nodes means that we are in presence of network partitioning. The following theorem describes the guarantees of the relaxed-ring in case of temporary failures with no network partitioning.

Theorem 2. Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.

Proof.

Every failure of a node is eventually detected by its successor, predecessor and other peers in the ring having a connection with the faulty node. The successor and other peers register the failure in the *crashed* set, and remove the faulty peer from the resilient sets *predlist* and *succlist*, but they do not trigger any recovery mechanism. Only the predecessor triggers failure recovery when the failure of its successor is detected, contacting only one peer from the successor list at the time. Then, there is only one possible candidate to replace each faulty peer, and then, it is impossible to have two responsible for the same range of keys. If a simultaneous join occurs (as in figure 4), there are two possible cases. If the recovery happens first, the join will just be as regular join. If the join happens first, the successor candidate will reject the recovery forwarding to the recovery to the new peer. This means that only one successor candidate for recovery will be contact at the time, preventing inconsistencies. □

The problem with respect to network partition is inherent to any overlay network, where a temporary uncertainty cannot be avoid, and some guarantees must be sacrificed. A deeper analysis is provided by Ghodsi [6], and it is related to the proof given in [3] about limitations of web services in presence of network partitioning.

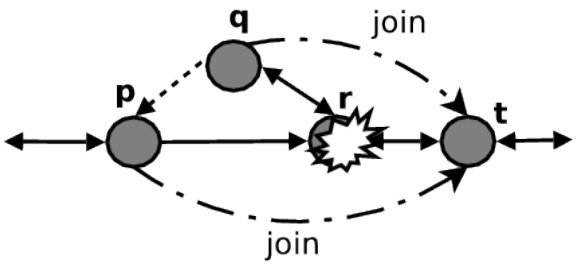


Fig. 6. The failure of the root of a branch triggers two recovery events

Figure 6 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 2 is based on the fact that per every failure detected, there is only one peer that triggers the recovery mechanism. In the case of the failure of the root of a branch, peer r in the example, there are two recovery messages triggered

by peers p and q. If message from peer q arrives first to peer t, the algorithm handle the situation without problems. If message from peer p arrives first, the branch will be temporary isolated, behaving as a network partition introducing a temporary inconsistency. This limitation of the relaxed-ring is well defined in the following theorem.

Theorem 3.

Let r be the root of a branch, *succ* its successor, *pred* its predecessor, and *predlist* the set of peers having r as successor. Let p be any peer in the set, so that $p \in predlist$. Then, the crash of peer r may introduce temporary inconsistent lookup if p contacts *succ* for recovery before *pred*. The inconsistency will involve the range $(p, pred]$, and it will be corrected as soon as *pred* contacts *succ* for recovery.

Proof. There are only two possible cases. First, *pred* contacts *succ* before p does it. In that case, *succ* will consider *pred* as its predecessor. When p contacts *succ*, it will redirect it to *pred* without introducing inconsistency. The second possible case is that p contacts *succ* first. At this stage, the range of responsibility of *succ* is $(p, succ]$, and of *pred* is $(p', pred]$, where $p' \in [p, pred]$. This implies that *succ* and *pred* are responsible for the range $(p', pred]$, where in the worse case $p' = p$. As soon as *pred* contacts *succ* it will become the predecessor because $pred > p$, and the inconsistency will disappear. □

Theorem 3 clearly states the limitation of branches in the systems, helping developers to identify the scenarios needing special failure recovery mechanisms. Since the problem is related to network partitioning, there seems to be no easy solution for it. An advantage of the relaxed-ring topology is that the issue is well defined and easy to detect, improving the guarantees provided by the system in order to build fault-tolerant applications on top of it.

3.4. *Resilient information*

During the starting and join algorithms we have mentioned *predlist* and *succlist* for resilient purposes. The basic failure recovery mechanism is triggered by a peer when it detects the failure of its successor. When this happens, the peer will contact the members of the successor list successively. The objective of the *predlist* is to recover from failures when there is no predecessor that triggers the recovery mechanism. This is expected to happen only when the tail of a branch has crashed.

Algorithm 7 describes how the update of the successor list is propagated while the list contains new information. The predecessor list is updated only during the join algorithm and upon failure recoveries.

4. Evaluation

This section is dedicated to the evaluation of the relaxed-ring. We analyse four aspects: the *amount of branches* that can appear on a network, the *size of branches*,

Algorithm 7 Update of successor list

```
1: upon event ⟨ upd_succlist | s, sl ⟩ do
2:     newsl := {s} ∪ sl \ getLast(sl)
3:     if (s == succ) ∧ (succlist ≠ newsl) then
4:         succlist := newsl
5:         send ⟨ upd_succlist | self, succlist ⟩ to pred
6:     end if
7: end event
```

the *number of messages* generated by the ring-maintenance protocol, and the verification of *lookup consistency* on unstable scenarios. The evaluation is done using a simulator implemented in Mozart [13, 10], where every node run autonomously on its own lightweight thread. Nodes communicate with each other by message passing using ports. We consider that these properties make the simulator more realistic. Every network is run several times using different seeds for random number generation. Charts are built using the average values of these executions.

4.1. *Branches and messages*

Figure 7 shows the amount of branches that can appear on networks with 1000 to 10000 nodes. The coefficient c represents the connectivity level of the network, where for instance $c = 0.95$ means that when a node contacts another one, there is only a 95% of probability that they will establish connection. A value of $c = 1.0$ means 100% of connectivity. On that value, no branches are created, meaning that the relaxed-ring behaves as a perfect ring on fault-free scenarios. The worse case corresponds to $c = 0.9$. In that case, we can observe that the amount of branches is less than 10% of the size of the network, as expected. Consider peers i and k, where i is the current predecessor of k. If they cannot talk to each either, k will form a branch. If another peer j joins in between i and k having good connection with both peers, the branch disappears.

On the contrary, if a node l joins the network between k and its successor, it will increase the size of the branch, decreasing the routing performance. For that reason, it is important to measure the average size of branches. If message *hint*, explained in section 3.2, works well for peer l, then, the branch will remain on size 1. Having this in mind, let us analyse figure 8. The average size of branches appears to be independent of the size of the network. The value is very similar for both cases where the quality of the connectivity is poor. In none of the cases the average is higher than 2 peers, which is a very reasonable value. If we want to analyse how the size of branches degrades routing performance of the whole network, we have to look at the average considering all nodes that belong to the core ring as providing branches of size 0. This value is represented by the curves *totalavg* on the figure. In both cases the value is smaller that 0.25. Experiments with 100% of connectivity are not shown because there are no branches, so the average size is always 0.

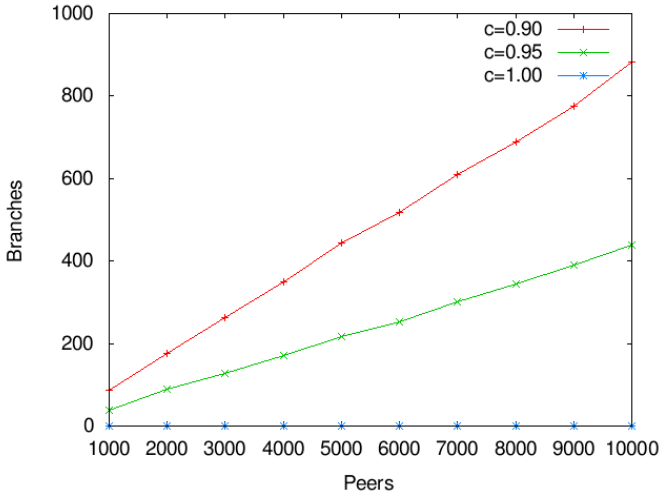


Fig. 7. Average amount of branches generated on networks with connectivity problems. Networks where tested with peers having a connectivity factor c, representing the probability of establishing a connection between peers, where $c \in \{0.9, 0.95, 1\}$.

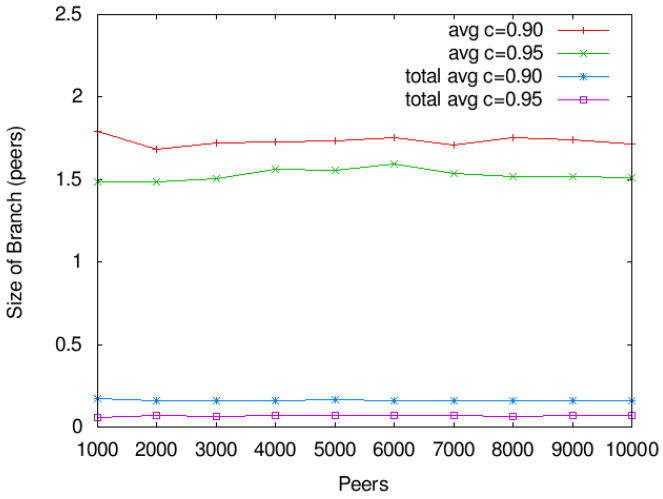


Fig. 8. Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

How many messages are exchanged by peers in order to maintain the relaxed-ring structure? How much is the contribution of the *hint* messages to the load in order to keep branches short? These questions are answered in figure 9. We can observe that the amount of messages increases linearly with the size of the network keeping

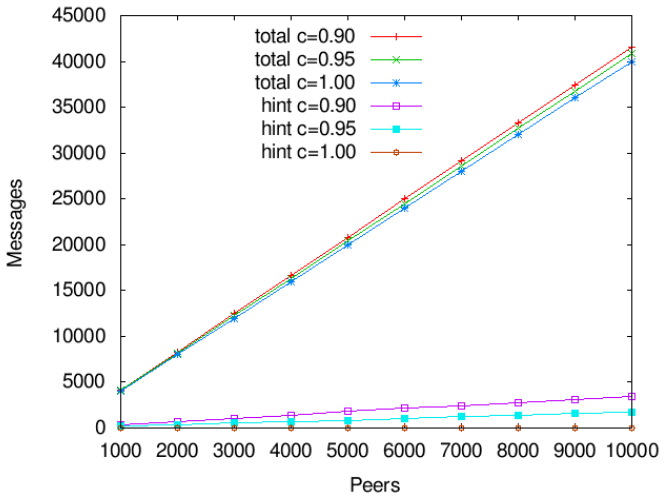


Fig. 9. Number of messages generated by the relaxed-ring maintenance. Three curves labeled *total* represent the total amount of messages exchanged between all peers depending on the connectivity coefficient. Curves labeled *hint* represent the contribution of *hint* messages to the total amount.

reasonable rates. The fault-free scenario has no *hint* messages as expected, but the total amount of messages is still pretty similar to the cases where connectivity is poor. This is because there are less normal join messages in case of failures, but this amount is compensated by the contribution of *hint* messages. We observe anyway that the contribution of *hint* messages remains low.

4.2. *Comparison with Chord*

We have also implemented Chord in our simulator. Experiments were only run in fault-free scenarios with full connectivity between peers, and thus, in better conditions than our experiments with the relaxed-ring. Even though, we observed many lookup inconsistencies on high churn. To reduce inconsistency, we trigger periodic stabilization on all nodes at different rates. The best results appeared after triggering stabilization after the join of every 4 nodes. We call this value stabilization rate. As seen in figure 10, the largest the network, the less inconsistencies are found. An inconsistency is detected when two *reachable* nodes are signalized as responsible for the same key. We can observe that stabilization rates of 5 converges pretty fast to 0 inconsistencies. Stabilization every 6 new joining peers only converge on networks of 4000 nodes. On the contrary, rate values of 7 and 8 presents immediately a high and non-decreasing amount of inconsistencies. Those networks would only converge if churn is reduced to 0. These values are compared with the worse case of the relaxed-ring (connectivity factor 0.9) where no inconsistencies where found.

We have observed that lookup consistency can be maintained in Chord at very good levels if periodic stabilization is triggered often enough. The problem is that

periodic stabilization demands a lot of resources. Figure 11 depicts the load related to every different stabilization rate. Logically, the worse case corresponds to most frequently triggered stabilization. If we only consider networks until 3000 nodes, it seems that the cost of periodic stabilization pays back for the level of lookup consistency that it offers, but this cost seems too expensive with larger networks.

In any case, the comparison with the relaxed-ring is considerable. While the relaxed-ring does not pass 5×10^4 messages for a network of 10000 nodes, a stabilization rate of 7 on a Chord network, starts already at 2×10^5 with the smallest network of 1000 nodes. Figure 11 clearly depicts the difference on the amount of messages sent. The point is that there are too many stabilization messages triggered without modifying the network. On the contrary, every join on the relaxed-ring generate more messages, but they are only triggered when they are needed.

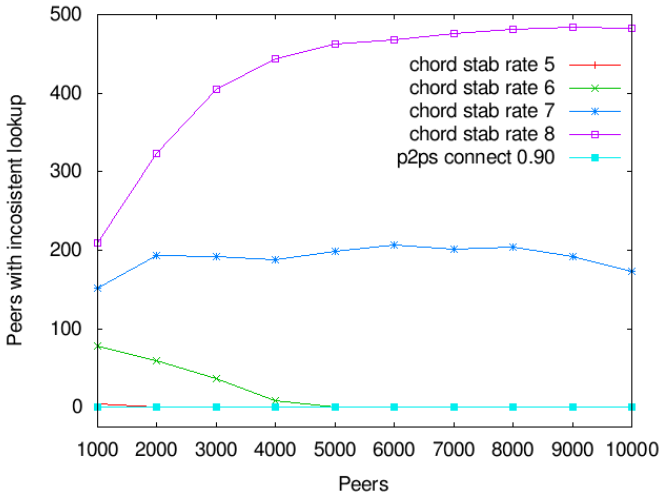


Fig. 10. Amount of peers with overlapping ranges of responsibilities, introducing lookup inconsistencies, on Chord networks under different stabilization rates for different network sizes. Comparison with the Relaxed-Ring (p2ps) with a bad connectivity. The stabilization rate represent the amount of peers joining/leaving the network between every stabilization round. The value of zero in the Y-axis has been raised in order to spot the curve of the Relaxed-Ring and Chord with a very frequent stabilization rate equal to 5.

5. Future Work

Apart from the simulator used for the validation, we have tested the Relaxed-Ring using a real implementation running distributed processes on small networks. We are currently testing our implementation on PlanetLab [1] to address more aggressive environments, and where we expect to report more about on failure recovery. The basic layers of P2PS providing point-to-point communication and the relaxed-

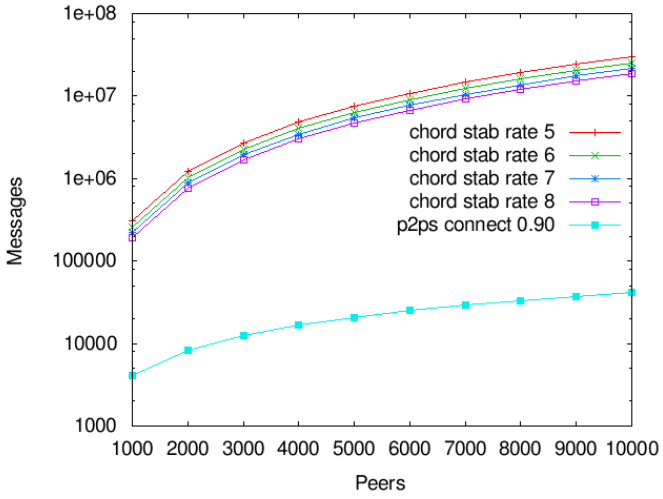


Fig. 11. Load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

ring maintenance are very stable. Our future work will be focused on the upper layers in order to deal with network partitioning. Apart from failure recovery, we are interested in building a service oriented architecture that will require a robust naming service and reliable broadcast. We also plan to build a replicated transactional distributed hash table based on a modified Paxos consensus algorithm [12].

6. Conclusion

In this paper we have presented a novel Relaxed-Ring topology for fault-tolerant and self-organizing peer-to-peer networks. The topology is derived from the simplification of the join algorithm requiring the synchronisation of only two peers at each stage. As a result, the algorithm introduces branches to the ring. These branches can only be observed in presence of connectivity problems between peers, and they help the system to work in realistic scenarios. The topology adds some complexity to the routing algorithm, but it does not degrade the complexity of its performance. We consider this issue a small drawback in comparison to the gain in fault tolerance and cost-efficiency in ring maintenance.

The topology makes feasible the integration of peers with very poor connectivity. Having a connection to a successor is sufficient to be part of the network. Leaving the network can be done instantaneously without having to follow a departure protocol, because the failure-recovery mechanism will deal with the missing node. The guarantees and limitations of the system are clearly identified and formally stated providing helpful indications in order to build fault-tolerant applications on top of this structured overlay network.

Acknowledgements

The authors would like to thank the *distoz* group at Universit catholique de Louvain and S. González for comments on this work. This research is mainly funded by SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] PlanetLab. *http://www.planet-lab.org*, 2008.
[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 344, Washington, DC, USA, 2003. IEEE Computer Society.
[3] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM Press.
[4] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.
[5] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://p2ps.info.ucl.ac.be*, 2008.
[6] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.
[7] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.
[8] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.
[9] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.
[10] Boris Mejías. CiNiSMO: Concurrent Network Simulator in Mozart-Oz, Université catholique de Louvain, Belgium. *http://p2ps.info.ucl.ac.be/cinismo*, 2008.
[11] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.
[12] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.
[13] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2008.
[14] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, June 2008.
[15] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.6 WinResMon: A Tool for Discovering Software Dependencies, Configuration and Requirements in Microsoft Windows

Abstract

This paper describes WinResMon, a system tool for determining resource usage and interactions among programs in Microsoft Windows environments. Think of WinResMon as a debugging tool to assist with software maintenance in a Microsoft Windows environment. It shows the current system state in terms of how resources are used and explains how the system arrived at that state. WinResMon can be used to determine how a program uses the registry and which files are needed by that program. WinResMon differs from other systems/tools in that it is integrated, designed to answer queries about resource usage and dependencies over time, and extensible, allowing the addition of new functions and tools.

Note

This is an earlier work which is not part of D1.3b deliverable but is an earlier system which has been enhanced as part of D1.3b. For example, network activity monitoring was added to WinResMon after the paper was published.

A.7 A Lightweight Binary Authentication System for Windows

A Lightweight Binary Authentication System for Windows

Felix Halim, Rajiv Ramnath, Sufatrio, Yongzheng Wu, Roland H.C. Yap

Abstract The problem of malware is greatly reduced if we can ensure that only software from trusted providers is executed. In this paper, we have built a prototype system on Windows which performs authentication of all binaries in Windows to ensure that only trusted software is executed and from the correct path. Binaries on Windows are made more complex because there are many kinds of binaries besides executables, e.g. DLLs, drivers, ActiveX controls, etc. We combine this with a simple software ID scheme for software management and vulnerability assessment which leverages on trusted infrastructure such as DNS and Certificate Authorities. Our prototype is lightweight and does not need to rely on PKI infrastructure; it does however take advantage of binaries with existing digital signatures. We provide a detailed security analysis of our authentication scheme. We demonstrate that our prototype has low overhead, around 2%, even when all binary code is authenticated.

1 Introduction

Malware such as viruses, trojan horses, worms, remote attacks, are a critical security threat today. A successful malware attack usually also modifies the environment (e.g. file system) of the compromised host. Many of the system security problems such as malware stem from the fact that untrusted code is executed on the system. We can mitigate many of these problems by ensuring that code which is executed only comes from trusted software providers/vendors and the code is executed in the correct context. In this paper, we show that this can be efficiently achieved even

Felix Halim, Yongzheng Wu, Roland H.C. Yap
School of Computing, National University of Singapore, e-mail: {halim,wuyongzh,ryap}@comp.nus.edu.sg

Rajiv Ramnath, Sufatrio
Temasek Laboratories, National University of Singapore, e-mail: {tslrr,tslsufat@nus.edu.sg}

on complex operating systems such as Windows. Our system provides two guarantees: (i) we only allow the execution of binaries (in the rest of the paper, we refer to any executable code stored in the file system as a *binary*) whose contents are already known and trusted — we call this *authenticating binary integrity*; and (ii) as binaries are kept in files, the pathname of the file must match its content — we call this *authenticating binary location*. Binary integrity authentication ensures that the binary has not been tampered with, e.g. `cmd.exe` is not a trojan. Binary location authentication ensures that we are executing the correct executable content. A following extreme example illustrates location authentication. Suppose the binary integrity of the shell and the file system format executables are verified. If an attacker swaps their pathnames, then running a shell would cause the file system to be formatted. In this paper, we refer to *binary authentication* to mean when both the binary's data integrity and location are verified.

Most operating systems can prevent execution of code on the stack due to buffer overflow, e.g. NX protection. Combining stack protection with binary authentication makes the remaining avenues for attack smaller and more difficult. Binary authentication is also beneficial because it is even more important for the operating system to be protected against malicious drivers and the loading of malware into the kernel.

Most work on binary integrity authentication is on Unix/Linux [1, 2, 3]. However, the problem of malware is more acute in Windows. There are also many types of executable code, e.g. executables (.exe), dynamic linked libraries (.dll), ActiveX controls, control panel applets, and drivers. In this paper, we focus on mandatory binary authentication of all forms of executables in Windows. Binaries which fail authentication cannot be loaded, thus, cannot be executed. We argue that binary authentication together with execute protection of memory regions (e.g. Windows data execution prevention) provides protection against most of the malware on Windows.

A binary authentication needs to be flexible to operate under different scenarios. Our prototype signs binaries using a HMAC [4] which is more lightweight than having to rely on PKI infrastructure, although it can also make use of it. The authentication scheme additionally allows for other security benefits. Not only is it important to authenticate software on a system but one also needs to deal with the maintenance of the software over time. Nowadays, the number of discovered vulnerabilities grows rapidly [5]. This means that binaries on a system (even if they are authenticated) may be vulnerable. This leads to a vulnerability management and patching problems. We propose a simple software ID system leveraging on the binary authentication infrastructure and existing infrastructures such as DNS and certificae authorities to handle this problem.

Windows has the Authenticode mechanism [6]. In Windows XP version and earlier, it alerts the users of the results of signature verification under a few situations. However, it is not mandatory, and can be bypassed. The Windows Vista UAC mechanism makes use of signed binaries but it only deals with EXE binaries. It is also limited to privilege escalation situations. One common drawback of existing Windows mechanisms is that they do not authenticate the binary location. Moreover, requiring PKI infrastructure and certificates, we believe, is too heavy for a general purpose mechanism.

The main contribution of this paper is that we believe that it provides the first comprehensive infrastructure for trusted binaries for Windows. This is significant given that much of the problems of security on Windows stems from inability to distinguish between trusted and untrusted software. It provides mandatory authentication for the full range of binaries under Windows, and goes beyond authenticated code in XP and Vista. We also protect driver loading which gives increased kernel protection. Our scheme provides mandatory driver authentication which 32-bit Windows does not, and can be integrated with more flexible policies which 64-bit Windows does not support. We also analyze the security of our system. Our benchmarking shows that the overhead of comprehensive binary authentication can be quite low, around 2%, with a caching strategy.

2 Windows Issues

We discuss below the complexities and special problems of Windows which make it more difficult to implement binary authentication than in other operating systems such as Unix. Windows NT (Server 2000, XP, Server 2003, Vista) is a microkernel-like operating system. Programs are usually written for the Win32 API but these are decomposed into microkernel operations. However, Windows is closed source — only the Win32 API is documented and not the microkernel API. Our prototype makes use of both the documented and undocumented kernel infrastructure. However, it is not possible to make any guarantees on the completeness of the security mechanisms (which would also be a challenge even if Windows was open source). Some of the specific issues in Windows which we deal with are:

- **Proliferation of Binary Types:** It is not sufficient to ensure the integrity of `EXE` files. In Windows, binaries can have any file name extension, or even no extension. Some of the most common extensions include `EXE` (regular executables), `DLL` (dynamic linked libraries), `OCX` (ActiveX controls), `SYS` (drivers) and `CPL` (control panel applets). Unlike Unix, binaries cannot be distinguised by an execution flag. Thus, without reading its contents, it is not possible to distinguish a binary from any other file.
- **Complex Process Execution:** A process is created using `CreateProcess()` which is a Win32 library function. However, this is not a system call since Windows is a microkernel, and in reality this is broken up at the native API into: `NTCreateFile()`, `NTCreateSection()`, `NTMapViewOfSection()`, `NTCreateProcess()`,`NTCreateThread()`. Notice that `NTCreateProcess()` at the microkernel level performs only a small part of what is needed to run a process. Due to this, it is more complex to incorporate mandatory authentication in Windows.
- **DLL loading:** To load a DLL, a process usually uses the Win32 API, `LoadLibrary()`. However, this is broken up in a similar way to process execution above.
- **Execute Permissions:** Many code signing systems, particularly those on Linux [1, 3], implement binary loading by examining the execute permission bit in the

access mode of file open system-call. The same mechanism, however, does not work in Windows. Windows programs often set their file modes in a more permissive manner. Simply denying a file opening with execute mode set when its authentication fails, will cause many programs to fail which are otherwise correct on Windows. Instead, we need to properly intercept the right API(s) with correctly intended operation semantics to respect Windows behavior.

Compared to other open platforms, Windows potentially also makes the issue of locating vulnerable software components more complicated. A great deal of binaries created by Microsoft contain an internal file version, which is stored as the file's meta-data. The Windows update process does not indicate to the user which files are modified. Moreover, meta data of the modified file might still be kept the same. Thus, it is difficult to keep track of files changes in Windows. More precisely, one cannot ensure whether a version of a program P_i remain vulnerable to an attack A. It is rather difficult for a typical administrator who examines vulnerability information from public advisories to trace through the system and pinpoint the exact affected components. Our software naming scheme, associates binaries with their version and simplifies software vulnerability management.

3 Related Work

Tripwire [9] is one of the first to do file integrity protection but is limited as it in user-mode program and checks file integrity off-line. It does not provide any mandatory form of integrity checking and there are many known attacks such as: file modification in between authentication times, and attacks on system daemons (e.g. cron and sendmail) and system files that it depends on [10, 11].

There a number of kernel level binary level authentication implementations. These are mainly for Unix such as DigSig [1], Trojanproof [2] and SignedExec [3], which modify the Unix kernel to verify the executable's digital signature before program execution. DigSig and SignedExec embed signatures within the the elf binaries. For efficiency, DigSig employs a caching mechanism to avoid checking binaries which have been verified already. The mechanism is similar to ours here but we need to handle the problems of Windows. It appears that DigSig provides binary integrity authentication but not binary location authentication.[1] In this paper, we examine the implementation issues and tradeoffs for Windows which is more complex and difficult than in Unix.

Authenticode [6] is Microsoft infrastructure for digitally signing binaries. In Windows versions prior to Vista, such XP with SP2, it is used as follows:

1. During ActiveX installation: Internet Explorer uses Authenticode to examine the ActiveX plugin and shows a prompt which contains the publisher's information including the result of the signature check.

[1] Mechanisms based solely on signatures embedded in the binaries do not have sufficient information for binary location authentication.

2. A user downloads a file using Internet Explorer: If this file is executed using the Windows Explorer shell, a prompt is displayed giving the signed the publisher's information. Internet Explorer uses an NTFS feature called Alternate Data Streams to embed the Internet zone information –in this case, the Internet– into the file. The Windows Explorer shell detects the zone information and displays the prompt. This mechanism is not mandatory and relies on the use of zone-aware programs, the browser and GUI shell cooperating with each other. Thus, it can be bypassed.

Since Authenticode runs in user space, it can be bypassed in a number of ways, e.g. from the command shell. It is also limited to files downloaded using Internet Explorer. Only the EXE binary is examined by Authenticode, but DLLs are ignored. One possible attack is then to put malware into a DLL and then execute it, e.g. with `rundll32.exe`. Furthermore, Authenticode relies heavily on digital certificates. Checking Certificate Revocation Lists (CRL) may add extra delay including timeouts due to the need to contact CA. In some cases, this causes significant slowdown.

The latest Windows Vista improves on signed checking because User Account Control (UAC) can be configured for mandatory checking of signed executables. However, this is quite limited since the UAC mechanism only kicks in when a process requests privileged elevation, and for certain operations on protected resources. UAC is not user friendly since there is a need for constant interactive user approval. Vista does not seem to prevent the loading of unsigned DLLs and other non EXE binaries. The 32-bit versions of Windows (including Vista) do not checked whether drivers are signed. However, the 64 bit versions (XP, Server 2003 and Vista) require all drivers to be signed (this may be too strict and restrict hardware choices).

The closest work on binary authentication in Windows is the Emu system in by Schmid et al. [13]. They intercept process creation by intercepting the NtCreateProcess system call. It is unclear whether they are able authenticate all binary code since trapping at NtCreateProcess is not sufficient to deal with DLLs. No performance benchmarks are given, so it is unclear how if their system is efficient.

4 Binary Authentication and Software IDs

We want a lightweight binary authentication scheme which can work under many settings without too much reliance on other infrastructure. Furthermore, it should help in the management of binaries, and incurs low overhead. Management of binaries includes determining which binaries should be authentic, dealing with issues arising from disclosed vulnerabilities, and software patching.

4.1 Software ID Scheme

We complement binary authentication with a software ID scheme meant to simplify binary management issues. The idea is that a *software ID* associates a unique string to a particular binary of a software product. The software ID should come either from the software developer or alternatively be assigned by the system administrator. The key to ensuring unique software_ID, even among different software developers, lies on the standardized format of the ID. We can define software_ID as follows:

$$\text{Software\_ID} ::= \langle\ \text{opcode\_tag} \,||\, \text{vendor\_ID} \,||\, \text{product\_ID} \,||\, \text{module\_ID} \,||\, \text{version\_ID} \rangle.\ ^{2}$$

Here, || denotes string concatenation. Opcode_tag distinguishes different naming convention, eg. $Software\_ID$ and $Custom\_ID$ defined below.

Ideally, we want to be able to uniquely assigning vendor_IDs to producers of software which can make the software_ID unique. This problem in practice might not be as difficult as it sounds since it is similar to domain name registration or the assignment of Medium Access Control (MAC) addresses by network card manufacturers. One can leverage on existing trust infrastructures to do this. For example, the responsibility for unique and well known software_IDs can be assigned to a Certificate Authority (CA), which then define the vendor_ID as $\langle$ CA_ID || vendor_name $\rangle$. Alternatively, one might be able to use the domain name of the software developer as a proxy for the vendor_ID.

A software_ID gives a one to one mapping between the binary and its ID string. This is useful for dealing with vulnerability management problems [15]. Suppose a new vulnerability is known for a particular version of a software. This means that certain binaries, providing that they correspond to that software version may be vulnerable. However, there is no simple and standard way of automatically determining this version information. Once we have software_IDs associated with binaries then one can check the software_ID against vulnerability alerts. The advisory may already contain the software_ID. Automatic scanners can then be used to tie-in this checking with the dissemination of vulnerability alerts to automatically monitor/manage/patch the software in an operating system. General management of patches in an operating system can also be done in much the same way.

In the case where no software_ID comes with a software product, one can alternatively derive one. It can be constructed, for instance, using the following (coarse-grained) string naming:

$$Custom\_ID ::= \langle\ \text{opcode\_tag} \,||\, hash(\text{vendor\_URL} + \text{product\_name} + \text{file\_name} + \text{salt})\ \rangle.$$

The salt expands the name space to reduce the risk of a hash function collision.

---

[2] Module_ID suffices to deal with software versioning. Having a separate version_ID, however, is useful to easily track different versions (or patched versions) of the same program.

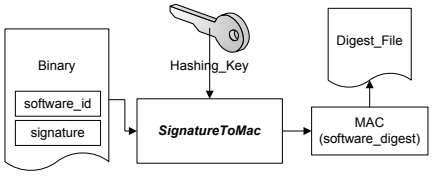


Fig. 1 SignatureToMac: Deriving the MAC

4.2 Binary Authentication System Architecture

In the following discussion, we assume here that binaries already come tagged with a software_ID. During the binary authentication set-up, preferably done immediately after the targeted binary installation, we generate the MAC values for each binary. In the case where binaries are digitally signed by its developer, then we verify the signature and then generate the MAC for each binary. Thus, only one public-key operation needs to be done at install time. We choose to use a keyed hash, the HMAC algorithm [4], so there is a secret key for the administrator. This is mainly to increase the security of the stored hashes. To authenticate binary integrity for any future execution of the code, only the generated HMAC needs to be checked. In what follows, we mostly write MAC which already covers the choice of HMAC.

One way of storing the generated MAC is by embedding it into the binary. However, doing so may interfere with file format of the signed binaries and may also have other complications. We instead use an authentication repository file which stores all the MAC values of authenticated binaries with their pathnames. During the boot-up process, the kernel creates its own in-memory data structures for binary authentication from this file. We ensure that the repository file is protected from further modification except under the control of the authentication system to add/remove binaries.[3] We can also customize binary authentication on a per user basis rather than system-wide which is a white-list of binaries approved for execution. In the case, when the initial binary does not have a digital signature, then the administrator can still choose to approve the binary and generate a MAC for it.

There are two main components of the system: the **SignatureToMac** and **Verifier**. The SignatureToMac maintains the authentication repository, *Digest_file*, consisting of ⟨path, MAC⟩ tuples. The Verifier is a kernel driver which makes use of *Digest_file* and decides whether an execution is to be allowed.

4.2.1 SignatureToMac

Once software is installed on the system, Fig. 1 shows how SignatureToMac processes the binaries:

[3] Further security can be achieved by integrating binary authentication with a TPM infrastructure. We do not do so in the prototype as that is somewhat orthogonal.

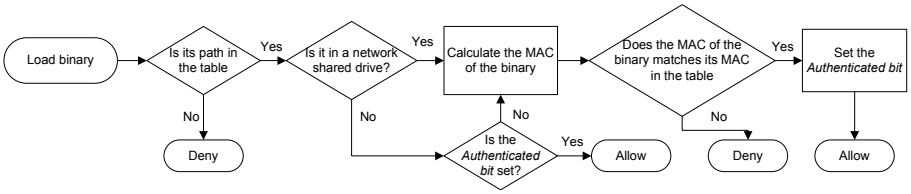


Fig. 2 Verifier: The Verifier in-kernel authentication process

1. Checks the validity of the binary's digital signature. If the signature is invalid, then report failure.
2. It consults the user or system administrator whether the software is to be trusted or not (this is similar to the Vista UAC dialog but only happens once). Other policies (possibly mandatory) can also be implemented.
3. It generates the MAC of the binary (including software_ID string) using a secret key, *Hashing_key*, to produce *software_digest*. The *Hashing_key* is only accessible by the authentication system, e.g. obtained on bootup.
4. It adds an entry for the binary as a tuple ⟨*path name*, *software_digest*⟩ into the *Digest_file* repository and informs the Verifier. The repository is protected against modification. Note that because the entries are signed, the repository can be read for other uses, e.g. version control and vulnerability management.

4.2.2 Verifier

The Verifier performs mandatory binary authentication — it denies the execution of any kind of Windows binary which fails to match the MAC and pathname. There are two general approaches for the checking. One is *cached MAC* which avoids generating the MAC for previously authenticated binaries. The other is *uncached MAC* which always checks the MAC. As we will see, they have various tradeoffs. The cached MAC implementation needs to ensure that binaries are unmodified. Hence, the Verifier monitors the usage of previously authenticated files on the cache, and removes them from the cache if it can be potentially modified.

The core data structure of the Verifier component can be viewed as a table of tuples in the form ⟨*Kernel_path*, *FileID*, *MAC*, *Authenticated_bit*⟩ representing the allowed binaries. It is indexed on *Kernel_path* and *FileID* for fast lookup. The fields are as follows:

- The *Kernel_path* is Windows kernel (internal) pathname representation of a file. In Window's user space, a file can have multiple absolute pathnames, due to: (i) 8.3 file naming format, e.g. "`C:\Program Files\`" and "`C:\progra~1\`" are the same; (ii) symbolic links (reparse points is similar); (iii) hard links; (iv) volume mount points; or (v) the `SUBST` and `APPEND` DOS commands. The *Kernel_path* is a unique representation for all the possible pathnames. When the

system loads ⟨*path name*, *software_digest*⟩ from *Digest_file* during the startup, *path name* is converted to *Kernel_path* since all subsequent checks by Verifier in the kernel all use the latter.

- The *FileID* is a pair of ⟨*device_name*, *NTFS_object_ID*⟩. The *device_name* is a Windows internal name to identify a disk or partition volume. For instance, the device name `HarddiskVolume1` usually refers to `C:\`. The *NTFS_object_ID* is a 128-bit length number uniquely identifying a file in the file system volume (this is not the same as Unix inode numbers) The Verifier uses the *FileID* to identify the same file given more than one hard link. This prevents an attacker from creating a hard link for modifying a binary without invalidating the binary cache. The *FileID* values will be queried from the system and filled into the table during system boot.
- The *MAC* is same as a *software_digest* entry in *Digest_file*. Our prototype implements the HMAC-MD5 [4], HMAC-SHA-1 and HMAC-SHA-256 [16] hash algorithms.[4]
- The *Authenticated_bit* remembers whether the binary has been previously authenticated. It is initially set to false, and set to true after successful authentication.

Fig. 2 shows the authentication process when a binary executes/loads:

1. It checks if the binary's *Kernel_pathname* exists in the table. If not, then deny the execution and optionally log the event. A notification is accordingly sent to the user.
2. It the file is on a network shared drive, goto step 4. The MAC is always recomputed as we cannot keep track of modification to files on network shares.
3. If the *Authenticated_bit* is set go to step 7.
4. It performs MAC algorithm operation on the binary.
5. If the resulting MAC doesn't match with the *MAC* stored in the table, execution is denied.
6. It sets the *Authenticated_bit* of the binary.
7. It passes the control to the kernel to continue the execution.

To control binary execution, we intercept the section creation action (`NtCreateSection()` system call) which is better than:

- Intercepting file opening (`NtCreateFile()` and `NtOpenFile()`). This would need to authenticate any file opened with with execute access mode. As discussed in Sec. 2, however, this introduces unnecessary overheads and can cause some correct programs to fail if the files do not pass authentication. There are also technical difficulties to distinguish between process creation and regular file IO operations, which is not always easy given its microkernel nature.
- Intercepting process creation (`NtCreateProcess()`). This method is not effective for our purpose. Firstly, we cannot use it to control `DLL` loading. Secondly, it is more difficult to get the pathname of the binary because process creating is broken down into microkernel operations.

[4] Due to recent concerns which show weaknesses and attacks against MD5 [17], we also have stronger hash functions, namely SHA-1 and the stronger SHA-256.

It turns out that since all code from any kind of binary needs to have a memory section to execute, it suffices to intercept `NTCreateSection()`.

The cached MAC verifier needs to ensure that binaries which have been already authenticated are not modified. However, the uncached verifier will not need to perform file monitoring. A binary with pathname *P* is considered modified, if the following occurs:

- *P* is created: Hence, we monitor system call `NtCreateFile()` and `NtOpenFile()`.
- *P* is opened with write access mode: the previous two system calls are also intercepted for this purpose. [5]
- Another file is renamed to *P*: We monitor file renaming (`NtSetInformationFile(FileRenameInformation)`) system call.
- A drive containing *P* is mounted: We monitor drive mounting `IRP_MJ_VOLUME_MOUNT`.

Note that we do not need to monitor file deletion since we only care about executing correct files but not missing files. The details of file modification monitoring are given in Fig. 3.

Upon modification of *P*, we reset the *Authenticated_bit* of binary *P*, and update the *FileID* in the table if it is changed. Should FAT file system be used, pathname is used to identify the binary as *FileID* is not supported but neither are hard and soft links. Since *FileID* is optional and can be removed, we monitor *FileID* removal (`NtFsControlFile(FSCTL_DELETE_OBJECT_ID)`) and deny the removal if the *FileID* is in the table. Dut to the semantics of NTFS, our use of *FileID* can coexist with other applications using it.

If additional hardware and infrastructure is available to support secure booting, such as the Trusted Platform Module (TPM) initiative, the system can benefit from increased security. Offline attacks would have to first attack the TPM. The Hashing_key can also be stored securely by the TPM.

4.3 Security Analysis

The security of binary authentication relies on the strength of the chosen hash functions (MD5, SHA-1, SHA-256) as well as the HMAC algorithm. Thus, we assume that any change in a binary can be detected through a changed MAC.

In our authentication on binary with digital signature, the subsequent invocations using MAC verification is sufficient to ensure the authenticity of the binary. In other words, MAC authentications "*preserve*" the previously established properties of binary authentication derived from digital signature. A subtlety comes when the

[5] An alternative way is to monitor the file (block) writing operation (`NtWriteFile()`). However, it is less efficient because file block writings take place more frequently than file openings as one opened file for modification might be subject to multiple block writings. Furthermore, it cannot capture file-memory mapping.

```
procedure UponModification (FilePath)
   if (FS is NTFS)
      FileID := GetFileID(FilePath) # FileID can be NULL
      if (FilePath is in the table)
         Entry := LookupTableByPath(FilePath)
         if (FileID == NULL)
            # this can happen when the file is deleted and created again.
            # generate a new FileID and update the table
            Entry.FileID := CreateFileID(FilePath)
         else if ( FileID != Entry.FileID in the table)
            # this can happen when the drive is unmounted,
            # id changed off-line and re-mounted
            Entry.FileID := FileID
         end if
         Entry.Authenticated := false
      else if ((FileID != NULL) AND (FileID is in the table))
         Entry := LookupTableByID(FileID)
         Entry.Authenticated := false
      end if
   else if ((FS is FAT) AND (FilePath is in table))
      Entry := LookupTableByPath(FilePath)
      Entry.Authenticated := false
   end if
end procedure
```

Fig. 3 Pseudo code of file modification monitor

certificate expires or is revoked at some point in time after SignatureToMac. We view that the question of whether one should keep trusting the binary for execution depends on one's level of trust on certificate expiration/revocation. If the certificate expiration or revocation means that the public key must no longer be used, but the fact that *previously established* goodness binary properties still hold, then we can keep trusting the binary for execution (as long as we still believe the issuer).

Here we discuss some possible attacks to the authentication system. All the attacks except the last two target the caching system. More precisely, the attacker attempts to modify an already authenticated binary without causing the *Authenticated_bit* to be set to false.

- **Manipulating symbolic links:** The attacker can use the path S which is a symbolic link of an authenticated P to indirectly modify P and subsequently execute P. However, the modified file will not be executed successfully, because Windows kernel resolves symbolic links to real paths. More precisely, the symbolic link S is resolved to the real path P. As a result, the *Authenticated_bit* of P will be set to false. When P is executed, its MAC will be recalculated and it will not pass the authentication.
- **Manipulating hard links:** The attacker can create a hard link H on an already authenticated file P and then modifies the file using path H. This attack will not succeed because we use *FileID* to identify files. H has the same *FileID* as P, thus the *Authenticated_bit* will be set to false. Note that this attack will not succeed in FAT file system either, even though we cannot use *FileID*. This is because hard link is not supported in FAT.
- **Manipulating FileID:** Recall that *FileID* consists of *device_name* and *NTFS_object_ID*. The latter is optional and thus can be removed. The attacker can re-

move the *NTFS_object_ID*, and then performs the previous attack. We handle this attack by denying *NTFS_object_ID* removal on authenticated files. This is implemented by monitoring the file system control event `FSCTL_DELETE_OBJECT_ID`

- **Remote File Systems:** Since we cannot keep track of modification on a network shared file system, we do not cache the authentication. More precisely, the MAC of the binary is always calculated upon loading. Same applies to removable media such as floppy in which we can not keep track of modification of files.
- **TOCTTOU:** TOCTTOU stands for Time-Of-Check-To-Time-Of-Use. It refers to a race condition bug of an access control system where the resource is changed during the time of checking the resource to the time of using the resource. In the binary authentication context, the binary may be modified after the time it is authenticated and before the time it is executed. However, we observed that all binaries are exclusive-write-locked when it is opened. That means binaries cannot be modified from the time it is opened to the time it is closed. Also note that the file is authenticated after it is opened and before it is executed. As a result, binaries cannot be modified during TOCTTOU.
  When the binary is in a network shared volume, i.e. SMB share, and the write-lock is not properly implemented in the SMB server, an attacker is able to modify the binary after authentication. However, we have observed that both Windows and Samba implement write-lock properly. Thus the attack is only possible when the SMB server is compromised. One way to prevent this is to disallow binary loading from SMB share.
- **Driver Loading:** The binary authentication system authenticates all binaries including kernel driver. This means all drivers are authenticated thus driver attacks such as kernel rootkits and malware drivers can be prevented.
- **Offline Attack:** Offline attack means modification of the file system when Windows is not in control. For example, boot another OS or remove the disk drive for modification elsewhere. Such an attack will require physical access to the machine. Offline attack can corrupt data or change programs/files and affect the general functioning and we cannot prevent that. What we can do is to ensure the integrity of executable code and other data loaded in memory for processes.
  We assume that the kernel is still secure, i.e. authentication occurs early in the boot. We also assume that kernel functioning is not impaired, e.g. deleting some system files does not cause the kernel to have an exploitable vulnerability.
  Since the hashing_key is not stored in the machine, it is not available to the attacker. The attacker can still change the digest file and the binaries, however, MACs of modified binaries cannot be produced without the hashing_key. Thus modified binaries cannot be executed when the system is online.

5 Empirical Results

Our authentication system can detect when a modified binary is loaded or run from the wrong pathname. In this section, we examine the three factors which impact on

system performance: (i) Verifier checking upon binary loading (execution); (ii) file modification monitoring; and (iii) binary set-up during the SignatureToMac process. The first two above are the most important as they directly affect user's waiting time for process execution and affect overall system operation. The tests here are meant to determine the worst case overhead as well as average overheads.

The benchmarks are run on a Core 2 Duo with 2GB of ram running Windows XP with SP2. Each benchmark is run five times. As we want to investigate the effect of the cached Verifier, each benchmark is run with caching and without caching. When caching is enabled, we ignore the result of the first run because the overhead of authentication overhead is already shown in the uncached case. Even if we count the first run, its impact will be very small because some of the microbenchmarks run for 10K times, so the authentication overhead becomes negligible.

To see the difference of using different hashing algorithms, we implement and benchmark three algorithms: MD5, SHA-1 and SHA-256. Only MD5 and SHA-256 are shown in Table 1 as the results of SHA-1 are always between these. When caching is enabled, results of different hashing algorithms are not distinguished (shown as Cached-MAC in the table), because binaries are not require MAC checking during the benchmark. The reason is that the first run is ignored, and the binaries are not modified during the benchmark.

To see the difference with digital signature based authentication system, we also compare the performances of our scheme against the Microsoft official Authenticode utility called `Sign Tool` [18], and another Sysinternals (now acquired by Microsoft) Authenticode utility `Sigcheck` [19]. Note that two tools are user-mode programs. They are there to illustrate the difference between non-mandatory strategies used with Authenticode with our in-kernel mandatory authentication.

The first two benchmarks investigate system performance under two scenarios:

1. **Micro-benchmark:** The micro-benchmark aims to measure the worst case performance overhead incurred by the scheme. Note that this is primarily intended to measure the authentication cost but not other overhead, which is done by the last file modification microbenchmark. Here, we have two micro-benchmark scenarios.
   a. **EXE Loading:** This executes the `noop.exe` program, a dummy program that immediately exits, for 10K times. This scenario measures the overhead for authenticating the `EXE` file. The benchmark program first calls `CreateProcess()`, and waits for the child process' termination using the `WaitForSingleObject()` function. We use different binary sizes (40KB, 400KB, 4MB and 40MB, only the 40K and 40MB results are displayed) for noop.exe to see how executable size impacts performance.
   b. **Loading DLL:** The second scenario executes the `load-dll.exe` program for 100 times. This scenario is used to find out how the number of loaded DLLs impacts the performance. Program `load-dll.exe` loads 278 standard Microsoft DLLs with a total file size of $\sim$75MB. The size of the `load-dll.exe` itself is 60KB. Note that in Windows, the bulk of code is

Authentication System	Micro-Benchmark: noop 40K time	noop 40K slowdown	noop 40M time	noop 40M slowdown	load-dll time	load-dll slowdown	Macro-Benchmark: build time	build slowdown
Clean	22.76	–	30.07	–	45.32	–	66.26	–
EXE Only:								
Signtool	2822	11637%	4850	16033%	73.49	62.16%	97.00	46.39%
Sigcheck	1720	7457%	5629	18623%	62.82	38.62%	110.5	66.72%
Uncached-MD5	25.96	14.08%	2150	7052%	45.34	0.05%	70.85	6.93%
Uncached-SHA256	30.29	33.07%	9005	29851%	45.34	0.05%	71.79	8.35%
Cached-MAC	23.20	**1.93%**	30.63	**1.88%**	45.33	**0.02%**	67.62	**2.06%**
All Binaries:								
Signtool	11867	52043%	14030	46565%	16018	35244%	–	–
Sigcheck	4283	18772%	6186	20478%	12548	27587%	–	–
Uncached-MD5	26.10	14.67%	3881	12811%	128.8	184.1%	79.31	19.69%
Uncached-SHA256	30.42	33.67%	9302	30839%	201.3	344.0%	91.80	38.55%
Cached-MAC	23.25	**2.14%**	30.58	**1.72%**	45.35	**0.07%**	67.88	**2.45%**

Table 1 Benchmark results showing times (in seconds) and slowdown factors. The worst slowdown factors for each benchmark scenario are shown with underline, whereas the best are in bold. We define $slowdown_x = (time_x - time_{clean})/time_{clean}$.

often in DLLs which is why the EXE file may be small, e.g. Open Office has over 300 DLLs.

2. **Macro-benchmark:** The macro-benchmark measures overhead under a typical usage scenario. Our benchmark is to create the Windows DDK sample projects using the build command. In each test run, 482 C/C++ source files in 43 projects are built. This benchmark is chosen as it is deterministic, non-interactive, creates many processes and uses many files.

We benchmark Sign Tool and Sigcheck in the following fashion. We first sign noop.exe and load-dll.exe using Sign Tool's signing operation. We then measure the execution time of authenticating and executing the two programs. For the macro-benchmark, we replace each development tool in the DDK (i.e. *build.exe*, *nmake.exe*, *cl.exe* and *link.exe*) with a wrapper program which first authenticates the actual development tool and then invokes it. For the micro-benchmark, we consider two settings: (i) EXE only; and (ii) all binaries (EXE + DLL). The macro-benchmark, however, only tests the EXE case. This is because, during the macro-benchmark, many programs are invoked, and each program each invocation may dynamically load a different set of DLLs. Thus, it is hard to keep track of what DLLs are loaded, and it is unfair to simulate with all DLLs used.

The results are given in Table 1 but we have not shown "noop 400K" and "noop 4M" because they are bounded by the results of "noop 40K" and "noop 40M". Other results not shown are that the overhead is approximately linear with respect to the file size, e.g. the results of the All-binaries/Uncached/SHA-256 benchmarks are 40K:30.42s, 400K:85.43s, 4M:598.0s, 40M:9302s.

We can see that the overhead of Signtool and Sigcheck makes it unusable if DLLs are to be checked (352x slower on load-dll). If only EXE are checked, then

at least 40% overhead and based on the `load-dll` benchmark, one could expect about an order of magnitude worse if all DLLS are checked. Of course, using these tools would incur additional overhead from creating a process and the main purpose is just to show the difference between what can be done in user-mode versus in-kernel. We can see that all the uncached-MD5/SHA256 are considerably faster than `Signtool` and `Sigcheck`.

Authenticating only `EXE`, the difference between uncached has overheads around 8% while cached brings this down to very small, around 2%x and almost neglible in the `load-dll` benchmark (0.02%). Note that as uncached overhead is quite small, the results are dominated by non-determinism in timing measurements. Moving to all DLLs (`EXE` + `DLL`), we can see the effect of Windows programs using many DLLs (more code in `DLL` than `EXE`). The overhead incurred by caching is still small while uncached can grow to between 20-40% depending on the hash algorithm. Note that the uncached overhead is applicable for files which cannot be cached.

The final microbenchmark investigates the tradeoffs between cached and uncached verification. Caching means that MAC verification is amortized over executions but has added overhead from monitoring file modification, while uncached is the opposite. Our micro-benchmark opens a file for writing 100K times to measure the worst case overhead incurred by file modification monitoring. We have 3 experiments: (i) a clean system without binary authentication; (ii) binary authentication with cache and the modified file is a binary; and (iii) binary authentication with cache and the modified file is not a binary.

The results for the file modification micro-benchmark show that for binary authentication with a cache, it doesn't matter whether the file being written to is a binary or not. Both cases incur about 60% overhead compared to a clean system. Since binary authentication with no-cache has no overheads for file modifications, this means that under some usage scenarios where file modification is very high, the uncached strategy may be preferable over cached even when Verifier overhead is higher.

6 Conclusion

We have shown a comprehensive system which authenticates both content and pathname for Windows to ensure that only trusted binaries are executed. Unlike other operating systems, Windows poses significant challenges. We show that it is possible to ensure that only trusted binaries can be loaded from files for execution. This can also be combined with a simple software ID scheme which simplifies binary version management, and dealing with vulnerability alerts and patches. Our system is lightweight and integrates well with PKI and trust mechanisms without having to rely on them. The overheads of our prototype are quite low when caching is is used. In the case of workloads with heavy file modifications, an uncached strategy might be preferable. The overheads are still low in this case, since the system overhead will be dominated by I/O rather than binary authentication, so the overall binary authen-

tication would still be low as a percentage of overall system overhead. In summary, although this is a prototype, it significantly adds to the security of any Windows system but at the same time is sufficiently flexible so that it can be tailored for different usage scenarios.

References

1. A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi and V. Roy, "DigSig: Run-time Authentication of Binaries at Kernel Level", *Usenix LISA*, 2004.
2. M.A. Williams, "Anti-Trojan and Trojan Detection with In-Kernel Digital Signature testing of Executables", NetXSecure NZ Ltd.. `http://www.netxsecure.net/downloads/sigexec.pdf`, 2002.
3. L. v. Doorn, G. Ballintijn, and W. A. Arbaugh, "Signed Executables for Linux", *Technical Report CS-TR-4256* University of Maryland, 2001.
4. H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", *RFC 2104*, 1997.
5. CERT, "Vulnerability Remediation Statistics", `http://www.cert.org/stats/vulnerability_remediation.html`, 2007.
6. R. Grimes, "Authenticode", Microsoft Technet, `http://www.microsoft.com/technet/archive/security/topics/secaps/authcode.mspx?mfr=true`.
7. Microsoft TechNet, "KnownDLLs", `http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/regentry/29908.mspx`.
8. "Microsoft Security Advisor Program: Microsoft Security Bulletin (MS99-006)", `http://www.microsoft.com/technet/security/bulletin/ms99-006.mspx`.
9. G.H. Kim and E.H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker", *ACM CCS*, 1993.
10. E.R. Arnold, "The Trouble With Tripwire", `http://www.securityfocus.com/infocus/1398`, 2001.
11. M. Slaviero, J. Kroon and M. S. Olivier, "Attacking Signed Binaries", *Proc. of the 5th Annual Information Security South Africa Conference (ISSA)*, 2005.
12. B. Acohido, "Security feature in Microsoft's new Windows could drive users nuts", USA Today, `http://www.usatoday.com/tech/products/2006-05-15-vista-security_x.htm?POE=TECISVA`, 2006.
13. M. Schmid, F. Hill, A.K. Ghosh, and J.T. Bloch, "Preventing the Execution of Unauthorized Win32 Applications", *DARPA Information Survivability Conf. & Exposition II (DISCEX)*, 2001.
14. S. Patil, A. Kashyap, G. Sivathanu, E. Zadok, "I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System", *USENIX LISA*, 2004.
15. Sufatrio, R. Yap and L. Zhong, "A Machine-Oriented Integrated Vulnerability Database for Automated Vulnerability Detection and Processing", *USENIX LISA*, 2004.
16. D.E. Eastlake and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, 2006.
17. X. Wang, H. Yu, "How to Break MD5 and Other Hash Functions", Eurocrypt '05, LNCS 3494, Springer, 2005.
18. "Sign Tool", `http://msdn2.microsoft.com/en-us/library/8s9b9yaz(vs.80).aspx`.
19. "Sigcheck", `http://www.microsoft.com/technet/sysinternals/Security/Sigcheck.mspx`.

A.8 pepino: peer-to-peer network inspector

PEPINO: PEer-to-Peer network INspectOr *

Donatien Grolaux, Boris Mejías, and Peter Van Roy
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

Abstract

PEPINO is a simple and effective peer-to-peer network inspector. It visualises not only meaningful pointers and connections between peers, but also the exchange of messages between them, providing a useful tool for debugging purposes. It can monitor running networks, simulate them and log them in order to reproduce interesting case scenarios. Failures can be explicitly introduced to study fault tolerant algorithms. The graphical representation of the network uses a physical model to attract or repel peers, allowing the user to study the system from different points of view. This demo aims to present the use of PEPINO in the development of a novel relaxed-ring topology for fault tolerant networks, where the representation of the ring based on predecessors may differ from the ring based on successors. We show how PEPINO is also useful for visualising other network topologies such as perfect ring or unstructured networks.

1. Introduction

Developing peer-to-peer systems requires the ability of visualising the network in terms of the designed algorithms. The more dynamic the case-studies become, the harder is to keep track of all interaction between peers. The obvious solution is to use a software to graphically represent the network, and thus, nearly every developer group creates its own network viewer to study their algorithms. Then, why do we present yet another viewer? The reason to do it is because existing tools are so ad-hoc to each network that studying a different network topologies in such tools did not allowed us to visualise our concrete issues.

We developed PEPINO, a PEer-to-Peer network INspectOr that adapts its graphical representation to several network topology such as ring, relaxed-ring, unstructured networks or even client-server. The graphical representation uses a physical model to attract and repel connected nodes depending on the weight of each kind of connection. This is how the viewer is able to dynamically adapt itself to different network architectures, without making any previous assumption on the topology.

Another crucial information for debugging is the exchange of messages between peers. PEPINO displays simultaneously with the representation of the network, the communication between peers and the events triggered by each one of them. Messages and events are annotated with different categories, allowing filters in order to get more meaningful information.

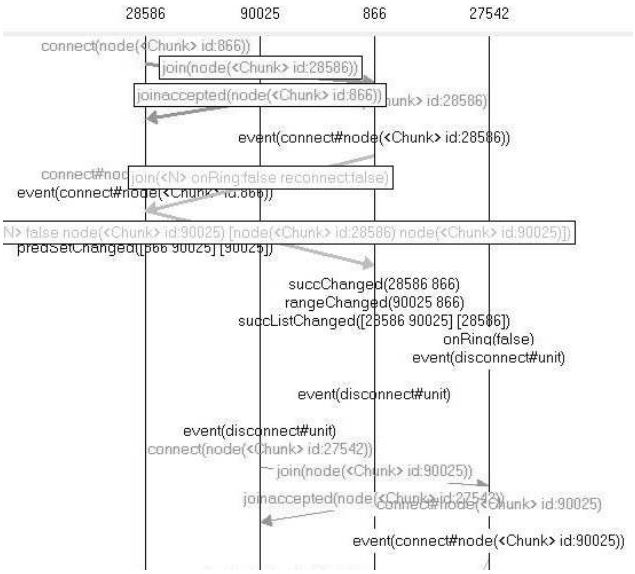


Figure 1. Message exchange between peers

2. PEPINO and P2PS

We use PEPINO in the development of P2PS*v*3 [2], a Chord-like platform to develop fault-tolerant peer-to-peer applications. P2PS uses a network topology based on a relaxed-ring where only the predecessor links form a perfect ring, guaranteeing a correct distribution the responsi-

*This research is mainly funded by EVERGROW (contract number:001935) and SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

bility of the keys. Successors follow a different invariant allowing the presence of branches in the ring. The topology and the algorithms of P2PS provide a network that can survive efficiently to failures of nodes and also to broken links (inaccurate failure detection), which are often ignored.

P2PS is implemented using a software architecture based on tiers, where the lowest tier implements point-to-point communication. The relaxed-ring maintenance is another layer placed upper in the architecture. As we previously mentioned, PEPINO can display messages between peers in different categories. In the particular case of P2PS, every tier is represented by a category. Figure 1 depicts how messages are represented, and how one category is highlighted. The figure is shown in grey scale, but it is possible to distinguish that every category has its own colour. Every category can be enabled or disabled in order to avoid unnecessary verbosity.

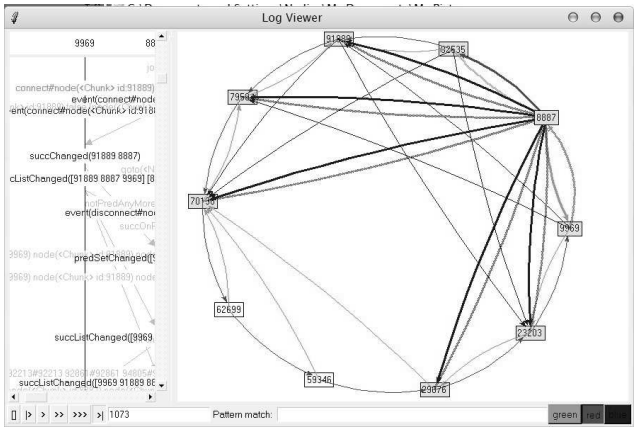


Figure 2. Fingers on a ring network

During the demonstration, different networks will be presented in order to observe their dynamic behaviour. Figure 2 is a screenshot of PEPINO visualising a network using ring topology. On the left side of the screenshot is possible to observe the frame with messages between peers. On the right side, in the graphical representation of the ring, the finger of a particular peer are highlighted. On the bottom right corner, there is a set of buttons allowing the election of the strongest connector between peers for the graphical representation. The underlay physical model will adapt the parameter for attraction or repulsion of nodes according to these settings. Like this, it is possible to observe the network from different points of view. For instance, putting the focus on the predecessors or successor links.

To check how the network reacts to network failures, it is possible to explicitly inject temporary or permanent failures on nodes. Failures can also be injected in the communication channel between peers, which is one way to study inaccuracy of failure detection.

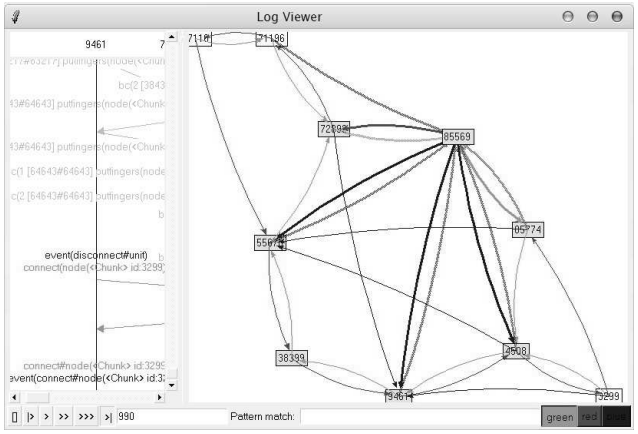


Figure 3. Relaxed-ring topology

Figure 3 depicts the visualisation of the relaxed-ring topology of P2PS, our main focus of interest for the demonstration. The information given by PEPINO helps the developer to understand why branches are created, and how the network recovers from crashed peers and broken links. P2PS implements different algorithms for the election of fingers such as Chord [5], Tango [1] and DKS [3], allowing the comparison between them.

PEPINO is also useful for bug reports. The history of a visualised network can be saved in a log file to be sent to developers. The log can be visualised at different speeds. In figures 2 and 3, a set of arrows can be seen at the bottom left corner. The speed of visualisation can be tuned with those buttons. It is also possible to run the visualisation until a particular event identified by a number, or matching a pattern.

PEPINO is implemented with Mozart [4], and it can be run on Linux, MacOSX and other Unix systems. It also runs on Windows 98/NT/XP.

References

[1] B. Carton and V. Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 1060–1067. Springer, 2004.

[2] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://gforge.info.ucl.ac.be/projects/p2ps*, 2007.

[3] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[4] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2007.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.9 Visualizing Transactional Algorithms for DHTs

Visualizing Transactional Algorithms for DHTs

Boris Mejías
Université catholique de Louvain
boris.mejias@uclouvain.be

Mikael Högqvist
Zuse Institute Berlin
hoegqvist@zib.de

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

1. Introduction

Distributed Hash Tables (DHT) provide interesting properties for storing and retrieving data in decentralized systems. They are usually built on top of Structured Overlay Networks (SON) which has self-organizing and fault-tolerant properties. A DHT offers a simple interface to store and lookup elements associated with a key. The operations are basically $put(key, value)$ and $get(key)$. Every peer is responsible for a set of keys and if a peer fails, another peer takes over its responsibility. But what happens with the data of the crashed peer? Either the data must be re-inserted into the system or it can be replicated and recovered on the new responsible node. A replication mechanism must guarantee that the recovered replicated value is the same as the last value stored before the failure.

Data replication becomes more complex when the application running on top of the peer-to-peer network requires the update of several values stored on the DHT at the same time. This is typically done as a *transaction* involving keys belonging to different sets, and hence, involving different peers. How are the different peers coordinated in order to decide if the whole transaction must *commit* or *abort*? How do the replicas of these peers get the last valid data?

The two-phase commit protocol (2PC) is one of the most popular choices for implementing distributed transactions, being used since the 1980s. Unfortunately, its use on peer-to-peer networks is very inefficient because it relies on the survival of the transaction manager, as explained further in section 2. A three-phase commit protocol (3PC) has been designed in order to overcome the limitation of 2PC. However, 3PC introduces an extra round-trip which results in higher latency and increased message load. We advocate the use of an algorithm based on Paxos consensus [4, 2]. This algorithm is especially adapted for the requirements of a DHT and can survive a crash of the coordinator during a transaction. Compared to 3PC, it reduces latency and overall message load by requiring less message round-trips.

Demonstrator

We implement two-phase commit and the Paxos consensus-based algorithm on top of a Chord-like structured overlay network [5], extending the PEPINO network inspector [3] for visualization. By introducing arbitrary failures, the demonstrator shows why two-phase commit does not work on peer-to-peer networks. Then, the robustness of Paxos consensus is tested by injecting failures on a certain amount of transaction managers and participants, showing the failure recovery mechanism of this protocol.

2. Two-phase commit

The pseudo-code below implements a swap operation within a transaction. The objective is that the instructions from the beginning of the transaction (BOT) until its end (EOT) are executed atomically to avoid race conditions with other concurrent operations. The values of $item\_i$ and $item\_j$ are stored on different peers. The operators put and get are replaced by $read$ and $write$ in order to differentiate a regular DHT from a transactional DHT.

```
BOT
    x = read(item_i);
    y = read(item_j);
    write(item_j, x);
    write(item_i, y);
EOT
```

In order to guarantee atomic commit of a transaction on a decentralized storage, two-phase commit uses a *validation* phase and a *write* phase, coordinated by a *transaction manager* (TM). All peers responsible for the items involved in the transaction, as well as their replicas, become *transaction participants* (TP). Initially, the TM sends a request to every TP to *prepare* the transaction. If the item is available, the TP will lock it and acknowledge the *prepare* request. Otherwise, it will reply *abort*. The *write* phase follows *validation* once the replies are collected by the TM. If none of the participants voted *abort*, then the decision will be *commit*. When the participants receive the commit message from the TM, they will make the update permanent and release the lock on the item. An abort message will discard any update and release the item locks.

The problem with the 2PC protocol is that relies too much on the survival of the transaction manager. If the TM fails during the validation phase, it will block all the TPs that acknowledged the prepare message. A very reliable TM is required for this protocol, but it cannot be guaranteed on peer-to-peer networks.

3. Paxos Consensus Algorithm

The 3PC protocol avoids the blocking problem of 2PC at the cost of an extra message round-trip. This solution might be acceptable for cluster-based applications but not for peer-to-peer networks, where it is better to have less rounds with more messages than adding extra rounds to the protocol. This problem lead to the recent introduction of [4] based on Paxos consensus [2].

The idea is to add replicated transaction managers (rTM) that can take over the responsibility of the TM in case of failure. The other advantage is that decisions can be made considering a majority of the participants reaching consensus, and therefore, not all participants needs to be alive or reachable to commit the transaction. This means that as long as the majority of participants survives, the algorithm terminates even in presence of failures of the TM and TPs, without blocking the involved items.

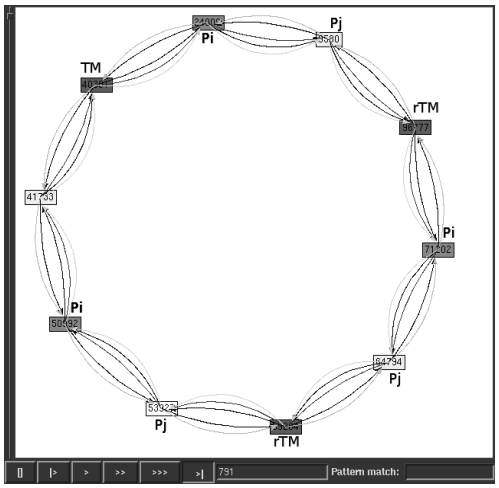


Figure 1. Network during a transaction with replicated manager and participants.

Figure 1 depicts a network visualized by the demonstrator. Different colours are assigned to TMs and TPs depending on the item they are responsible for. The labels in the figure are not in the simulator, but where added here for clarity. The TMs and TPs in the protocol are replicated using symmetric replication as described in [1]. Figure 2 shows the initial effect of introducing an arbitrary failure on the transaction manager, breaking the connection of the ring. The demonstrator continues with the recovery of the ring, and the election of a new TM from the rTMs.

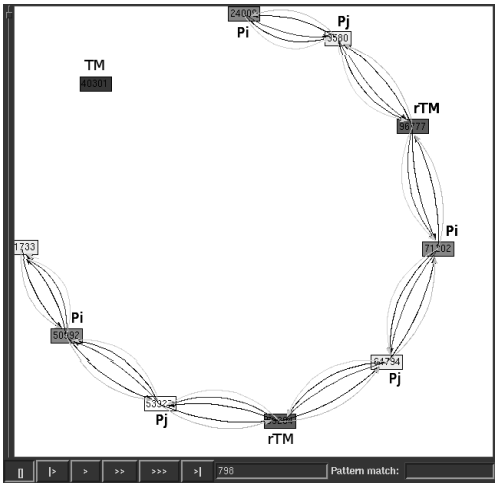


Figure 2. Failure of the transaction manager

4. Summary

The focus of this demonstrator is on the study of algorithms for implementing transactions on peer-to-peer networks. Their visualization contributes to the analysis and test of the protocols, verifying their tolerance to failures. In particular, we show a DHT running two-phase commit and the Paxos consensus algorithm.

5. Acknowledgements

The authors would like to thank Monika Moser and Seif Haridi for their help on the understanding of the Paxos consensus algorithm. This work is mainly funded by project SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[2] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[3] D. Grolaux, B. Mejías, and P. Van Roy. PEPINO: PEer-to-Peer network INspectOr. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007.

[4] M. Moser and S. Haridi. Atomic commitment in transactional DHTs. In *Proceedings of the CoreGRID Symposium*, 2007.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.10 Partitioning and Merging the Ring

Partitioning and Merging the Ring

Boris Mejías, and Peter Van Roy
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

1. Introduction

It is claimed by many structured overlay networks (SON) that they survive network partitions. Ideally, if two sets of nodes get isolated from each other due to a network partition, it is expected that they would form two separated SONs that might get merged when the connection between both sets is recovered. This principle should apply if the network gets partitioned into more groups. We study in particular SONs built using Chord-like ring topology [5]. According to [2], the ring topology is one of the most resilient to failures, and it is competitive with any other SON with respect to proximity. This make ring-based system an interesting and representative case study.

Even when SONs survive network partition, very little has been done with respect to efficient merging algorithms. In general, it is expected that two rings can merge by taking peers from one of the ring, and one by one let them join the second ring. This is of course effective, but not efficient. An interesting algorithm presented by Shafaat [4] introduces a gossip-based ring unification algorithm. It is claimed that this algorithm is efficient and resilient to churn during the merging process.

This demonstrator is focused on the study of the tolerance of ring-based systems to network partitions and the behaviour of merging algorithms. A comparison between the default unification and the gossip-based algorithm is presented using a graphical application.

2. Demonstrator

We implement a system called P2PS [3] that uses a relaxed-ring topology. By relaxing the ring, P2PS allows peers with connectivity problems to join the network in branches attached to the ring. Just like a Chord ring, P2PS survives a network partition resulting on two or more rings depending on the nature partition. To test the behaviour of P2PS, we use PEPINO [1], a graphical peer-to-peer network inspector.

We extends PEPINO in order to arbitrarily assign different domains to each peer on the network. To study a network partition, we introduce a disconnection between different domains. If the amount of resilient pointers is sufficient, two or more rings will be created, as it is shown in Figure 1.

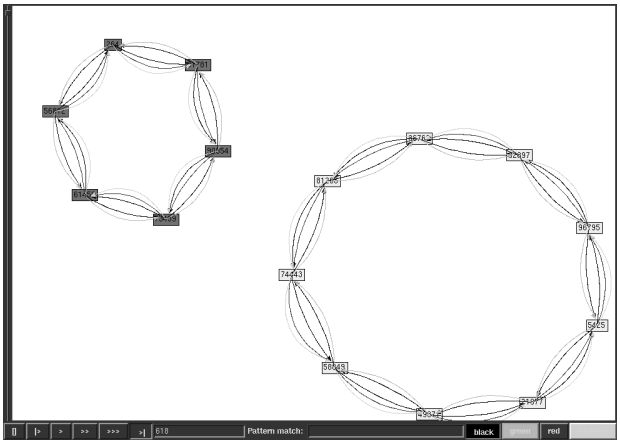


Figure 1. Network partitioned into two rings.

After the network partition has occurred, we reestablish the connection between different artificial domains to observe the merging of the rings in two different ways. The first one, as described in the introduction, corresponds to the default behaviour, where nodes from one ring join the second ring one by one. Figure 2 depicts a similar and more or less chaotic reorganization of a ring where peers join the network at high churn.

We compare this behaviour with the gossip-based reunification. In general, the algorithm consists on two steps. When a peer p from ring 1 detects a peer q from ring 2, it requests a *mlookup* to identify its possible successor on the other ring. Peer q does the equivalent *mlookup*. Once p and q have found their candidates, they send *trymerge* to their new predecessor and successor. Once the *trymerge* succeeds, new *mlookup*

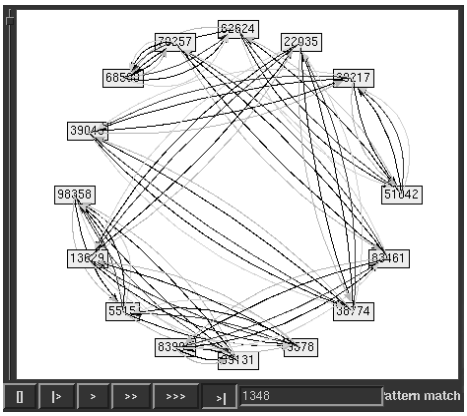


Figure 2. Peers joining a ring one by one.

messages are triggered by the new involved peers in order to continue with the unification. In order to make this mechanism more efficient, random *mlookup* messages are trigger in parallel. This produces that peers from both rings can find each other faster. More details of the algorithm can be found on the reference.

Considering that the algorithm is designed to work on any ring-based system, we implement it on P2PS to improve the recovery of the relaxed-ring after network partitions. The behaviour of the algorithm can be observed with the demonstrator in a graphical manner, and also at the level of messages exchanged. PEPINO offers a log viewer to analyze the messages sent between peers.

3. Summary

This demonstrator offers a graphical way to study how tolerant a system is with respect to network partitions. We also analyze how efficient are the proposed merging algorithms. The study is focused on ring-based networks, with two different approaches for unifying rings. A very simple one moving one peer at the time from one ring to the other, and a gossip-based algorithm that can unify rings even in presence of churn during the merging.

4. Acknowledgements

The authors would like to thank Tallat Shafaat and Seif Haridi for their help on the understanding of the gossip-based ring unification algorithm. This work is mainly funded by project SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] D. Grolaux, B. Mejías, and P. Van Roy. PEPINO: PEer-to-Peer network INspectOr. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007.

[2] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.

[3] B. Mejías and P. Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.

[4] T. M. Shafaat, A. Ghodsi, and S. Haridi. Handling network partitions and mergers in structured overlay networks. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007. To appear.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.11 Transactions for Distributed Wikis on Structured Overlays

Transactions for Distributed Wikis on Structured Overlays*

Stefan Plantikow, Alexander Reinefeld, and Florian Schintke

Zuse Institute Berlin

Abstract. We present a transaction processing scheme for structured overlay networks and use it to develop a distributed Wiki application based on a relational data model. The Wiki supports rich metadata and additional indexes for navigation purposes.
Ensuring consistency and durability requires handling of node failures. We mask such failures by providing high availability of nodes by constructing the overlay from replicated state machines *(cell model)*. Atomicity is realized using two phase commit with additional support for failure detection and restoration of the transaction manager. The developed transaction processing scheme provides the application with a mixture of pessimistic, hybrid optimistic and multiversioning concurrency control techniques to minimize the impact of replication on latency and optimize for read operations. We present pseudocode of the relevant Wiki functions and evaluate the different concurrency control techniques in terms of message complexity.

Keywords. Distributed transactions, content management systems, structured overlay networks, consistency, concurrency control.

1 Introduction

Structured overlay networks (SONs) provide a scalable and efficient means for storing and retrieving data in distributed environments without central control. Unfortunately, in their most basic implementation, SONs do not offer any guarantees on the ordering of concurrently executed operations.

Transaction processing provides concurrently executing clients with a single, consistent view of a shared database. This is done by bundling client operations in a transaction and executing them as if there was a global, serial transaction execution order. Enabling structured overlays to provide transaction processing support is a sensible next step for building *consistent* decentralized, self-managing storage services.

We propose a transactional system for an Internet-distributed content management system built on a structured overlay. Our emphasis is on supporting

* This work was partially supported by the EU projects SELFMAN and CoreGRID.

transactions in dynamic decentralized systems where nodes may fail at a relatively high rate. The chosen approach provides clients with different concurrency control options to minimize latency.

The article is structured as follows: Section 2 describes a general model for distributed transaction processing in SONs. The main problem addressed is masking the unreliability of nodes. Section 3 presents our transaction processing scheme focusing on concurrency control. This scheme is extended to the relational model and exemplified using the distributed Wiki in Section 4. Finally, in Section 5, we evaluate the different proposed transaction processing techniques in terms of message complexity.

2 Transactions on Structured Overlays

Transaction processing guarantees the four ACID properties: *Atomicity* (either all or no data operations are executed), *consistency* (transaction processing never corrupts the database state), *isolation* (data operations of concurrently executing transactions do not interfere with each other), *durability* (results of successful transactions survive system crashes). Isolation and consistency together are called *concurrency control*, while *database recovery* refers to atomicity and durability.

Page model. We only consider transactions in the *page model* [1]: The database is a set of uniquely addressable, single objects. Valid elementary operations are reading and writing of objects, and transaction abort and commit. The model does not support predicate locking. Therefore, phantoms can occur and consistent aggregation queries are not supported. The page model can naturally be applied to SONs. Objects are stored under their identifier using the overlay's policy for data placement.

2.1 Distributed Transaction Processing

Distributed transaction processing guarantees the ACID properties in scenarios where clients access multiple databases or different parts of the same database located on different nodes. Access to local databases is controlled by *resource manager (RM)* processes at each participating node. Additionally, for each active transaction, one node takes the role of the *transaction manager (TM)*. The TM coordinates with the involved RMs to execute a transaction on behalf of the client. The TM also plays an important role during the execution of the distributed atomic commit protocol.

Distributed transaction processing in a SON requires distribution of resource and transaction management. The initiating peer can act as TM. For resource management, it is necessary to minimize the communication overhead between RM and storing node. Therefore, in the following, we assume that each peer of the overlay performs resource management for all objects in its keyspace partition.

2.2 The Cell Model for Masking Churn

Distributing the resource management over all peers puts tight restrictions on messages delivered under transaction control. Such messages may only be delivered to nodes that are currently responsible for the data. This property is known as *lookup consistency*. Without lookup consistency, a node might erroneously grant a lock on a data item or deliver outdated data. It is an open question how lookup consistency can be guaranteed efficienty in the presence of frequent and unexpected node failures *(churn)*. Some authors (e.g. [2]) have suggested protocols that ensure consistent lookup if properly executed by *all* joining and leaving nodes. Yet large-scale overlays are subject to considerable amounts of churn [3] and therefore correct transaction processing requires masking it.

Cell model. Instead of constructing the overlay network using single nodes, we propose to build the overlay out of *cells*. Each cell is a dynamically sized group of physical nodes [4] that constitute a *replicated state machine (RSM, [5])*. Cells utilize the chosen RSM algorithm to provide replicated, atomic operations and high availability. This can be exploited to

- mask churn and therefore guarantee lookup consistency,
- provide stable storage for transactional durability,
- ensure data consistency using atomic operations,
- minimize overhead for routing to replicas (cell nodes form a clique).

For the underlying nodes, we assume the *crash-stop* failure model. This model is common for SONs because it is usually unknown wether a disconnected node will rejoin again later. We do not cover the distribution of physical nodes on cells, nor do we consider Byzantine failures. We assume that cells never fail unexpectedly and always execute the overlay algorithm orderly. If too many cell nodes fail, the cell destroys itself by executing the overlay's leave protocol. The data items are re-distributed among neighboring cells. For simplification, we also assume that the keyspace partition associated to each cell does not change during transaction execution.

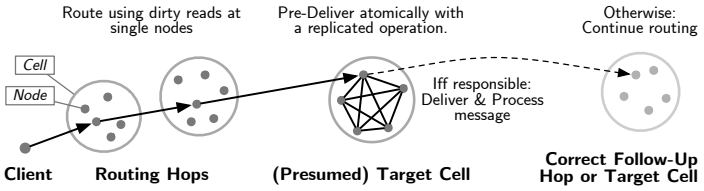


Fig. 1. Cell routing using dirty reads.

Cell routing. The execution of replicated operations within the cells comes at a considerable cost: RSMs are implemented using some form of atomic broadcast which in turn depends on an implementation of a consensus protocol. Yet, modern consensus algorithms like *Fast Paxos* [6] require at least $N(\lfloor 2N/3 \rfloor + 1)$ messages for consensus between N nodes. While this cost is hardly avoidable for consistent replication, it is as well unacceptable for regular message routing. Hence we propose to use dirty reads (i.e. to read the state of one arbitrary node). When the state of a node and its cell are temporarily out of sync, routing errors may occur. To handle this, the presumed target cell will pre-deliver the message using a replicated operation (Fig. 1). Pre-delivery first checks, wether the presumed target cell is currently responsible for the message. If that is the case, the message is delivered and processed regularly. Otherwise, message routing is restarted from the node that initiated pre-delivery.

Replicated operations will only be executed at a node after all of its predecessor operations have been finished. Therefore, at pre-delivery time, the presumed target cell either actually is responsible for the message or a previously executed replicated operation has changed the cell's routing table consistently such that the *correct* follow-up routing hop or target cell for the message is known. A messages reaches its destination under the assumption that cell routing table changes are sufficiently rare, and intermediate hops do not fail.

3 Concurrency Control and Atomic Commit in SONs

In this section, we present appropriate concurrency control and atomic commit techniques for overlays based on the cell model.

Atomic Operations. Using RSMs by definition [5] allows the execution of atomic and totally ordered operations. This already suffices to implement transaction processing, e.g. by using *pessimistic two phase locking (2PL)* and an additional distributed atomic commit protocol. However, each replicated operation is expensive, thus any efficient transaction processing scheme for cell-structured overlays should aim at minimizing the number of replicated operations.

Optimistic concurrency control (OCC). OCC executes transactions against a *local* working copy (working phase). This copy is validated just before the transaction is committed (validation phase). The transaction is aborted if conflicts are detected during validation. As every node has (a possibly temporarily deviating) local copy of its cell's shared state, OCC is a prime candidate for reducing the number of replicated operations by executing the transaction against single nodes of each involved cell.

3.1 Hybrid Optimistic Concurrency Control

Plain OCC has the drawback that long-running transactions using objects which are frequently accessed by short-running transactions may suffer starvation due

to consecutive validation failures. This is addressed by *hybrid optimistic concurrency control (HOCC, [7])* under the assumption of *access invariance*, i.e. repeated executions of the same transaction have identical read and write sets.

HOCC works by executing *strong two phase locking (SS2PL)* for the transaction's read and write sets at the beginning of the validation phase. In case of a validation failure, locks are kept and the transaction logic is re-executed. Now, access invariance ensures that this second execution cannot fail because all necessary locks are already held by the transaction. However, it is required that optimistically read values do not influence the result of the re-execution phase. Otherwise, consistency may violated.

The use of SS2PL adds the benefit that no distributed deadlock detection is necessary if a global validation order between transactions is established. A possible technique for this has been described by Agrawal et. al [8]: Each cell v maintains a strictly monotonic increasing timestamp t_v for the largest, validated transaction. Before starting the validation, the transaction manager suggests a validation time stamp $t > t_v$ to all involved cells. After each such cell has acknowledged that $t > t_v$ and updated t_v to t, the validation phase is started. Otherwise the algorithm is repeated. Gruber [9] optimized this approach by including the current t_v in every control message.

3.2 Distributed Atomic Commit

Distributed atomic commit (DBAC) requires consensus between all transaction participants on the transaction's termination state (committed or aborted). If DBAC is not guaranteed, the ACID properties are violated.

We propose a blocking DBAC protocol that uses cells to treat TM failures by replicating transaction termination state. Every transaction is associated with a unique identifier (TXID). The overlay cell corresponding to that TXID is used to store a *commit record* holding the termination state and the address of the TM node (an arbitary, single node of the TXID cell). If no failures occur, regular *two-phase atomic commit (2PC)* is executed. Additionally, after all prepared-messages have been received and before the final commit messages are sent, the TM first writes the commit record. If the record is already set to abort, the TM aborts the transaction. If RMs suspect a TM failure, they read the record to either determine the termination state or initiate transaction abort. Optionally, RMs can restore the TM by selecting a new node and updating the record appropriately. Other RMs will notice this when they reread the modified record.

3.3 Read-only Transactions

In many application scenarios simple read-only transactions are much more common than update transactions. Therefore we optimize and extend our transaction processing scheme for read-only transactions by applying techniques similar to *read-only multiversioning (ROMV, [10])*.

All data items are versioned using unique timestamps generated from each node's loosely synchronized clock and globally unique identifier. Additionally, we maintain a *current version* for each data item. This version is accessed and locked exclusively by HOCC transactions as described above and implicitly associated with the cell's maximum validation timestamp t_v. The current version decouples ROMV and HOCC.

Our approach moves newly created versions to the future such that they never interfere with read operations from ongoing read-only transactions. This avoids the cost associated with distributed atomic commit for read-only transactions but necessitates it to execute reads as replicated operations. Read-only transactions are associated with their start time. Every read operation is executed as a replicated operation using the standard multiversioning rule [11]: The result is the oldest version which is younger than the transaction start time. If this version is the current version, the maximum validation timestamp t_v will be updated. This may block the read operation until a currently running validation is finished. Update transactions create new versions of all written objects using $t > t_v$ during atomic commit.

4 Algorithms for a Distributed Wiki

In this section, we describe the basic algorithms of a distributed content management system built on a structured overlay with transaction support.

4.1 Mapping the Relational Model

So far we only considered uniquely addressable, uniform objects. In practice, many applications use more complex, relational data structures. This raises the question of how multiple relations with possibly multiple attributes can be stored in a single structured overlay. To address this, first, we assume that the overlay supports range queries [12, 13] over a finite number of index dimensions.

Storing multiple attributes requires mapping them on index dimensions. As the number of available dimensions is limited, it is necessary to partition the attributes into disjoint groups and map these groups instead. The partition must be chosen in such a way that fast primary-key based access is still possible. Depending on their group membership, attributes are either primary, index, or non-indexed data attributes. Multiple relations can be modeled by prefixing primary keys with a unique relation identifier.

4.2 Notation

Table 1 contains an overview of the pseudocode syntax from [14]. Relations are represented as sets of tuples and written in CAPITALS. Relation tuples are addressed by using values for the primary attributes in the fixed order given by the relation. For reasons of readability, tuple components are addressed using unique labels (Such labels can easily be converted to positional indexes). Range queries are expressed using labels and marked with a "?".

Table 1. Pseudocode notation

Syntax	Description
Procedure Proc $(arg_1, arg_2, \ldots, arg_n)$	Procedure declaration
Function Fun $(arg_1, arg_2, \ldots, arg_n)$	Function declaration
begin...**commit (abort) transaction**	Transaction boundaries
ADDRESS$_{"ZIB"}$	Read tuple from relation
ADDRESS$_{"ZIB"} \leftarrow$ ("*Takustr.* 7", "*Berlin*")	Write tuple to relation
$\Pi_{attr_1, \ldots, attr_n}(M) = \{\pi_{attr_1, \ldots, attr_n}(t) \mid t \in M\}$	Projection
$\forall t \in$ tuple set : RELATION $\overset{+}{\leftarrow} t$ bzw. $\overset{-}{\leftarrow} t$	Bulk insert and delete
$\text{DHT}^{?}_{key_1="a",\ key_2}$ *or* $\text{DHT}^{?}_{key_1="a",\ key_2=*}$	Range query with wildcard

4.3 Wiki

A *Wiki* is a content management system that embraces the principle of minimizing access barriers for non-expert users. Wikis like `www.wikipedia.org` comprise millions of pages written in a simplified, human-readable markup syntax. Each page has a unique name which is used for hyperlinking to other Wiki pages. All pages can be read and edited by any user, which may result in many concurrent modification requests for hotspot pages. This makes Wikis a perfect test-case for our distributed transaction algorithm.

Modern Wikis provide a host of additional features, particularly to simplify navigation. In this paper we exemplarily consider backlinks (a list of all the other pages linking to a page) and recent changes (a list of recent modifications of all Wiki pages). We model our Wiki using the following two relations:

Relation	Primary attributes	Index attributes	Data attributes
CONTENT	*pageName*	*ctime* (change time)	*content*
BACKLINKS	*referencing* (page), *referenced* (page)	-	-

Wiki operations use transactions to maintain global consistency invariants:

- CONTENT always contains the current content for all pages,
- BACKLINKS contains proper backlinks for all pages contained in CONTENT,
- users cannot modify pages whose content they have never seen (explained below).

The function WikiRead (Alg. 4.1) delivers the content of a page and all backlinks pointing to it. This requires a single read for the content and a range query to obtain the backlinks. Both operations can be executed in parallel.

The function WikiWrite (Alg. 4.2) is more complex because conflicting writes by multiple users must be resolved. This can be done by serializing the write requests using locks or request queues. If conflicts are detected during (atomic) writes by comparing last read and current content, the write operation is aborted. Users may then manually merge their changes and retry. This approach is similar to the compare-and-swap instructions used in modern microprocessors and to the concurrency control in version control systems.[1] We realize the compare-and-swap in WikiWrite by using transactions for our distributed Wiki. First, we precompute which backlinks should be inserted and deleted. Then, we compare the current and old page content and abort if they differ. Otherwise all updates are performed by writing the new page content and modifying BACKLINKS. The update operations again can be performed in parallel.

Algorithm 4.1 WikiRead: Read page content

```
1: function WikiRead(pageName)
2:     begin transaction read-only
3:         content ← π_content(CONTENT_pageName)
4:         backlinks ← Π_referenced(BACKLINKS^?_{referencing=pageName, referenced})
5:     commit transaction
6:     return content, backlinks
7: end function
```

Algorithm 4.2 WikiWrite: Write new page content and update backlinks

```
1: procedure WikiWrite(pageName, content_old, content_new)
2:     refs_old ← Refs(content_old)
3:     refs_new ← Refs(content_new)
4:     refs_del ← refs_old \ refs_new          — precalculation
5:     refs_add ← refs_new \ refs_old
6:     txStartTime ← CurrentTimeUTC()
7:     begin transaction
8:         if π_content(CONTENT_pageName) = content_old then
9:             CONTENT_pageName = (txStartTime, content_new)
10:            ∀t ∈ {(ref, pageName) | ref ∈ refs_add} : BACKLINKS ←^+ t
11:            ∀t ∈ {(ref, pageName) | ref ∈ refs_del} : BACKLINKS ←^− t
12:        else
13:            abort transaction
14:        end if
15:    commit transaction
16: end procedure
```

[1] Most version control systems provide heuristics (e.g. merging of different versions) for automatic conflict resolution that could be used for the Wiki as well.

Algorithm 4.3 SetPageMetadata: Write page metadata attributes

Require: *changeEnv* environment describing changes to be made

```
1: procedure SetPageMetadata (pageName, content_old, changeEnv)
2:     begin transaction
3:         if π_content(CONTENT_pageName) = content_old then
4:             ∀(anAttrName ⇐ anAttrValue) ∈ changeEnv :
5:                 METADATA_pageName, anAttrName ← anAttrValue
6:         else
7:             abort transaction
8:         end if
9:     commit transaction
10: end procedure
```

The list of recently changed pages can be generated by issuing a simple range query inside a transaction and sorting the results appropriately.[2]

4.4 Wiki with Metadata

Often it is necessary to store additional metadata with each page (e.g. page author, category). To support this, we add a third relation METADATA with primary key attributes *pageName* and *attrName* and data attribute *attrValue*. Alternatively we could also add metadata attributes to CONTENT. But this would not be scalable as current overlays only provide a limited number of index dimensions.

Modifying page metadata (Alg. 4.3) requires verifying that the page has not been changed by some other transaction. Otherwise new metadata could be associated wrongly to a page (This is similar to storing wrong backlinks). For reading page metadata, a simple range query suffices [14].

5 Evaluation

The presented algorithms for ensuring consistency mainly require the atomicity property while only few restrictions are placed on the serial execution order of operations. Thus in theory, a high degree of concurrency is possible. This is especially interesting for range queries like RecentChanges which can utilize the overlay's capabilities to multicast to many nodes in parallel.

Table 2 shows the communication overhead of various concurrency control schemes. We compare the different schemes using an example transaction that consists of k serial steps. Each step executes data operations in parallel on N cells (one operation per cell).

For every scheme, we distinguish the number and type of operations necessary to carry out the transaction: U is a simple unreplicated operation, R is a replicated operation, and L is a lookup (routing) operation. The cost is split into one-time (initial and DBAC) overhead, the cost per step, and the total cost.

[2] The complete range query is: $\{\text{CONTENT}^{?}_{pageName=*,\ ctime=*}\}^{\overleftarrow{ctime}}_{\#<resultLimit}$

Table 2. Comparison of concurrency control methods

Transaction type	**One-time overhead for N cells**	**Ops per step on N cells in parallel**	**Total for k serial steps**
(1) Atomic Write	$1\,L$	$1\,R$	$1\,L + 1\,R$, because $k, N = 1$
(2) Read-Only Trans.	$N\,L$	$N\,R$	$N\,L + kN\,R$
(3) Pess. 2PL + 2PC	$N\,L + 2N\,R$	$N\,R$	$N\,L + (k+1)N\,R$
(4) Hyb. Opt. + 2PC	$N\,L + 2N\,R$	$N\,U$	$N\,L + (k-1)N\,U + 2N\,R$
(5) Hyb. Opt. + 2PC + Validation Error	$N\,L + 3N\,R$	$2N\,U$	$N\,L + (2k-2)N\,U + 3N\,R$

(2) to (4) use the 2PC variant described in 3.2. For our evaluation, we assume that no failures occur during the commit.

Totals include DBAC costs and take the possible combined sending of messages into account (e.g. combining last write operation with validate and prepare). The evaluated concurrency control schemes are:

(1) a simple, replicated operation on a single cell,
(2) a read-only multiversioning transaction (Sec. 3.3),
(3) a pessimistic 2PL transaction,
(4) a HOCC (Sec. 3.1) transaction without validation failure, and
(5) a HOCC transaction with validation failure and re-execution of transaction logic.

HOCC reduces the number of necessary replicated operations for $k > 1$. For $k = 1$ and a transaction on a single cell, ACID is already provided by using a RSM and no DBAC is necessary. For $k = 1$ and a transaction over multiple cells, HOCC degenerates into 2PL: the data operations on the different cells are combined with validate-and-prepare messages and executed using single replicated operations.

Read-only transactions use more replicated operations but save the DBAC costs of HOCC. This makes them well-suited for quick, parallel reads. But long running read transactions might be better off using HOCC if the performance gained by optimism outweights DBAC overhead and validation failure chance.

Using cells yields an additional benefit. If replication was performed above the overlay layer, additional routing costs of $(r-1)N$ lookup messages would be necessary (r is the number of replicas).

6 Related Work

Mesaros et. al describe a transaction processing scheme for overlays based on 2PL [15]. Lock conflicts are resolved by giving higher priority to older transactions and forcing the loosing transaction into the 2PL shrinking phase. Transactions are executed by forming a dynamic multicast group consisting of all

participating nodes. The article does not address issues of lookup consistency and replication.

OceanStore [16] uses a two-tier approach for multiversioning-based replication. On the first layer, a small set of replicas forms a primary ring. On the second layer, additional replicas cache object versions. Replicas are located using the Tapestry overlay network. Primary ring replicas use a Byzantine agreement protocol to serially execute atomic operations.

Etna [17] is a system for executing atomic read and write operations in a Chord-like overlay network. Operations are serialized using a primary copy and replicated over k sucessors using a consensus algorithm.

Both articles do not describe how full transaction processing can be built on top of atomic operations. For OceanStore, multiversioning [11] is proposed [16]. The inherent cost of replicated transaction execution is handled using the caching tier. However, this comes at the price of reduced consistency.

As an alternative to our solution for atomic commitment, Moser et al. [18] describe a non-blocking approach based on Paxos commit. Their solution treats the set of all replicas of all accessed items as a whole and fixes this set at commit time. They suggest the use of symmetric replication [19] to achieve availability. Instead of using RSMs inside cells, encoding schemes like Reed-Solomon codes could be used, as proposed by Litwin et al. [20] to ensure proper availability.

7 Summary

We presented a transaction processing scheme suitable for a distributed Wiki application on a structured overlay network. While previous work on overlay transactions has not addressed node unreliability, we identified this as a key requirement for consistency and proposed the cell model as a possible solution.

The developed transaction processing scheme provides applications with a mixture of concurrency control techniques to minimize the required communication effort. We showed core algorithms for the Wiki that utilize overlay transaction handling support and evaluated different concurrency control techniques in terms of message complexity.

References

1. Gray, J.: The transaction concept: Virtues and limitations. In: Proceedings of the 7th Intl. Conf. on Very Large Databases. (1981) 144–154
2. Ghodsi, A.: Distributed k-Ary System: Algorithms for Distributed Hash Tables. PhD thesis, KTH Stockholm (2006)
3. Li, J., Stribling, J., Gil, T.M., Morris, R., Kaashoek, M.F..: Comparing the performance of distributed hash tables under churn. In: IPTPS '04. (February 2004)
4. Schiper, A.: Dynamic group communication. Distributed Computing **18**(5) (2006) 359–374
5. Schneider, F.B.: The state machine approach: A tutorial. Technical Report TR 86-800, Dept. of Comp. Sci. , Cornell University (December 1986)

6. Lamport, L.: Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research (January 2006)
7. Thomasian, A.: Distributed optimistic concurrency control methods for high-performance transaction processing. IEEE Transactions on Knowledge and Data Engineering **10**(1) (January/February 1998) 173–189
8. Agrawal, D., Bernstein, A.J., Gupta, P., Sengupta, S.: Distributed optimistic concurrency control with reduced rollback. Distributed Computing (2) (1987) 45–59
9. Gruber, R.E.: Optimistic Concurrency Control for Nested Distributed Transactions. PhD thesis, Massachusetts Institute of Technology (June 1989)
10. Mohan, C., Pirahesh, H., Lorie, R.: Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In: SIGMOD '92: Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of data, New York, NY, USA, ACM Press (1992) 124–133
11. Reed, D.P.: Naming and synchronization in a decentralized computer system, PhD thesis. Technical Report MIT-LCS-TR-205, MIT (September 1978)
12. Schütt, T., Schintke, F., Reinefeld, A.: Structured Overlay without Consistent Hashing: Empirical Results. In: Proceedings of the Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06). (May 2006)
13. Andrzejak, A., Xu, Z.: Scalable, efficient range queries for grid information services. In: 2nd IEEE Intl. Conf. on Peer-to-Peer Computing (P2P2002). (2002)
14. Plantikow, S.: Transactions for distributed wikis on structured overlay networks (in German). Diploma thesis, Humboldt-Universität zu Berlin (April 2007)
15. Mesaros, V., Collet, R., Glynn, K., Roy, P.V.: A transactional system for structured overlay networks. Technical Report RR2005-01, Université catholique de Louvain (UCL) (March 2005)
16. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: The OceanStore Prototype. In: Proceedings of the 2nd USENIX Conf. on File and Storage Technologies. (2003) 1–14
17. Muthitacharoen, A., Gilbert, S., Morris, R.: Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-CSAIL-TR-2005-044 and MIT-LCS-TR-993, CSAIL, MIT (2005)
18. Moser, M., Haridi, S.: Atomic commitment in transactional DHTs. In: First CoreGRID European Network of Excellence Symposium. (2007)
19. Ghodsi, A., Alima, L.O., Haridi, S.: Symmetric replication for structured peer-to-peer systems. In: The 3rd Intl. Workshop on Databases, Information Systems and peer-to-Peer Computing. (2005)
20. Litwin, W., Schwarz, T.: LH*RS: a high-availability scalable distributed data structure using Reed Solomon Codes. In: SIGMOD '00: Proceedings of the 2000 ACM SIGMOD Intl. Conf. on Management of Data, ACM Press (2000) 237–248

A.12 Transactional DHT Algorithms

Transactional DHT Algorithms

Monika Moser[1], Seif Haridi[2], Tallat M. Shafaat[2], Thorsten Schütt[1], Mikael Högqvist[1], Alexander Reinefeld[1]

[1] Zuse Institute Berlin (ZIB), Germany
[2] Royal Institute of Technology (KTH), Sweden

Abstract. We present a framework for transactional data access on data stored in a DHT. It allows to atomically read and write items and to run distributed transactions consisting of a sequence of read and write operations on the items. Items are symmetrically replicated in order to achieve durability of data stored in the SON. To provide availability of items despite the unavailability of some replicas, operations on items are quorum-based. They make progress as long as a majority of replicas can be accessed. Our framework processes transactions optimistically with an atomic commit protocol that is based on Paxos atomic commit. We present algorithms for the whole framework with an event based notation. Additionally we discuss the problem of lookup inconsistencies and its implications on the one-copy serializability property of the transaction processing in our framework.

Table of Contents

1 Introduction

DHTs are fully decentralized and highly scalable systems that provide the ability to store and lookup data. They use the lookup service of a Structured Overlay Network for Internet-scale applications. The interface DHTs provide on their data is mostly a simple put/get interface. Often data in DHTs is immutable or consistency guarantees on data are weak. However many distributed systems require stronger guarantees like they are given by atomic data operations. We present a framework with transactional access to data stored in a DHT. It provides high availability of data and one-copy serializability for transactions on that data. A transaction consists of a sequence of one or more read and/or write operations that is executed atomically.

DHTs are dynamic systems where nodes are able to join the system or crash at any time. In order to maintain durability and availability of data, items are replicated. Each item consists of a fixed number of replicas. To tolerate the unavailability of a subset of replicas our transaction mechanisms are majority-based. This means that they are able to make progress if a majority of replicas is accessible. Therefore replication factor has to be chosen in a way that the availability of a majority of replicas is very high.

Read and write operations access at least a majority of replicas and choose the one with the highest timestamp. The atomic commit protocol that is needed to coordinate a distributed transaction also makes use of the majority idea. In order to prevent a single transaction manager from blocking the whole protocol if it fails, the framework uses a Paxos based non-blocking atomic commit protocol[7]. There, the single transaction manager is replaced by a set of nodes that all together act as the transaction manager. The protocol makes progress if the majority of these nodes does not fail until every participant in the transaction receives the outcome of the transaction.

In this paper we present the algorithms for our transactional framework. We use an event-based notation as it is well suited to present an asynchronous message-passing system. The framework builds on various techniques known from distributed database systems. The processing of transactions is done optimistically with a non-blocking atomic commit protocol in the end. Concurrency control is included in the atomic commit phase. The basic idea is to either acquire all necessary locks at the same time or to abort the transaction, thus avoiding distributed deadlock detection. Timestamps are used for each replica to determine whether items read in the read phase of the transaction are still valid in the commit phase. As the transaction processing is optimistic locks are only held during the commit phase. We combine the non-blocking Paxos atomic commit protocol with quorum techniques for access on replicated data.

2 Structured Overlay Network

In this section we introduce the model of the SON which underlies our framework. Thereby we refer to self-stabilization mechanisms that are used in Chord

[12]. However our framework is not restricted to a DHT based on Chord but can be applied to other key-based SONs.

2.1 The Overlay Model

A Model of a Ring-based SON. A DHT makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some a priori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has an unique identifier from the identifier space. Each node keeps two pointers: *succ*, to its *successor* on the ring, and *pred* to its *predecessor*. The successor of a node with identifier p is the first node found going in a clockwise direction on the ring starting at p. The predecessor of a node with identifier p is the first node met going anti-clockwise on the ring starting at p. For each node p, a *successor-list* is also maintained consisting of p's c immediate successors, where c is typically set to $\log_2(n)$, where n is the network size.

Ring-based DHTs also maintain additional routing pointers, called fingers, on top of the ring to enhance routing [1]. For our analysis, we assume that these pointers are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(\mathcal{N})$. Our results are independent of the chosen scheme for placing the fingers.

Dealing with Joins and Failures in Chord. A DHT system is a continuously running system and there is no notion of crash recovery. Whenever a node fails there is another node that becomes responsible for the items of the failed nodes. The protocols are based on a crash-stop model of nodes. This implies that if a node crashes and then reboot to re-join the network, it will be considered as a new node.

Chord handles joins and failures using a protocol called *periodic stabilization*. Figure 1 shows part of the protocol presented in [12]. Failures of predecessors are handled by having each node periodically check whether its *pred* is alive, and setting $pred := nil$ if the predecessor is found dead. Moreover, each node periodically checks to see if *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Each node periodically asks for its successor's *pred* pointer, and updates its *succ* pointer if it gets a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*.

Joins are also handled by the ring stabilization protocol. Joining nodes lookup their successor s on the ring, and sets $succ := s$. Periodic stabilization will eventually fix its predecessor and successor. Hence, any joining node is eventually properly incorporated into the ring.

Failure Detectors. DHTs provide a platform for Internet-scale systems, aimed at working on an asynchronous network. Informally, a network is asynchronous if there is no bound on message delay. Thus, no timing assumptions can be made

```
n.join(n')
  predecessor = nil;
  sucessor = n'.findsuccessor(n);

n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor])
    successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n])
    predecessor = n';

n.check predecessor()
  if (predecessor has failed)
    predecessor = nil;
```

Fig. 1: Part of the Chord protocol, presented in [12], which is related to successor and predecessor pointers.

in such a system. Due to the absence of timing restrictions in an asynchronous model, it is difficult to determine if a node has actually crashed or is very slow to respond. This gives rise to inaccurate suspicion of node failure.

Failure detectors are modules used by a node to determine if its neighbors are alive or dead. Since we are working in an asynchronous model, a failure detector can only provide probabilistic results about the failure of a node. Thus, we have failure detectors working probabilistically.

Failure detectors are defined based on two properties: *completeness* and *accuracy* [3]. In a crash-stop model, completeness requires the failure detector to eventually detect all crashed nodes. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not. A perfect failure detector is accurate all the times, while the accuracy of an unreliable failure detector is defined by its probability of working correctly.

2.2 Lookup Consistency and Responsibility

A consequence of imperfect failure detectors are inconsistent lookups and inconsistent responsibilities. We explain these terms in the following. Lookup consistency and responsibility consistency are important concepts when we reason about data consistency in our transactional DHT. Basically responsibility consistency is a requirement for guaranteeing data consistency.

In the following, we define lookup consistency and responsibility consistency, and explain how they can be violated. We use the term **configuration** of a SON to denote the set of all nodes and their pointers to neighboring nodes at a certain

point in time. A SON evolves by either changing a pointer, or adding/removing a node.

Definition 1 *A lookup for a key is consistent, if in a configuration lookups for this key made from different nodes, return the same node.*

Lookup consistency can be violated if some nodes' successor pointers do not reflect the current ring structure. Figure 2a illustrates a scenario, where lookups for key k can return inconsistent results. It shows nodes with their successor and predecessor pointers. This configuration may occur if node $N1$ falsely suspected $N2$ as failed, while at the same time $N2$ falsely suspected $N3$ as failed. A lookup for key k ending at $N2$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N1$ would return $N3$.

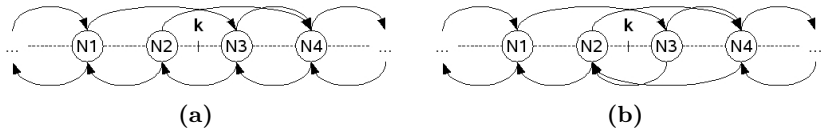


Fig. 2: Lookup inconsistency and responsibility inconsistency. Nodes with successor and predecessor pointers: (a) Example with wrong successor pointers. (b) Example with wrong successor pointers and overlapping responsibilities.

Definition 2 *A node n is said to be* locally responsible *for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. We call a node* globally responsible *for a key, if it is the only node in the system that is* locally responsible *for it.*

The responsibility of a node changes whenever its predecessor is changed. If a node has an incorrect predecessor pointer, the range of keys it is responsible for can overlap with another node's key range. Thus there are several nodes responsible for a part of the key range.

Definition 3 *The responsibility for a key k is consistent if there is a node globally responsible for k.*

A configuration where responsibility consistency for key k is violated is shown in Figure 2b. Here, lookup consistency for k cannot be guaranteed and both nodes, $N3$ and $N4$, are locally responsible for k. However, in Figure 2a, $N3$ is globally responsible despite lookup inconsistency and $N4$ is not responsible. The configuration depicted in Figure 2b can arise from the configuration shown in Figure 2a with an additional wrong suspicion of node $N4$ about its predecessor $N3$.

Lookup consistency and responsibility consistency cannot be guaranteed in a SON. As we will show later responsibility consistency is an assumption of our

system in order to guarantee data consistency. However in [11] we show that the probability for a violation of responsibility consistency is very low. E.g. with a reasonable probability for a failure detector to make false positives with two percent the probability to get consistent responsibility for a replica is more than 99.999%.

2.3 Availability

Another important property in our system is the availability of a key. In order to make progress operations in our system have to be able to access a sufficient number of replicas.

Definition 4 *A key k is available if there exists a reachable node n such that n is locally responsible for k.*

Availability of a key in a SON is both affected by churn and inaccurate failure detectors. Due to churn a key is unavailable when the node that is responsible for it fails until a successor node takes over responsibility and is reachable in the system. This is illustrated in Figure 3b where the key k is unavailable because of the failure of node $N2$. A node n is said to be reachable for a node n', if there exists a path from n to n'. Also during a join process when a node is transferring responsibility for a certain key range to the joining node, keys in that range are unavailable until the joining node is reachable in the system. Figure 3c illustrates a scenario where the joining node $N2$ already took over responsibility for key k but is not yet reachable in the system as node $N1$ has not set its successor pointer to $N2$. In the second case inaccurate failure detectors cause unavailability when a node that falsely suspects its successor will remove the pointer to this node. Thus, keys for which the suspected node is responsible will temporarily become unavailable. In figure 3d node $N1$ suspects node $N2$, thus k becomes unavailable.

3 Transactions on a SON

Usually DHTs provide a simple put/get interface to store and retrieve data. Hardly they provide consistency guarantees on data and often they are restricted to immutable data. Our framework is able to provide a transactional interface on top of a SON. It provides read and write operations that are executed transactionally as well as the ability to execute transactions that consist of a sequence of different operations.

3.1 1-Copy Serializability

Our algorithms provide 1-copy serializability. In our system items are replicated and there might exist replicas with different versions. However transactions produce a serializable history as if there was only one copy available to transactions. A history H of transactions is serializable if all committed transactions in H issue the same operations and receive the same responses as in some sequential history S that consists of the transactions committed in H [8].

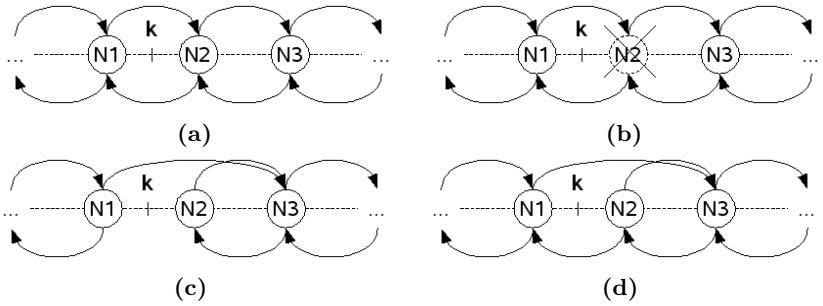


Fig. 3: Availability. Nodes with successor and predecessor pointers: (a) Example where key k is available. (b) Example where k is unavailable due to the failure of $N2$. (c) Example where k is unavailable during the joining process of $N2$. (d) Example where k is unavailable because $N1$ suspected $N2$ to have failed.

3.2 Transaction Processing

Transactions in our system are executed optimistically. They are processed in the following three phases:

- **Read phase (R):** Operations that are part of the transaction are executed within a transaction managers local workspace that is private to the transaction. Changes made by write operations are not visible to other transactions.
- **Validation phase (V):** Once a transaction should be committed, all involved data managers that are responsible for the data that is part of the transaction, check whether the operations are valid. Version numbers are used to determine if another transaction has made changes after the transaction's read phase.
- **Write phase (W):** If all data managers successfully validated the operations on their data, changes can be made permanent.

Atomic Commit. The validation phase and the write phase are executed within an *atomic commit protocol*. An atomic commit protocol coordinates all processes that are involved in a transaction. It ensures that all data managers decide on the same outcome of the transaction. The decision is commit if all data managers are able to validate the operations or abort if there exists at least one data manager that cannot validate an operation. Figure 4 shows a basic commit algorithm called the 2-Phase-Commit Protocol. There is one node called the *transaction manager (TM)* that coordinates the protocol. Data managers are called *transaction participants (TP)*. The TM asks the TPs to validate by sending a *prepare* request. The TPs either reply with *prepared* or *abort*. The TM collects the votes and sends *commit* if all TPs voted to be prepared otherwise it sends *abort*. When a TP receives commit it will make all changes permanent if promised to do when sending the prepared message. If a TP receives abort it won't make any changes permanent. Our framework executes transactions

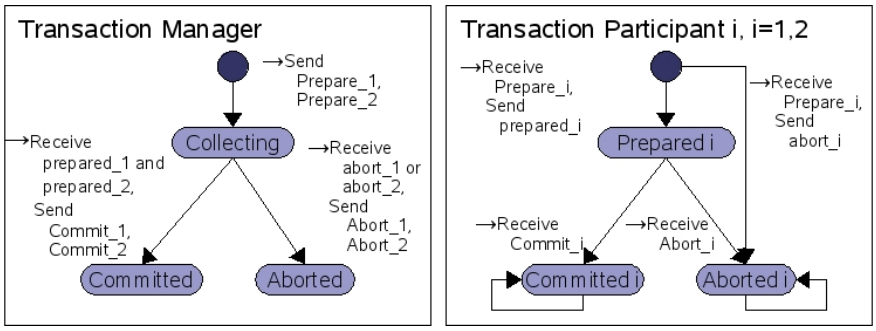


Fig. 4: State-charts for a 2-Phase-Commit Protocol with 2 Participants and 1 Transaction Manager

optimistically in the read phase. Thus the commit protocol will decide on abort if other transactions were committed in between.

A 2-Phase-Commit protocol is blocking if the TM fails in the state collecting and the TPs are not able to retrieve the outcome of the transaction. Therefore we use a non-blocking atomic commit protocol that is based on Paxos [10] in our framework. Gray introduced Paxos Atomic Commit in [7]. It uses replicated transaction managers which all collect votes from the data managers. If the leading transaction manager fails, these replicated transaction managers take over and distribute the decision of the atomic commit protocol to all participants. The Paxos Protocol and the Paxos Atomic Commit protocol are described later in this section.

Concurrency Control. Once a data manger successfully validates an operation on an item it has to ensure that no other transaction gets validated on the same item with a conflicting operation until the atomic commit protocol decides on the outcome. Therefore read and write locks are used during the commit phase that prevent concurrent conflicting validations.

Locks are only held during the atomic commit phase. Instead of letting transactions wait for a lock a TP will vote to abort in the atomic commit phase if it cannot acquire the lock for an operation. In that case the transaction has to be re-executed. This avoids distributed deadlock detection. We assume that read and write operations are less frequent than in traditional database systems. Thus the ratio of aborted transactions should be small.

The Paxos Protocol Paxos is an algorithm which guarantees uniform consensus. Consensus is necessary when a set of nodes has to decide on a common value. Uniform consensus satisfies the following properties: 1. *Uniform agreement*, which means that no two nodes decide differently, regardless of whether they fail after the decision was taken; 2. *Validity* describes the property that the value which is decided can only be a value that has been proposed by some

node; 3. *Integrity*, meaning no node may decide twice and finally 4. *Termination*, every node eventually decides some value [9]. Paxos assumes an eventual leader election to guarantee termination. Eventual leader election can be built by using inaccurate failure detectors.

Paxos defines different roles for the nodes. There are *Proposers*, which propose a value, and *Acceptors*, which either accept a proposal or reject it in a way that guarantees uniform agreement. Paxos as described in [10] assumes that each node may act as both proposer and acceptor. In our solution presented below we use different nodes as proposers and acceptors.

The above mentioned properties of uniform agreement can be guaranteed by Paxos whenever a majority of acceptors is alive. That means, it tolerates the failure of F acceptors out of initially $2F + 1$ acceptors.

Paxos basically consists of two phases called the read and write phase. In the *read phase* a node makes a proposal and tries to get a promise that his value will be accepted by a majority or it gets a value that it must adopt for the write phase. In the *write phase* a node tries to impose the value resulting from the read phase on a majority of nodes. Either the read or write phase may fail. Proposals are ordered by proposal numbers. By using an eventual leader to coordinate different proposals, the algorithm will eventually terminate.

Atomic Commit with Paxos Uniform consensus alone is not enough for solving atomic commit. Atomic commit has additional requirements on the value decided. If some node proposes abort or is perceived to have crashed by other nodes before a decision was taken, then all nodes have to decide on abort. To decide on commit, all nodes have to propose prepared.

In the Paxos Commit protocol [7] we have a set of acceptors, with a distinguished leader, and a set of proposers. The set of acceptors play the role of the coordinator and the set of proposers are those who have to decide in the atomic commit protocol.

Each proposer creates a separate instance of the Paxos algorithm with itself as the only proposer to decide on either prepared or abort. All instances share the same set of acceptors. It can be noted that the Paxos consensus can be optimized, because there is only one proposer for each instance. If a proposer fails, one of the acceptors, normally the leader, acts on behalf of that proposer in the particular Paxos instance and proposes abort.

Acceptors store the decision of all proposers and send the acknowledgment for the vote of a TP's Paxos instance to the leader. Whenever the leader has collected enough acknowledgments for each participant's Paxos instance, it decides on commit if all instances have decided on prepared or it decides on abort if there is at least one Paxos instance of a participant that decides on abort. Thereafter the final abort/commit is sent to the initial proposers. If the leader is suspected by the eventual failure detector, another leader will take over and can extract the decision from a majority of acceptors and complete the protocol.

The state-chart of a proposer is similar to the state-chart of a TP in the original 2PC protocol, as shown in figure 4. Also the state-chart of an acceptor

is similar to that of the TM, referring to the same figure. But instead of sending the decision commit to the participants, the acceptors send the outcome to the leader.

3.3 Replication

To provide higher reliability items are replicated. Each item has a fixed number of replicas. The replication scheme used here is key based. A key based replication essentially means that an item, which is a key-value pair, is stored under r replica keys. Thus, to store an item under key k, the value will be stored in the DHT under keys $Kr = \{k_1, k_2, k_3 \dots k_r\}$. We say that the replication degree is r and for key k, the set Kr to be the set of keys under which k is replicated. Each replica can be accessed symmetrically as the function to determine the replica keys is system-wide known [4].

As SONs usually are highly dynamic systems, operations on an item should make progress despite the unavailability of a number of replicas. Reads and writes thus require that a majority of replicas is accessible. A minority of replicas might be temporarily unavailable without hindering progress. Operations in our SON use majority-based algorithms. Majority-based algorithms are a special case of quorum algorithms. Quorum algorithms were introduced by Gifford [6] in order to maintain replicated data. Each replica is assigned a certain amount of votes. Read operations have to collect rv votes and write operations have to collect wv votes, where $rv + wv$ exceeds the total number of votes assigned to all replicas of an item. This ensures that read operations include at least one replica that was included by the latest write operation. In majority-based algorithms each replica is assigned exactly one vote and read and write operations have to include a majority of $m = \lfloor \frac{r}{2} \rfloor + 1$ votes. Thus they intersect in at least one replica.

3.4 Serializability in Presence of Responsibility Inconsistency

In order to ensure that $rv + wv$ always exceeds the total number of votes assigned to all replicas, the number of replicas in the system has to be constant for our majority-based algorithms. Each operation on an item has to ensure that it includes at least a majority of replicas, while the majority is based on the system's replication factor r. An additional replica that is added to the system would violate the above mentioned condition. However a responsibility inconsistency is equal to adding an additional replica. In that case two conflicting operations might end up with working on two disjoint sets with a majority of replicas, which we call *majority set*. This happens if two operations work on majority sets that both include distinct nodes that are involved in a responsibility inconsistency for one replica, but have no other replica in common. In that case it is not possible to detect a conflict between these operations, which can violate serializability. As it is not possible to ensure responsibility consistency, it is not possible to ensure serializability. However the existence of a responsibility inconsistency does not necessarily implicate disjoint majority sets for two conflicting operations.

In [11] we calculated the probability for two operations in one configuration to work on non-disjoint majority sets. If the probability for a failure detector to make false positives is 2% and therefore the probability to have a consistent responsibility is 99.999%, the probability for non-disjoint majority sets is 99.9999% if $r = 3$.

4 Transaction Algorithms

In this section we present the algorithms for the transactional DHT. We use an event based notation similar to the one used in [9].

4.1 System Architecture

Nodes in the system can take different roles in a transaction. For each transaction there exists the role of a leading Transaction Manager (TM), called *Leader*, which is the node the client is connected to. Additionally, a number of replicated Transaction Managers (rTM) are created according to the set of acceptors in Paxos commit. Nodes that are responsible for a replica of an item that is involved in the transaction have the role of a Transaction Participants (TP) in the protocol. Each node of the SON can have any number of TMs and TPs that are involved in different transactions. If there are multiple TPs on a node, they must share the database that contains the items with information about read and write locks. Each TP maintains a set of records for ongoing transactions, that have not yet been committed. Each record has a transaction ID, the new proposed value for the items the TP maintains and the new proposed version.

4.2 Transaction ID and Transaction Item

The leader of each transaction creates an unique transaction ID (TID). This ID is part of the SON's key space and can be treated like a item key. The leader creates the TID in a way such that it has a replica key of TID in its own key space. According to the replication scheme there are $r-1$ additional replica keys for the TID. The set of rTMs is determined by the nodes that are responsible for these associated replica keys. Thus the number of all TMs (Leader + rTMs) is equal to the replication factor r. At the end of a transaction each TM will store a replica of a so called *transaction item* with $\{TID, Decision\}$ as the {key, value} pair. We assume that a majority of rTMs does not change its responsibility such that the replica of the TID would not be part of its key space any more. Therefore a node that did not recieve the decision of the transaction can retrieve the decision by doing a quorum read on an item with the TID as key. The transaction item is maintained in the same way as normal items are. However it has to be garbage collected after a certain time.

4.3 System Assumptions

We identify the assumptions related to liveness, no nodes are blocked, and safety, no data is corrupted and 1-copy serializability is not violated. For liveness, it is assumed that direct communication between nodes as well as the bulk procedure is reliable. In addition, a majority of TMs must be alive and keep the TID within their range of responsibility until all alive TPs receive the transaction decision. This assumption is an extension of Paxos where all acceptors (TMs) must be alive during the protocol. For safety, we assume that a majority of replicas for an item are alive and that a majority of lookups targeting these replicase are consistent. A violation of the safety requirements may lead to inconsistent state. The probability of this happening is directly related to the replication factor.

4.4 Identifiers, Modules and Operations

Identifiers. Figure 5 lists all identifiers and variables used in the algorithms. The first part contains general identifiers that are used at transaction managers and transaction participants. The second part contains variables that are maintained by a transaction manager. Additionally, we introduce structures for votes that are received by transaction managers and for acknowledgments of votes. The last part contains variables that store information kept by a transaction participant.

External Modules Used in the Algorithms. The following modules are used by the algorithms

- EventuallyPerfectFailureDetector ($\Diamond P$) [9]
- EventualLeaderDetector (Ω) [9]

A leader uses a failure detector on every replica of the involved items. If it does not get a vote for a replica within a certain time threshold, it will start a failure handling procedure. A failure detector raises the event SUSPECT(tp) when it suspects the transaction participant tp to have failed. A leader election mechanism is used to guarantee progress of the atomic commit protocol. The set of replicated transaction manager will elect a new leader if they suspect the leader to have failed. The leader detector module raises the event TRUST($newleader$) to install a new leader.

Bulk Operation. The algorithms make use of a so called bulk operation [5]. This operation sends events to all nodes that are responsible for a key in a specified set of identifiers. E.g. a read operation on an item can be done with a bulk operation on the set of replica keys for that item.

Identifiers	
item	record
item.key	key
item.val	value
item.ts	timestamp/version number
item.op	kind of operation: write or read
tm	transaction manager
tp	transaction participant
r	replication degree of the system
Information maintained by a TM	
tid	ID of the transaction
TPs	set of Transaction Participants
TMs	set of replicated Transaction Managers
I	set of items involved in the transaction
Votes	Votes of the participants
AcksTMs	Acknowledgments sent by the TMs to the Leader
outcome	Overall outcome of the transaction
state	Either collectingNodes/collectingVotes/locallyDecided/decided
Suspected	set of nodes which are suspected to have failed
leader	the address of the leader
client	the address of the client issuing the transaction
vts	timestamp of a vote
rvts	timestamp of a vote acknowledged in a read phase
wvts	timestamp of a vote acknowledged in a write phase
ItemsInTrans	set of items that are currently involved in a transaction
Information contained in a vote	
i.key	key of the item the vote refers to
rkey	the key of the replica
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal - number of the proposal
Votes[i.key][rkey] = (vote, rts, wvts)	
vote	PREPARED/ABORT decision of a tp
rvts	timestamp of the vote that was accepted during the read phase
wvts	timestamp of the vote that was accepted (write phase)
AcksTMs[i.key][rkey]: {(vote, vts)*}	
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal that was accepted (write phase)
Information kept by a TP	
tid	ID of the transaction
TMs	transaction managers
i	the item
decision_of_tp	the decision it made

Fig. 5: Identifiers used in the algorithms

4.5 Algorithms

In the following we present the algorithms for the transaction processing. We first show the algorithm for the fault-free scenario. The algorithms for failure handling are shown separately in Section 5. The whole transaction processing algorithms refers to the execution of exactly one transaction. Thus we commit information that identifies a particular transaction for better readability.

The algorithms can be structured into different phases. Figure 6 identifies two main phases of a transaction. One is the *Read Phase* where the client determines the operations that are part of the transaction. The second one is the *Commit phase* which we further divide into *Initialization*, *Validation* and *Consensus*.

In the initialization phase the leader determines all nodes that act as replicated transaction managers (TMs). It determines the nodes by a key based search (lookup). After initialization these nodes communicate directly with each other without using a key based search. In the validation phase all TPs are sent a prepare request by a key based search. They are asked to validate the operations on the items they are responsible for. After validation the consensus on the outcome of the transaction is started, based on the validation results. The outcome is sent to the TPs directly with out doing a lookup.

4.6 Read Phase

During the read phase the client determines the operations that are part of the transaction. It can be any sequence of read and write operations. A client is connected to a certain node in the system. This node becomes the initial transaction manager which will act as the leader during the protocol. Figure 7 shows the particular communication steps in the read phase. The client instructs the leader to start a transaction (Algorithm 1) and to do operations until the client tells the leader to commit the transaction. The leader keeps track of the operations and keeps updates on items private to its local workspace. For read operations it will retrieve the value and version number of the item the client wants to read, while for write operation it has to retrieve the version number only (Algorithm 2). When the client signals the end of the transaction the leader will start a commit phase. Instead of instructing the leader to commit the transaction the client can also instruct it to abort the transaction before the commit phase. E.g. if the client reads a value that does not meet a certain condition. In that case the leader can simply throw away logged information on that transaction as the TPs have not yet made changes to their state, such that they do not have to be notified about this user triggered abort. Once a client tells the leader to commit a transaction the client cannot abort it any more on its own behalf.

Algorithm 2 includes a function *latest(Items)* which extracts the item with the highest version number from a set of items. It uses a *DB(key, rkey)* function that reads a replica from the local database. The replica is identified by the key of the item and a replica key or replica number. The function *replicakeys(key)* return all keys of replicas for a certain key.

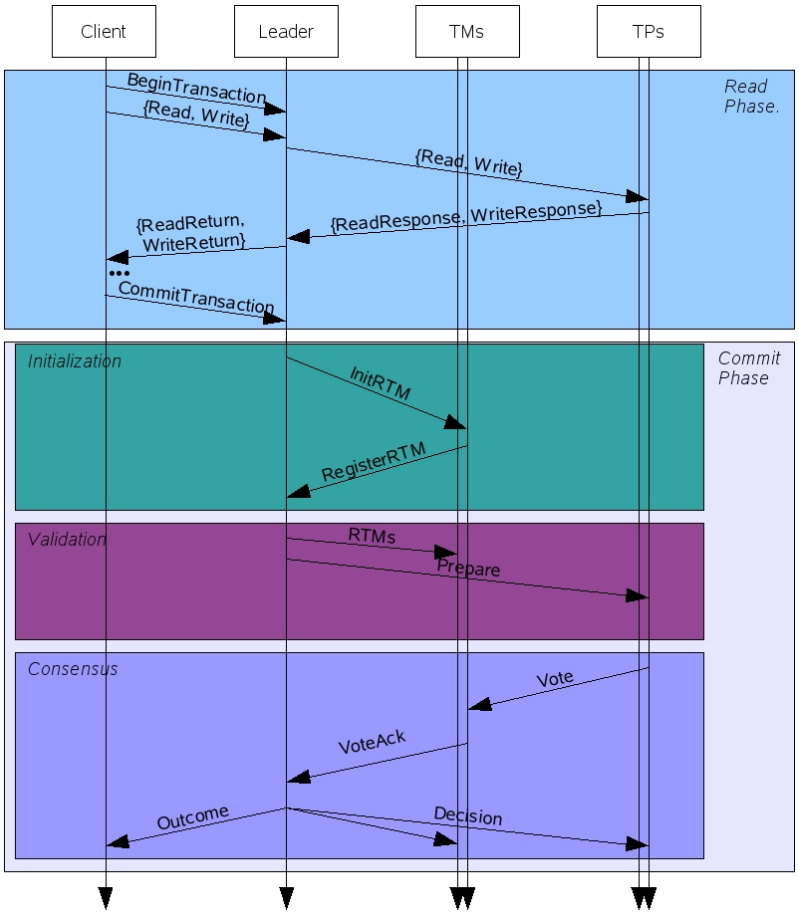


Fig. 6: The figure shows the different phases for a transaction together with the messages sent between the participating nodes

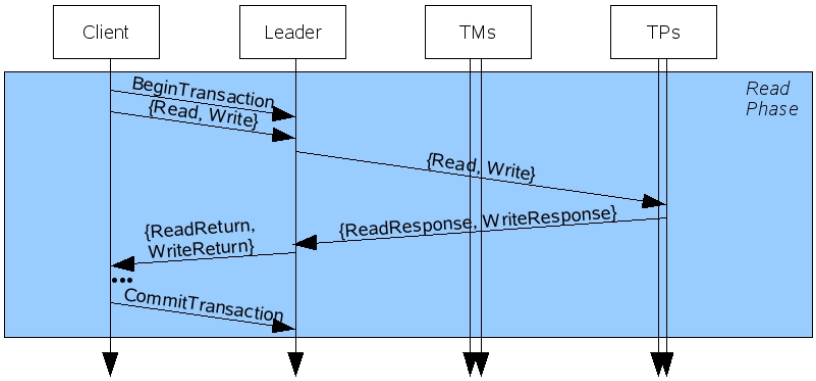


Fig. 7: The figure shows the messages which are part of the read phase. To start a transaction a client issues a BeginTransaction and signals the end of a transaction by a request to commit the transaction. In between it will add several *read* and *write* operations.

Algorithm 1 Interface to the Transaction Manager: Client signals Begin and End of a Transaction

```
1: upon event beginTransaction() from client at tm
2:     client := client
3:     I := ∅
4:     readID:= writeID:= ⊥
5:     tid := generateTID()
6: end event

7: upon event commitTransaction() from client at tm
8:     trigger startCommit()
9: end event

10: upon event abortTransaction() from client at tm
11:     delete information on transaction
12: end event
```

Algorithm 2 Processing of a Read Operation due to a Client's Read Request

```
1: function latest(Items) returns item is
2:     tmp_item := item{key:= ⊥, val:=⊥, ts:=−1, op:=⊥}
3:     foreach i in Items do
4:         if i.ts >tmp_item.ts then
5:             tmp_item := i
6:         end if
7:     end foreach
8:     return tmp_item

                                ▷ A client requests a read operation at the Transaction Manager
9: upon event READ(key) from client at tm
10:    readID := createRID()
11:    Reads := ∅
12:    trigger BULK(replicakeys(key), {READ, key, readID})
13: end event

                                                              ▷ At the Transaction Participant
14: upon event BULK(rkey,{READ, key, readID}) from tm at tp
15:    i := DB(key, rkey)
16:    sendto tm : READRESPONSE(key, i.val, i.ts, readID)
17: end event

                                                                  ▷ At the Transaction Manager
18: upon event READRESPONSE(key, val, ts, id) from tp at tm
19:    if readID=id then
20:        Reads := Reads ∪{(key, val, ts)}
21:    end if
22: end event

23: upon |Reads| ≥ (⌊r/2⌋ + 1) ∧ readID ≠⊥ do
24:    (k, val, v):= latest(Reads)
25:    I:= I ∪ item{key:= k, val:=val, ts:=v, op:=r}
26:    sendto client : READRETURN(value)
27:    readID := ⊥
28: end event
```

Algorithm 3 Processing of a Write Operation due to a Client's Write Request

▷ A client requests a write operation at the Transaction Manager
1: **upon event** WRITE($key, value$) **from** $client$ **at** tm
2: writeID := createWID()
3: Writes := ∅
4: **trigger** BULK(ReplicaKeys(key), {WRITE, key, writeID})
5: **end event**

▷ At the Transaction Participant
6: **upon event** BULK($rkey$,{WRITE, key, $writeID$}) **from** tm **at** tp
7: i := DB(key, $rkey$)
8: **sendto** tm : WRITERESPONSE($key, i.ts, writeID$)
9: **end event**

▷ At the Transaction Manager
10: **upon event** WRITERESPONSE(key, ts, id) **from** tp **at** tm
11: **if** writeID=id **then**
12: Writes := Writes ∪{(key, ts)}
13: **end if**
14: **end event**

15: **upon** |Writes| ≥ ($\lfloor r/2 \rfloor + 1$) ∧ writeID ≠⊥ **do**
16: (k, v):= latest(Writes)
17: I:= I ∪ $item${key:= k, val:=$value$, ts:=v+1, op:=w}
18: **sendto** client : WRITERETURN(success)
19: writeID := ⊥
20: **end event**

4.7 Commit Phase

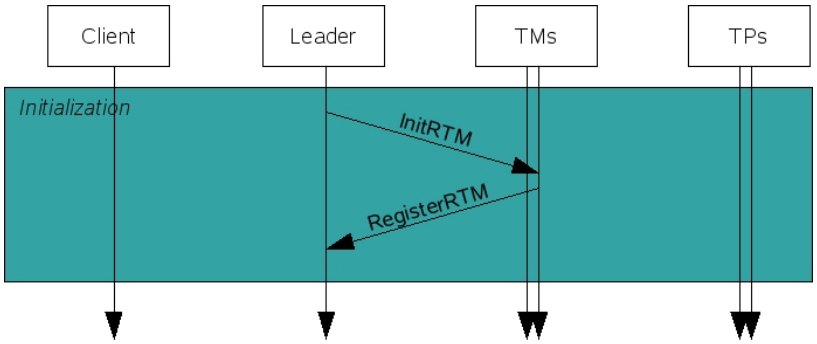


Fig. 8: Before starting the commit protocol the Leader determines the nodes that act as replicated TMs.

Initialization Figure 8 shows the course of events of the initialization phase. Initially nodes that have to act as rTMs are determined by a key based search based on the replica keys of the transaction ID. They have to be known before the commit protocol is started in order to enable leader election among the TMs and make the protocol fault-tolerant and non-blocking. In the next phase the leader will tell the rTMs the addresses of all other rTMs. As long as a majority of rTMs is reachable the protocol can decide on an outcome of the transaction.

Algorithm 4 contains the events and event handlers of the initialization phase. A node that gets a request to initialize a rTM has to initialize its data structures to collect the votes and the acknowledgments. For each item key and its replica keys the vote is initialized with a *rts* (read timestamp) with a value 1. The reason is that a TP will immediately start a write phase in its Paxos instance as it can be sure to be the first one that votes in that particular instance [7]. A TP's proposal number thus is 1.

Once the leader has collected enough TMs it can start the next phase. The leader always tries to collect all TMs. However if some of these nodes do not respond it can start the next phase after a timeout if it has collected at least a majority of TMs. A majority of TMs including the leader is necessary to guarantee progress for Paxos atomic commit.

Algorithm 4 Initialize the involved processes

```
1: upon event startCommit() at leader
                                        ▷ client is the process issuing the transaction
2:     trigger bulk(replicas(tid), {InitRTM, leader, tid, I, client})
3: end event

4: upon event bulk(rtid, {InitRTM, l, id, Items, cl}) from s at rtm
5:                                      ▷ A new transaction manager instance is created
6:     leader := l
7:     tid:= id
8:     I := Items
9:     client := cl
10:    foreach i in I do
11:        foreach rkey in replicaKeys(i.key) do
12:            Votes [i.key][rkey] := (⊥, 1, 0)
13:            AcksTMs [i.key][rkey] := ∅          ▷ Necessary if it becomes a leader
14:        end foreach
15:    end foreach
16:    sendto leader : registerRTM(rtm)
17:    state:= collectingVotes
18: end event

19: upon event registerRTM() from rtm at tm
20:    TMs := TMs ∪ rtm
21: end event

22: upon (|TMs| = r) do
23:    trigger startValidation
24: end event
```

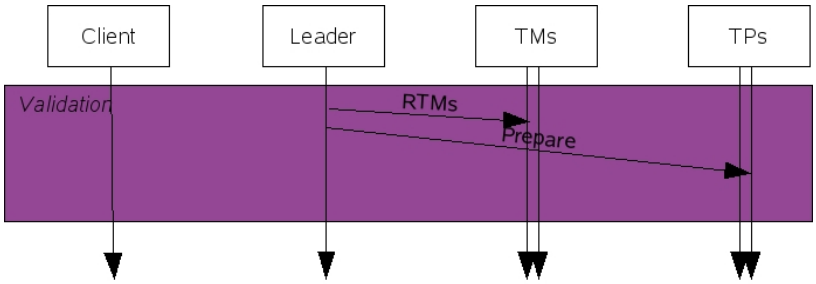


Fig. 9: The Leader sends a Prepare messages to the TPs which will start the validation and tells the TMs about all rTMs.

Validation After the initialization phase the leader tells each rTM the addresses of the other rTMs such that these are able to run a leader election mechanism among them. Additionally the leader sends the prepare request to the TPs together with the addresses of the nodes that act as rTMs. The prepare request is sent with a lookup operation on all replicas of the involved items. Figure 9 shows the course of events and Algorithm 5 the event handlers at the TMs.

The TPs check whether they can validate the operations sent by the prepare request. Each TP therefore locally applies the concurrency control mechanism. This is based on timestamps and locks. A node uses two dictionaries *readLock* and *writeLock* that contain locks on items. While write locks are exclusive, several read locks can be set at the same time. The locks are globally stored in a node. The $storeToLOG(Params)$ is a function that stores any information on a transaction in a TP's LOG. The $getFromLOG()$ function accordingly gets the information from the LOG. Algorithm 6 contains the procedure for validation.

For read operations a TP checks whether there is no lock for a concurrent write and whether the timestamp of the read request is valid, i.e. larger than or equal to the local item. If both checks are successful it will add a read lock and return prepared. For write operations the TP first ensures that there are no read or write locks set for concurrent conflicting operations. Then it compares the timestamp of the proposed item and the local item. This timestamp must to be equal to the currently stored timestamp + 1. If this is not the case, it means that a write operation has changed the item since it was accessed during the read phase. If both of the checks were successful the procedure will return prepared, otherwise abort. After the validation procedure, the TP starts to propose in a Paxos instance for this particular validation result.

Algorithm 5 Transaction Manager: Sending of a Prepare request

```
1: upon event startValidation at tm
2:     sendtoall TMs : RTMs(TMs, I)
3:     foreach i in I do
4:         trigger bulk(replicas(i), {Prepare, leader, tid, i, rkey, TMs})
5:     end foreach
6:     state:= collectingVotes
7: end event

8: upon event  RTMs(rtms, I) from tm at rtm
9:     TMs:= rtms
10:    (Ω).init(rtms)
11:    foreach i in I do
12:        ◇P.init({replica : (i.key, rkey) ∈ i})
13:    end foreach
14: end event
```

Algorithm 6 Validation Procedure at a Transaction Participant/Concurrency Control

```
1: upon event PREPARE(item, rkey, tid, TMs) from tm at tp
2:     ItemsInTrans:= ItemsInTrans ∪ (rkey, tid)
3:     vote:= validate(item, rkey)
4:     trigger PROPOSE(item.key, rkey, TMs, vote, 1)
5: end event

6: procedure validate(item rkey, tid) returns PREPARED/ABORT is
7:     i:= DB(item.key, rkey)
8:     result := ⊥
9:     if item.op = read then
10:        if writeLock[(i.key, rkey)] = ⊥ & (item.ts ≥i.ts) then
11:            readLock[(i.key, rkey)] := readLock[(i.key, rkey)] + 1
12:            storeToLOG(tid, item.key, item.ts,r, rkey,PREPARED)
13:            result:= PREPARED
14:        else
15:            storeToLOG(tid, item.key, item.ts,r, rkey,ABORT)
16:            result := ABORT
17:        end if
18:    else
19:        if writeLock[(i.key, rkey)] = ⊥ & readLock[(i.key, rkey)] = ⊥ & (item.ts =
       i.ts+1) then
20:            writeLock[(key, rkey)] := 1
21:            storeToLOG(tid, item.key, item.ts, item.val,w, rkey, PREPARED)
22:            result:= PREPARED
23:        else
24:            store (tid, item.key, item.ts, item.val,w, rkey, ABORT)
25:            result := ABORT
26:        end if
27:    end if
28:    return result
```

Consensus This phase uses the Paxos atomic commit protocol [7]. At the end of it each TP has to receive the same decision on the transaction to ensure the uniform consensus properties. The decision will be commit if the decision for all items is prepared, it will be abort if the decision for at least one item is abort.

A Paxos instance is started for each replica. The TP that is responsible for the replica uses this Paxos instance to distribute its vote on the set of replicated transaction managers that act as the acceptors. The replicated transaction managers accept the vote of a TP if they did not get a read request with a higher timestamp for the particular Paxos instance. Instead of sending the acknowledgment to the TP they send it to the leader (Algorithm 8). The reason is that the decision on the outcome of the atomic commit protocol is based on the decisions for all items. Therefore, the leader collects all acknowledgments from where it can derive the outcome of the atomic commit protocol. As soon as the outcome is known the leader sends it to all involved nodes and notifies the client.

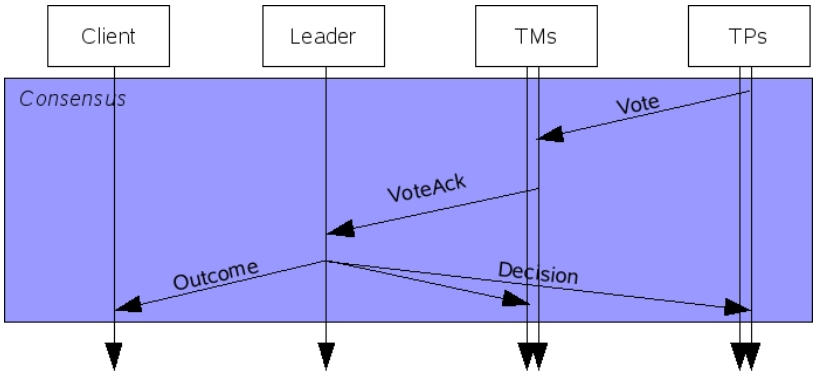


Fig. 10: After validation the TPs start a consensus to distribute their validation result. The leader will collect the decision for each particular consensus instance and conclude on the overall outcome of the transactions based on these instances.

As items in our DHT are replicated and operation on items require at least a majority of replicas to be accessible, the decision for an item also has to be based on a majority of replicas. Therefore the TM collects the votes of the TPs per item. A decision for an item is prepared if a majority of TPs that are responsible for a replica of that item vote to be prepared. The decision for an item is abort if there cannot be a majority of TPs that are responsible for a replica that vote to be prepared (Algorithm 9).

A TP that retrieves the decision for a transaction reads the information about the item it voted for from its LOG (Algorithms 10 and 11). It has to reset locks and write the item to the database if necessary. The learners are related to replica maintenance, which is explained in Section 6.

A TM that retrieves the decision for a transaction stores the decision in a special item, called *transaction item*, by calling $storeTransactionItem(decision)$. This item is replicated like all other items. The key for this transaction item can be deduced from the transaction ID. All TMs are nodes that are responsible for a replica key of the transaction item. If a node does not receive the decision by the normal execution of the protocol it can retrieve the decision by reading the transaction item like every normal item. We assume that a majority of replicated transaction managers may not fail and may not change their responsibility of the key range where the replica key of the TID is a member of until the outcome of the transaction is stored in the DHT. This can be achieved by choosing a proper replication factor.

Algorithm 7 Start Paxos Atomic Commit: Propose in a Paxos Instance

```
1: upon event PROPOSE(key, rkey, TMs, vote, ts) at p
2:     if ts=1 then
3:         sendtoall TMs : VOTE(key, rkey, vote, ts)
4:     else
5:         sendtoall TMs : READVOTE(key, rkey, ts)        ▷ See failure handling in 13
6:     end if
7: end event
```

Algorithm 8 Paxos Atomic Commit - Write Phase

```
1: upon event VOTE(key, rkey, vote, vts) from n at tm   ▷ n is either a tp or a tm
2:     if vts = 1 then
3:         TPs[key]:= TPs[key] ∪ (p, rkey)
4:     end if
5:     (currVote, rvts, wvts) := Votes[key][rkey]
6:     if vts ≥rvts and vts ≥ wvts then
7:         Votes[x.key][rkey] := (vote, rvts, vts)
8:         sendto Leader : VOTEACK(key, rkey, vote, vts)
9:     end if
10: end event

11: upon event VOTEACK(key, rkey, vote, vts) from tm at leader
12:     AcksTMs[key][rkey] := AcksTMs[key][rkey] ∪ {(vote, vts)}
13: end event
```

Algorithm 9 Paxos Atomic Commit - Making the Decision

▷ Ensure that a majority of TMs has stored the decision for a particular replica
1: **function** isPreparedReplica($i, rkey$) **returns** boolean **is**
2: acks := AcksTMs[$i.key$][$rkey$]
3: **return** $\exists vts : |\{(\text{PREPARED}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

4: **function** isAbortReplica(i, n) **returns** boolean **is**
5: acks := AcksTMs[$i.key$][$rkey$]
6: **return** $\exists vts : |\{(\text{ABORT}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

▷ Check whether the votes for a majority of replicas for an item is PREPARED
7: **function** isPrepared(i) **returns** boolean **is**
8: **return** $|\{rkey : \text{isPreparedReplica}(i, rkey)\}| \geq \lfloor r/2 \rfloor + 1$

9: **function** isAbort(i) **returns** boolean **is**
10: **return** $|\{rkey : \text{isAbortReplica}(i, rkey)\}| \geq \lceil r/2 \rceil$

11: **upon** $\forall i \in$ I: isPrepared(i) **at** *leader* **do**
12: **sendtoall** TPs : DECISION(COMMIT)
13: **sendtoall** TMs : DECISION(COMMIT)
14: state:= DECIDED
15: **sendto** client : OUTCOME(COMMIT)
16: **end event**

17: **upon** $\exists i \in$ I: isAbort(i) **at** *leader* **do**
18: **sendtoall** TPs : DECISION(ABORT)
19: **sendtoall** TMs : DECISION(ABORT)
20: state:= DECIDED
21: **sendto** client : OUTCOME(ABORT)
22: **end event**

23: **upon event** DECISION($decision$) **from** tm **at** tm
24: storeTransactionItem($decision$)
25: state:= DECIDED
26: **end event**

27: **upon event** DECISION($decision$) **from** tm **at** tp
28: **trigger** DECIDE($decision$)
29: **end event**

Algorithm 10 Paxos Atomic Commit - Decide COMMIT at a TP

```
1: upon event DECIDE(COMMIT) at tp
2:     if not stored(COMMIT) then
3:         item := getFromLOG(tid)
4:         if Item = ⊥ then
5:             sendafterdelay(time, DECISION(COMMIT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)] := 0
15:                    DB(key, rkey) := (val, ts)
16:                else
17:                    DB(key, rkey) := (val, ts)
18:                end if
19:            end if
20:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
21:            foreach l in learners do
22:                remove(learners, Learners[rkey])
23:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
24:                if ltidrest = ∅ then
25:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
26:                end if
27:            end foreach
28:            ItemsInTrans := ItemsInTrans \rkey
29:            storeToLOG(COMMIT)
30:        end if
31:    end if
32: end event
```

Algorithm 11 Paxos Atomic Commit - Decide ABORT at a TP

```
1: upon event DECIDE(ABORT) at tp
2:     if not stored(ABORT) then
3:         item := getFromLOG()
4:         if item = ⊥ then
5:             sendafterdelay(time, DECISION(tid, ABORT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                    ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)]) := 0
15:                end if
16:            end if
17:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
18:            foreach l in learners do
19:                remove(learners, Learners[rkey])
20:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
21:                if ltidrest = ∅ then
22:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
23:                end if
24:            end foreach
25:        end if
26:        storeToLOG(ABORT)
27:    end if
28: end event
```

5 Transaction Algorithms: Failure Handling

This section covers failure handling of the atomic commit protocol during the consensus phase. Critical failures are those that block the transaction participants leaving a replica in a locked state. These can occur after the leader has sent the prepare request to the TPs. If the leader fails before that or if the leader cannot contact enough TPs or rTMs the transaction will simply be aborted. Failures have to be handled in a way that guarantees progress of the transaction processing while guaranteeing the properties of uniform consensus and atomic commit. In the following we distinguish between the failure of a TP and the failure of a leader. The failure handling reflects the one described in the Paxos atomic commit paper[7].

5.1 Failure of the Leader

When the leader fails the leader election mechanism will elect a new leader among all TMs. This new leader has to retrieve enough acknowledgments from the TPs' Paxos instances in order to be able to decide on the outcome of the transaction. Therefore it starts with a read phase in each single Paxos instance of the TPs (Algorithm13). In the write phase it will adopt the votes that have been proposed so far or vote to abort if there hadn't been a vote in the Paxos instance. As soon as the new leader got enough acknowledgments it can distribute the outcome of the commit phase. Figure 11 shows the course of events when a leader fails.

Algorithm 12 Replicated Transaction Manager: Trusting a New Leader

```
1: upon event TRUST(leader) at tm
2:     Leader := leader
3:     newts := nextVts()
4:     if state ≠ DECIDED ∧Leader = self then
5:         foreach i in I do
6:             foreach rkey in replicas(i.key) do
7:                 trigger PROPOSE(i.key, rkey, TMs, ⊥, newts)
8:             end foreach
9:         end foreach
10:    end if
11: end event
```

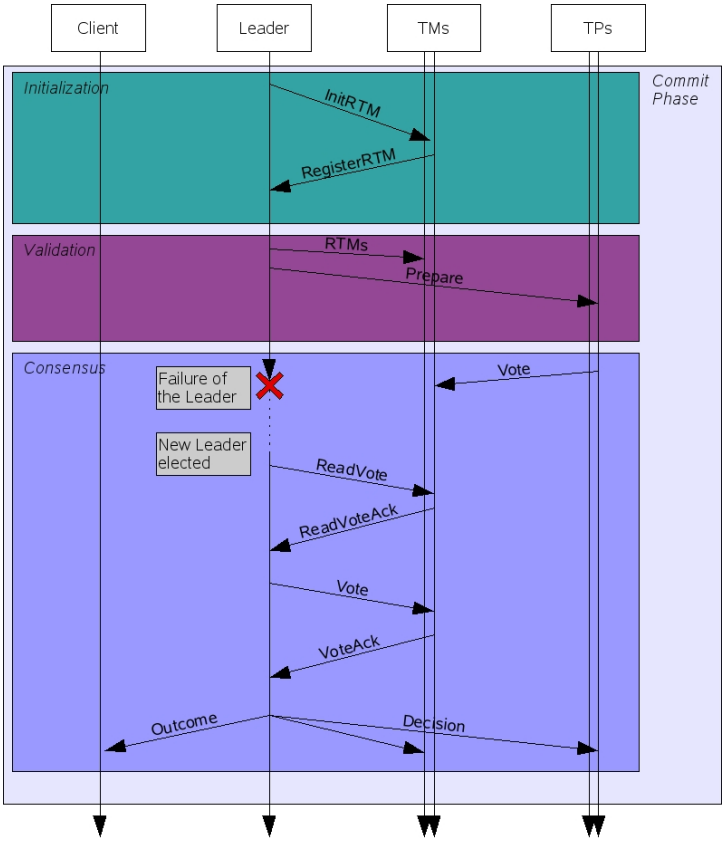


Fig. 11: If a Leader fails, another node will become the new leader and start a read phase in each TP's Paxos instance.

Algorithm 13 Paxos Atomic Commit - Read phase

```
 1: upon event readVote(i.key, rkey, vts) from leader at tm
 2:     (currVote, rvts, wvts) := Votes[i.key][rkey]
 3:     if vts >rvts and vts >wvts then
 4:         Votes[i.key][rkey] := (currVote, vts, wvts)
 5:         sendto Leader : readVoteAck(i.key, rkey, vts, (currVote, wvts))
 6:     end if
 7: end event

 8: upon event readVoteAck(i.key, rkey, vts, (vote, wvts)) from tm at leader
 9:     ReadVotes[i.key][rkey][vts] := ReadVotes[i.key][rkey][vts] ∪ {(vote, wvts)}
10: end event

11: upon ∃i.key, rkey, vts : | ReadVotes[i.key][rkey][vts]| ≥ ⌊r/2⌋ + 1 at Leader do
12:     vote := highest(ReadVotes[i.key][rkey][vts])
13:     if vote = ⊥ then
14:         vote := ABORT
15:     end if
16:     sendtoall TMs : vote(i.key, rkey, vote, vts)
17: end event
```

5.2 Failure of a TP

It must be noted that the protocol makes progress as long as foreach item a majority of TPs that are responsible for a replica is alive. In that case it is not necessary to start a failure handling for a TP. If there exists one item for which a majority of TPs cannot be reached, a leader would have to start voting in as many Paxos instances it needs to get a majority of votes for the item. However this situation occurs only if the system is broken, as we assume that for each item a majority of nodes being responsible for a replica is always available.

When the leader does not get a vote for a replica it suspects the corresponding TP to have failed. It will start to propose in the Paxos instance for that replica. As it does not know whether its suspicion is correct or the TP is alive and has already started to vote, the leader starts with a read phase. It will learn from the TMs whether there has been a vote made by the TP. If this is not the case the leader is free in its decision and will start a write phase with the value abort. Thus the leader decides to abort on behalf of the suspected TP.

Algorithm 14 Leader: Suspecting a TP during Consensus - Start a Read Phase

```
1: function nextVts() returns vts is
2:     select the next vts depending on the nodes key
3:                                  ▷ TM_r will come with a proposal numbered r+1
4:                                  ▷ Set of TMs: TM_1, TM_2, .., TM_r
5:

6: upon event SUSPECT((key, rkey)) at leader
7:     foreach i in I do
8:         trigger PROPOSE(key, rkey, TMs, nextVts())
9:     end foreach
10: end event
```

6 Transactional Replica Maintenance

When a node joins the structured overlay network it will take over the responsibility for a certain key range from an existing node in the system. Similarly, when a node fails, its successor in the structured overlay network has to take over responsibility for the failed node's key range. A node knows that its responsibility has changed whenever it has to update its predecessor pointer.

The framework has to handle joins and failures of a node on the data level in a way that maintains data consistency. Handling these events on the data level is done after they are handled on the overlay level. E.g. a joining node n first sets its successor and predecessor pointers and notifies the corresponding nodes about itself. Thereafter the new node n has to retrieve the data it is responsible for. Data retrieval mechanisms may not decrease the number of up-to-date copies for an item and they may not violate serializability of transactions.

Whenever the predecessor of a node is changed the event is handled on the data-level. There are two possible situations. Either the range the node is responsible for was increased or it was decreased. In the first case the node has to fetch the data it has not stored yet, in the second case the node has to drop data it is no longer responsible for.

A new copy of an item can be initialized by any transaction that writes that item or it is initialized explicitly[2]. As the first possibility may potentially take a long time and other copies of the item might fail during that time, it is better to do an explicit initialization of the copy to decrease the probability of a system failure. The system fails if an operation that is performed on a majority of nodes holding a replica cannot return an up-to-date item. Explicit initializing of a new copy can be done by a copy operation that reads the existing copies of the item and stores the current value in the new copy. Additionally all transactions that update the item must know of the new copy. Otherwise a non-serial history could arise by the following three events:

- Transaction $T1$ updates x and y
- Node $N1$ joins and will become responsible for replicas x_1 and y_1

– Transaction $T3$ reads x and y

The non-serial sequential history of events and transaction phases that can occur:

1. $T1$ - Initialization phase: Get addresses of involved nodes
2. $T1$ - Validation phase: Send prepare request with lookup
3. $N1$ joins the ring (updates its successor, predecessor and the others)
4. $N1$ copies x_1 from an existing copy: Gets the old value
5. $T1$ - Consensus phase: Outcome is received by all TPs
6. $N1$ copies y_1 from an existing copy: Gets the new value
7. x_1 at $N1$ is out of date

In this situation the number of replicas that are out of date would be increased. This happens if the initialization of an item does not take into account ongoing transactions on the item. The copy operation to initialize a replica has to be done either with a transaction or by adding the new node as learner to ongoing transactions, as we will explain in the following. A node that has to initialize the replicas in its range has to know which items exist that have replicas in that range and it has to copy the data for these replicas. If the initialization would have been done by one transaction this transaction would fail if there is even one single write operation on one of the items. Another possibility is that all ongoing transactions have the new node as a so called *learner* for the outcome. This concept is used in our framework and introduced in the following.

6.1 Copy Operation

The operation that initializes copies in a certain key range will be called *copy operation* in the following. Within this operation a node asks the nodes that are responsible for the corresponding remaining replicas of the items in its new range to send their replicas to it. It has to read from at least a majority of replicas. Once the nodes being responsible for the remaining replicas get a copy request, they have to remember the new node as a learner for all ongoing transactions. They send to the new node all replicas that are not involved in a transaction. Items that are currently involved in the transaction are sent as soon as there is no transaction run on them any more for which the new node is registered as learner. This is a way to let all ongoing transactions know the new copy. The new node will not be able to answer requests for items in its range until it has retrieved the data for it.

6.2 Join and Leave

When a node joins or leaves the system the successor of the joining or leaving node changes its predecessor. This means that the key range of the successor node is changed. In the following we assume that the routing layer triggers an event called NewPredecessor. Upon such an event a node checks whether the

range it is responsible for has increased or has decreased. In the first case it has to copy data, in the second case it has to drop data. Similarly, a new node will fetch data once it knows the range it is responsible for.

Algorithm 15 Data Level: New Predecessor Event from the Routing Level

```
        ▷ The NewPredecessor event is triggered on the routing level, after a new
    predecessor is set due to join or periodic stabilization
 1: upon event NewPredecessor(prevpred, newpred) at n
 2:     if newpred ≠ nil then
 3:         if newpred ∈ (prevpred, n) then
 4:             trigger dropData(prevpred, newpred)
 5:         else
 6:             trigger fetchData(newpred, prevpred)
 7:         end if
 8:     else
 9:         oldpred=prevpred
10:     end if
11:     if prevpred = nil then
12:         trigger dropData(n, newpred)
13:         if oldpred ∈ (newpred, n) then
14:             trigger fetchData(newpred, oldpred)
15:         end if
16:     end if
17: end event
```

Algorithm 16 shows the copy operation. A node that has to fetch data first calculates all the other replica keys that correspond to its key range. It starts a bulk operation with these replica keys to read the data it has to store. Each node that gets a copydata request will send the values or add the requesting node as a learner to the transactions on items that are in the requested range.

Algorithm 16 Data level: Drop and fetch data

```
                                  ▷ After setting successor and predecessor and notifying the others
 1: upon event FETCHDATA(start, end) at n
 2:     ReplicaKeys:= getCorrespondingReplicaKeys(start, end),
 3:     trigger BULK(ReplicaKeys, {COPYDATA})
 4: end event

                                  ▷ After a new predecessor is set on the overlay level
 5: upon event DROPDATA(start, end) at n
 6:     DB:= DB \ DB ([start, end])
 7: end event

 8: upon event BULK([x, y], {COPYDATA})) from n' at n
 9:     foreach (r.key, tid) in ItemsInTrans do
10:         if r.key ∈ [x, y] then
11:             Learners([r.key]) = Learners([r.key]) ∪ (n', tid)
12:         end if
13:     end foreach
14:     Items := DB([x, y])
15:     Items := Items \ ItemsInTrans
16:     sendto n' : COPYDATARESPONSE(Items)
17: end event

18: upon event COPYDATARESPONSE(Replicas) from n at newnode
19:     foreach i in Replicas do
20:         myrkey := getOwnReplicaKey(i.key)
21:         MyData[(myrkey, i.key)]:= MyData([(myrkey, i.key)]) ∪ (i.val, item.ts)
22:         if | MyData [(myrkey, i.key)]| ≥ ⌊r/2⌋ + 1 then
23:             if latest(MyData[(myrkey, i.key)]) > DB(i.key, myrkey) then
24:                 DB(i.key, myrkey):=(i.val, i.ts)
25:             end if
26:         end if
27:     end foreach
28: end event
```

7 Evaluation

7.1 Analytical Evaluation of the Commit Protocol

Number of messages Figure 12 illustrates the different communication steps that are necessary for a failure free execution of the protocol. Including the initialization phase six steps are required to make a decision on the outcome of the transaction. Note that in our algorithms the leader is at the same time a transaction manager. The number of messages depends on the replication factor r in the system and the number n of items involved in the transaction. This are the number of messages sent in each step:

1. r Lookup message for TMs

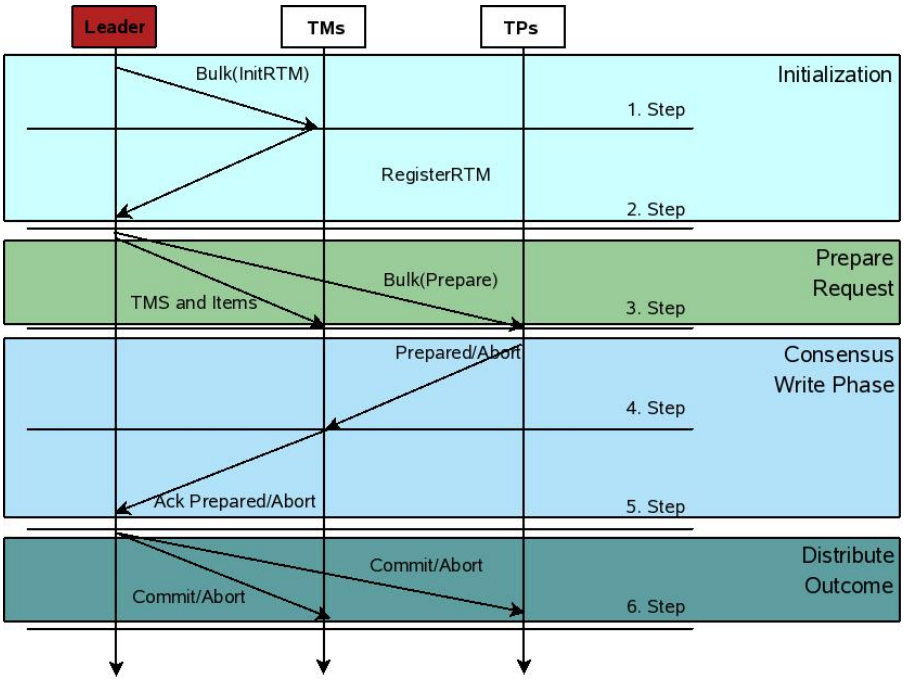


Fig. 12: The figure shows the communication steps required for the whole atomic commit protocol

2. r Registration messages from TMs
3. $n * r + r$ Prepare request to TPs and information message to TMs
4. $n * r * r$ Vote of TPs to all TMs
5. $n * r * r$ Acknowledgment for each vote
6. $n * r + r$ Decision sent to the TPs and TMs

Overall we need $(1+r)2nr+4r$ messages. The r messages from the first step use a bulk operation that is based on lookups. Similarly the $n*r$ prepare requests in the third step also use lookups. Note that the number of messages can be optimized. A TM can send all the acknowledgments for the votes it got within one message instead of sending a separate message for each vote.

Upper Timebounds

1. $O(logN)$ Lookup request to TMs
2. $O(1)$ Direct communication: Latency of slowest TM
3. $O(logN)$ Lookup request to TPs
4. $O(1)$ Direct communication: Latency of slowest conncection from a TP to a TM in a majority for an item

5. $O(1)$ Direct communication: Latency of slowest TM to leader connection
6. $O(1)$ Direct communication: Latency of slowest leader to TMs and TPs connection

8 Discussions

The algorithms do not include garbage collection issues. A transaction manager has to keep the information for a transaction long enough to be sure that each transaction participant knows the outcome. Transaction participants could acknowledge that they got the decision of the atomic commit protocol.

References

1. L. O. Alima, A. Ghodsi, and S. Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *Post-proceedings of Global Computing*, Lecture Notes in Computer Science (LNCS), pages 223–250. Springer Verlag, 2004.
2. Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
3. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.
4. A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems . In *Proceedings of the 3rd International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'05)*, volume 4125 of *Lecture Notes in Computer Science (LNCS)*, pages 74–85. Springer-Verlag, 2005.
5. Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.
6. David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.
7. Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.
8. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.
9. Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
10. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
11. Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, and Alexander Reinefeld. On consistency of data in structured overlay networks. In *CoreGRID Integration Workshop 2008*, 2008.
12. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.13 Key-based Consistency

Key-Based Consistency and Availability in Structured Overlay Networks

Tallat M. Shafaat
Royal Institute of Technology (KTH)
Stockholm, Sweden
tallat@kth.se

Monika Moser, Thorsten Schütt and Alexander Reinefeld
Zuse Institute Berlin
Berlin, Germany
{moser,schuett,ar}@zib.de

Ali Ghodsi and Seif Haridi
Swedish Institute of Computer Science
Stockholm, Sweden
{ali,seif}@sics.se

ABSTRACT

Structured Overlay Networks provide a promising platform for high performance applications since they are scalable, fault-tolerant and self-managing. Structured overlays provide lookup services that map keys to nodes that can be used as processing or storage resources. The lookups for a key may return inconsistent results. Consequently, it is nontrivial to provide consistent data services on the top of structured overlays that are built on key-based search. In this paper, we study the frequency of occurrence of inconsistent lookups. We show that the effect of lookup inconsistencies can be reduced by assigning responsibility of key intervals to nodes. We present our results as a trade-off between consistency and availability of keys. Further, since many distributed applications employ quorum techniques at their core, we analyze the probability that majority-based quorum techniques will function correctly in a structured overlay with inconsistent lookups. Our analysis shows that the probability of majority-based algorithms to function correctly despite lookup inconsistencies is high.

1. INTRODUCTION

Structured Overlay Networks, such as Chord [13] and DKS [3], form a major class of peer-to-peer systems. Structured overlays provide lookup services for Internet-scale applications, where a lookup maps a key to a node in the system. The node mapped by the lookup can then be used for data storage or processing. Distributed Hash Tables (DHTs) [3] use an overlay's lookup service to store data and provide a put/get interface for distributed systems. Since structured overlays are "best-effort", DHTs built on these overlays typically guarantee eventual consistency. In contrast, many distributed systems, such as distributed file systems and distributed databases , require stronger consistency guarantees. These systems generally rely on services such as consensus and atomic commit.

DHTs are designed to cope with high rates of churn (node joins and leaves). Due to consistent hashing [6] in a DHT, existing nodes take over key responsibilities of inaccessible nodes, and newly joined nodes take over a fraction of the responsibilities of existing nodes. Similarly, DHTs tolerate partitions in the underlying network by creating multiple independent DHTs and provide availability for all keys in each DHT.

It has been proved that it is impossible for a web service to provide the following three guarantees at the same time: consistency, availability and partition-tolerance [5]. These three properties have also been proved to be impossible to guarantee by a DHT working in an asynchronous network such as the Internet [3]. Thus, choosing to provide guarantees for two properties will violate the guarantee for the third. In this work, we focus on availability and consistency while assuming there is no network partition.

As we discuss in section 3, inconsistent data in DHTs mainly arises due to inconsistent lookups. In this paper, we study the causes and frequency of occurrences of lookup inconsistencies under different scenarios in a DHT. We discuss and evaluate techniques that can be used to decrease the effect of lookup inconsistencies. We show how decreasing the effect of lookup inconsistencies affects availability. Based on our simulation results, we give an analytical model that gives the probability under which a majority-based quorum technique works correctly. Using techniques to decrease the effect of lookup inconsistency, we show that we can achieve key consistency with high probability.

Outline: First, we define the DHT model that our work is based on in Section 2. Section 3 introduces lookup consistency and explains how it can be violated. Section 4 explains techniques that can be used to reduce consistency violation. Simulations which study the probability of a violation of lookup consistency and the affect of techniques to reduce inconsistency are presented in Section 5. In Section 6 we discuss related work. Finally, Section 7 presents the conclusion of our work.

2. BACKGROUND

Ring-based DHT: A DHT makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$. Every node in the system, has an unique identifier from the identifier space. Each node keeps a pointer *succ* to its *successor* (first node met going clockwise) and a pointer *pred* to its *predecessor* (first node

Infoscale2008, June 4–6, 2008, Vico Equense, Napoli, Italy.

met going anti-clockwise) on the ring. Ring-based DHTs also maintain additional routing pointers on top of the ring to enhance routing.

We choose Chord [13] for our analysis, which is one of the most popular ring-based overlay. Chord handles joins and failures using a protocol called *periodic stabilization*. The protocol works such that each node n should eventually point to the first node clockwise from n as *succ* and the first node anti-clockwise from n as *pred*.

Failure Detectors: DHTs provide a platform for Internet-scale systems, aimed at working on an asynchronous network. Informally, a network is asynchronous if there is no bound on message delay. Since timing assumptions can not be made in asynchronous networks, it is difficult to determine if a node has crashed or is very slow to respond. This gives rise to inaccurate suspicion of node failure. Thus, failure detectors - modules used by a node to determine if another node is alive of dead - work probabilistically.

Failure detectors are defined based on two properties: *completeness* and *accuracy* [2]. In a crash-stop process model, completeness requires the failure detector to eventually detect all crashed nodes. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not.

3. CONSISTENCY VIOLATION

In this section, we show how lookup inconsistencies may arise and discuss how lookup inconsistencies can lead to data inconsistency. Unless specified, the term consistency refers to lookup consistency, otherwise, we explicitly say data consistency. A configuration of the DHT is a set of all nodes and their pointers to neighboring nodes. A DHT evolves by either changing a pointer, or adding/removing a node.

Definition 1. *A configuration of the system is consistent if, in that configuration, lookups made for the same key from different nodes, return the same node.*

In a configuration where consistency is violated, we have inconsistent lookups *i.e.* multiple lookups for the same key may return different nodes in that configuration. Lookup consistency can be violated if some node's successor pointers do not reflect the current ring structure. Figure 1(a) illustrates such a configuration where lookups for key k can return inconsistent results. This configuration arises when, due to inaccuracy of the failure detector, $N1$ falsely suspects $N2$ and $N3$ as failed. Thus, $N1$ believes that the next (clockwise) alive node on the ring in $N4$, so it points its successor pointer to $N4$. Subsequently, a lookup for key k ending at $N1$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N2$ would return $N3$.

In the scenario depicted in figure 1(a), an update for the data stored under key k will be stored at either $N3$ or $N4$. A read for data at k will return inconsistent/old results if it reaches the node that didn't receive the update.

4. REDUCING INCONSISTENCIES

In this section, we discuss two techniques to reduce lookup inconsistencies: (1) Local responsibilities (2) Quorum-based algorithms. These techniques can be used separately, or together to get the best results.

4.1 Local Responsibilities

Definition 2. *A node n is said to be* locally responsible *for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. We call a node* globally responsible *for a key, if it is the only node in the configuration that is* locally responsible *for the key.*

The responsibility of a node changes whenever its predecessor is changed. As can be noted, a configuration is consistent if there is a globally responsible node for each key. Similarly, the responsibility for a key k is consistent if there is a node globally responsible for k.

We modify the lookup operation of a such that a lookup always returns from the locally responsible node. Thus, before returning the result of a lookup, the node checks if it is locally responsible for the key being looked up. In case the node is not locally responsible, it can either forward the request to its predecessor or ask the initiator of the lookup to retry.

Although the configuration depicted in figure 1(a) is inconsistent, yet it is consistent with respect to local responsibilities. This is because, instead of replying, the lookup for k at $N1$ will be forwarded to $N4$. Since $N4$ is not locally responsible for k, it will not reply. On the other hand, the lookup at $N2$ will be forwarded to $N3$ which will reply as it is locally responsible for k. Thus, updates and reads for data items stored under key k will give consistent results.

If a node has an incorrect predecessor pointer, the range of keys it is responsible for can overlap with another node's key range. In such a case, there will be multiple nodes responsible for the same key leading to inconsistency. This can be seen in figure 1(b). Here, both $N3$ and $N4$ are locally responsible for k. This situation may arise if $N1$ falsely suspects $N2$ while both $N2$ and $N4$ falsely suspect $N3$.

Figure 1(b) shows that the method of local responsibilities does not completely solve the problem of inconsistencies, but it decreases inconsistencies. This is mainly because without local responsibility, only one node doing inaccurate failure detections is enough to introduce inconsistencies, while multiple nodes have to do simultaneous inaccurate failure detections to introduce responsibility inconsistencies

4.1.1 Key Availability

Unfortunately, as a side effect, local responsibilities give rise to keys being unavailable.

Definition 3. *In a configuration, a key k is available if there exists a reachable node n such that n is locally responsible for k.*

Here, a node n is *reachable* in a configuration if there exists a node m such that n is the successor of m, *i.e.* $m.succ = n$ and $n \neq m$.

Availability of a key is affected by both churn and inaccurate failure detectors. When a node joins the system, it changes the responsibilities of its successor. This leads to temporary unavailability of some keys. Figure 1(c)(i) shows a configuration when $N2$ joins the overlay. $N3$ points to $N2$ as its predecessor thus making $k1$ unavailable. Key $k1$ remains unavailable until $N1$ runs periodic stabilization and sets $N1.succ = N2$ and $N2$ sets $N2.pred = N1$.

Similarly, failure of a node leads to temporary unavailability of keys until the failure is detected. Such a case is shown in figure 1(c)(ii) where $N2$ crashes. Key $k2$ remains unavailable until $N1$ detects failure of $N2$ and sets $N1.succ = N3$ and $N3$ sets $N3.pred = N1$.

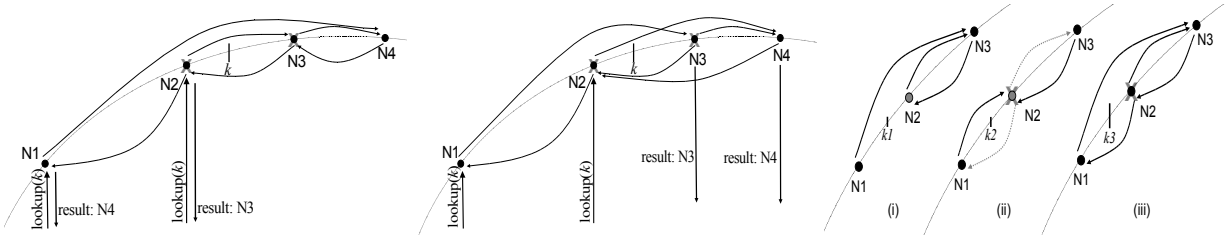


Figure 1: **(a) An inconsistent configuration. Due to imperfect failure detection, $N1$ suspects $N2$ and $N3$, thus pointing to $N4$ as successor. (b) An inconsistent configuration with respect to *local responsibilities*. $N1$ falsely suspects $N2$ while $N2$ and $N4$ falsely suspect $N3$. (c) Unavailability of key when a node (i) $N2$ joins (ii) $N2$ fails (iii) $N1$ falsely suspects $N2$ and updates its successor.**

Inaccuracy of failure detectors also leads to unavailability of keys. This occurs when a node falsely suspects its successor and removes its pointer to the suspected node. Keys for which the suspected node is responsible will temporarily become unavailable. Such a scenario is shown in figure 1(c)(iii) where $N1$ suspects $N2$ leading to unavailability of $k3$ as $N2$ becomes unreachable.

Systems that implement atomic join and graceful leaves such as DKS [3] will alleviate the case in fig. 1(c)(i), but not cases shown in fig. 1(c)(ii) and fig. 1(c)(iii).

4.2 Quorum-based Algorithms

Like most distributed systems, DHTs replicate data on different nodes to increase availability and prevent loss of data. Some examples of replication in DHTs include successor list replication [13] and key-based replication such as symmetric replication [3]. In what follows, we assume key-based replication, where an item is stored under various keys [3].

The basic idea of quorum-based algorithms is that conflicting operations acquire a sufficient number of votes from different replicas such that they have at least one intersection at one replica. Gifford introduced an algorithm for the maintenance of replicated data that uses weighted votes [4]. In our work, we consider majority-based algorithms which are a special case of quorum algorithms. The reason for choosing majority-based quorum algorithms (MBQAs) is that they are most robust and widely used form of quorum algorithms, e.g. in group membership, concurrency control and non-blocking atomic commit. In a MBQA, every replica is assigned exactly one vote and every operation has to collect at least a majority of votes (called a *majority set*). Quorum techniques can be used separately on the data-level as well to reduce data inconsistencies, yet our focus is to show how to use these techniques to reduce the affect of routing inconsistencies which will in-effect reduce data inconsistencies in DHTs. As we discuss shortly, using replicas and majorities distributes the problem of lookup inconsistency over all replicas.

4.2.1 Key-based Consistency with MBQAs

Consider a DHT with replication degree three. A data item to be stored under key k is thus stored under keys $\{k1, k2, k3\}$. Say nodes $N1$ and $N3$ are responsible for $k1$ and $k3$, while due to lookup inconsistency, two nodes $N2, N2'$ are responsible for $k2$ [1]. Any update or read for k has to operate on a majority *i.e.* two nodes in this case. Consistency in the afore-mentioned case depends on the way we choose majorities. Figure 2(a) shows a case where majorities for multiple updates overlap, thus only one update

[1]just like $N3$ and $N4$ are responsible for k in figure 1(b)

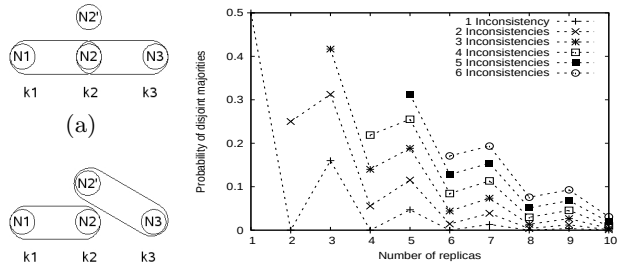


Figure 2: **(a) Example with intersecting majority sets. (b) Example with non-intersecting majority sets. (c) Probability of getting disjoint majority sets (Y-axis) for a replica set given replica degree (X-axis) and the number of lookup inconsistencies for the keys in the replica set.**

succeeds and the data remains consistent. On the other hand, figure 2(b) shows a case where the majorities do not overlap, hence updates will happen on different majority sets thus creating data inconsistent. Using MBQAs in DHTs increases the chances of consistency since even with lookup inconsistencies, multiple majorities exist that will lead to data consistency.

Probability model for disjoint majority sets: In this section, we use the counting principle to analytically derive the probability that two operations work on disjoint/non-overlapping majority sets given the system configuration is the same for the two operations. The probability of disjoint majority sets is the ratio between the number of possible disjoint majority sets and the number of all combinations of majority sets that two operations in one configuration can include. We assume that for a responsibility inconsistency in the configuration, only two nodes are responsible for the inconsistency.

$$A_{i,r} = \sum_{j=\max(m-r+i,0)}^{\min(m,i)} \sum_{k=\max(2m+i-r-2j,0)}^{\min(m,i-j)} 2^{k+j} \binom{r-i}{m-j}\binom{i}{j} \binom{r-m-i+2j}{m-k}\binom{i-j}{k} \quad (1)$$

$$T_{i,r} = \left(\sum_{j=max(m-r+i,0)}^{min(i,m)} \binom{r-i}{m-j}\binom{i}{j} 2^j \right)^2 \quad (2)$$

$$pi_r = \sum_{i=1}^{r} (1-p)^{r-i} p^i \frac{A_{i,r}}{T_{i,r}} \quad (3)$$

Consider a DHT with replication degree r (where $r > 0$), a configuration with i number of responsibility inconsistencies (where $i > 0$) and size of the smallest majority set m (where $m = \lfloor \frac{r}{2} \rfloor + 1$). $T_{i,r}$ (eq. (2)) gives the number of all possible combinations for two majority sets. Here, j is the number

of inconsistencies j included in a majority set. Since each inconsistency creates two possibilities to select a node, we multiple with 2^j.

$A_{i,r}$ (eq. (1)) gives the number of possible combinations for two disjoint majority sets $mset1$ and $mset2$. We compute $A_{i,r}$ by choosing $mset1$ and calculating every possible $mset2$ that is disjoint to $mset1$. j denotes the number of inconsistencies that are included by $mset1$. $mset2$ can share a subset of these j inconsistencies and additionally include up to $i - j$ remaining inconsistencies. The derived formula is similar to a hyper-geometric distribution.

Assuming inconsistencies are independent, pi_r calculates the probability that two subsequent operations in one configuration work on disjoint majority sets, where p is the probability of an inconsistent responsibility.

Figure 2(c) plots the probability of having disjoint majority sets pr for two operations as it is calculated by $\frac{A_{i,r}}{T_{i,r}}$. It shows how pr depends on the system's replication factor r and on the number of inconsistencies i in the replica set. An important observation is that an even replication degree reduces pr considerably. The reason for such a behaviour is that for majority-based quorums with even replication degree, any two quorums overlap over at least two replicas (say r1 and r2). Due to lookup inconsistency, even if quorums don't overlap at r1, there is a significant chance that they will overlap at r2. This reduces the probability of getting disjoint majority sets.

As lookup consistency cannot be guaranteed in a DHT, even with using local responsibilities and quorum techniques, it is impossible to ensure data consistency. However the violation of lookup consistency when using the afore-mentioned techniques is a result of a combination of very infrequent events which is evaluated in the following section.

5. EVALUATION

In this section, we evaluate the frequency of occurrence of lookup inconsistencies, overlapping responsibilities and unavailability of keys resulting from unreliable failure detectors and churn. The measure of interest is the fraction of nodes that are correct, i.e. do not contribute to inconsistencies and the percentage of keys available. The evaluations are done for a network size of 1000 nodes in a stochastic discrete event simulator in which we implemented Chord [13].

For our simulations, we employ failure detectors that are complete but not accurate. The level of reliability of a failure detector is defined by its probability of working correctly. For the graphs, the probability of a *false positive* (detect an alive node as dead) is the probability of inaccuracy of failure detectors. We implemented failure detectors in two styles: *independent* and *mutually-dependent*. For independent failure detectors, two separate nodes falsely suspect the same node as dead independently. For mutually-dependent failure detectors, if a node p is suspected dead, all nodes doing detection on p will detect p as dead with higher probability, representing a positive correlation between suspicions of different failure detectors. This may be similar to a realistic scenario where due to p or the network link to p being slow, nodes do not receive ping replies from p thus detecting it as dead. Unless specified, we use independent failure detectors. For our experiments, we varied the accuracy of the failure detectors from 95% to 100% which is a reasonable range [14].

Lookup inconsistencies: Figure 3(a) illustrates the increase in lookup inconsistencies with inaccuracy of failure detectors and churn. As the figure shows, churn does not effect lookup inconsistencies much. Even with a perfect failure detector (false positive=0), there will be a non-zero though extremely low number of lookup inconsistencies given churn. An inconsistency in such a scenario happens if multiple nodes join between two old nodes m, n (where $m.succ = n$) before m updates its successor pointer by running periodic stabilization.

Affect of local responsibilities: Next, we evaluate the effect of unreliable failure detectors and churn on responsibility consistency. The results of our simulations (omitted due to space constraints) show that responsibility consistency is also not effected by churn. Our results for unreliable failure detectors are shown in Figure 3(b) (Y2-axis).

Figure 3(b) also shows that given a lookup inconsistency, the probability of overlapping responsibilities is only 0.01. This can be seen by the scale of the lookup inconsistency (Y-axis) and overlapping responsibility (Y2-axis).

Mutually-dependent failure detectors: Figure 3(c) shows results for a scenario without churn using mutually-dependent failure detectors, where if a node n is suspected, the accuracy of all failure detectors detecting n falls to 70%. In the scenario for the simulations, we suspect 32 random nodes. Compared to independent failure detectors, mutually dependent failure detectors produce higher lookup inconsistencies.

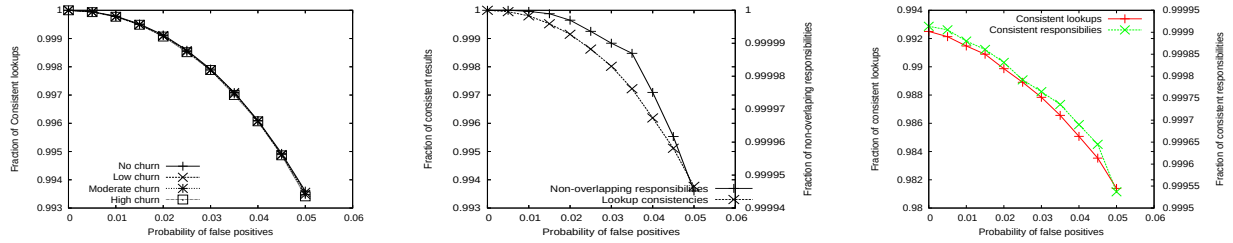
Key availability: Next, we evaluated the percentage of keys available in a system with churn and inaccurate failure detectors. Experimental studies [12] show that lifetime of nodes staying in the system ranges from tens of minutes to more than an hour. Further, experiments show that where node's mean lifetime is 1 hour, the optimal freshness threshold for periodic stabilization is about 72s [7]. Consequently, for our experiments, we choose a stabilization rate of 1 minute and varied the lifetime of nodes in tens of minutes. The results for our experiments are shown in figure 4(a), which shows that availability is effected by both inaccuracy of failure detectors and churn. Also, even with perfect failure detectors, churn results in unavailability of keys.

The affect of churn on key availability can be reduced by using atomic ring maintenance algorithms [3] [11]. These algorithms give a consistent view of the ring in the presence of node joins and leaves by transferring responsibilities of keys before a join or leave completes.

Affect of MBQA: By substituting p in Equation 3 with the simulation results using local responsibilities, we get the results illustrated in Figure 4(b). Figure 4(b) shows the probability for two non-disjoint majority sets in a certain configuration of a DHT depending on the probability that a failure detector makes false positives. Reflecting the results of figure 2(c), the probability that two majority sets are disjoint in a system with an even number of replicas is almost zero. However, in such as system, lesser number of unavailable replicas can be tolerated for an operation to succeed.

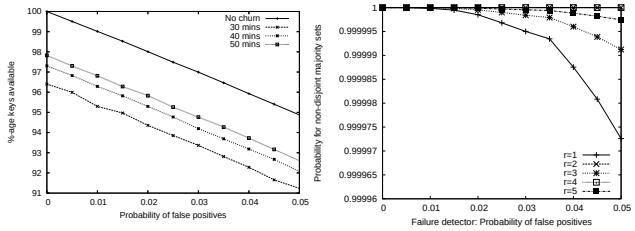
6. RELATED WORK

An important design goal for distributed systems is to provide data consistency. Since DHTs are aimed to work over asynchornous networks with high rate of churn, providing consistency in DHTs becomes an intersting and nontrivial problem. The problem at hand can be attacked on two levels: *routing level* and *data level*. We focus on the routing



(a) (b) (c)

Figure 3: **(a) Evaluation of lookup inconsistency under churn, with only node joins. (b) Comparison of lookup inconsistency (Y-axis) and overlapping responsibilities (Y2-axis). (c) Comparison of lookup inconsistency (Y-axis) and overlapping responsibilities (Y2-axis) with mutually dependent failure detectors.**



(a) (b)

Figure 4: **(a) Evaluation of keys availability under churn generated by different lifetime of nodes. (b) Probability of disjoint majority sets for two majority based operations in the same configuration.**

level by providing techniques to reduce the affect of lookup inconsistencies. Solutions on the data level (e.g. [9]) might have constraints or depend on the application and semantics of data operations.

Atomic ring maintenance algorithms [3, 8, 11] provide lookup consistency under joins and leaves, ignoring failures and inaccurate failure detectors. As we have shown, the main contributors to lookup inconsistency are inaccurate failure detectors, which is the focus of our work.

There has been work done on studying lookup inconsistencies under churn. Rhea *et. al.* [10] have explored lookup inconsistencies for Chord. Their work overlooks the fact that imperfect failure detectors mainly cause inconsistent lookups.

Bhagwan *et. al.* [1] attack the problem of availability in peer-to-peer systems. Contrary to our work, they focus on availability of hosts and thus data stored at the hosts. Since we are working on the routing level, we focus on availability of keys and thus nodes responsible for keys.

7. CONCLUSIONS [2]

We studied the frequency of lookup inconsistencies and found that its main cause is inaccurate failure detectors. Hence, choice of a failure detection algorithm is of crucial importance in DHTs.

While effects of lookup inconsistencies can be reduced by using local responsibilities, we show that using responsibility of keys may affect availability of keys. This is a trade-off between availability and consistency. Many data dependent applications may prefer unavailability to inconsistency.

We show that using quorum-based techniques amongst replicas of data items further reduce lookup inconsistencies. Since majority-based quorum techniques require a majority of the replicas to make progress, these algorithms may still make progress even with unavailability of some keys/nodes. Thus, using a combination of local responsibilities and quorum techniques is attractive in scalable applications.

Due to the dynamics and decentralization of DHTs, it is difficult to build abstractions with stronger consistency guarantees on top of DHTs. We propose using techniques on the routing level to decrease data inconsistencies. These techniques can be used with techniques at the data level to get best results. Our results show that it is reasonable to build reliable services on top of a DHT. In our future work, we plan to evaluate an implementation of a transactional storage service on top of a DHT using routing-level techniques described in this paper.

[2] **THIS WORK HAS BEEN FUNDED BY THE EUROPEAN UNION IN THE SELFMAN PROJECT (CONTRACT 034084) AND COREGRID NETWORK OF EXCELLENCE (CONTRACT 004265).**

8. REFERENCES

[1] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.

[3] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH — Royal Institute of Technology, Sweden, Dec. 2006.

[4] D. K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[5] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[6] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, 1997.

[7] J. Li. *Routing tradeoffs in dynamic peer-to-peer networks*. PhD thesis, MIT — Massachusetts Institute of Technology, Nov. 2005.

[8] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasudaram. Guaranteeing correctness and availability in p2p range indices. In *Proceedings of 2005 ACM SIGMOD*, pages 323–334, 2005.

[9] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in distributed hash tables. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 295–305, London, UK, 2002. Springer-Verlag.

[10] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. Technical report, EECS Department, University of California, 2003.

[11] J. Risson, K. Robinson, and T. Moors. Fault tolerant active rings for structured peer-to-peer overlays. *lcn*, 0:18–25, 2005.

[12] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *In Proc. of MMCN*, 2002.

[13] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM*, 2001.

[14] S. Zhuang, D. Geels, I. Stoica, and R. Katz. On failure detection algorithms in overlay networks. In *Proc. of INFOCOM*, 2005.

A.14 Consistency of Data in Structured Overlays

ON CONSISTENCY OF DATA IN STRUCTURED OVERLAY NETWORKS *

Tallat M. Shafaat†, Monika Moser‡, Ali Ghodsi†, Thorsten Schütt‡, Seif Haridi†, Alexander Reinefeld‡

Abstract Data consistency can be violated in Distributed Hash Tables (DHTs) due to inconsistent lookups. In this paper, we identify the events leading to inconsistent lookups and inconsistent responsibilities for a key. We find the inaccuracy of failure detectors as the main reason for inconsistencies. By simulations with inaccurate failure detectors, we study the probability of reaching a system configuration which may lead to inconsistent data. We analyze majority-based algorithms for operations on replicated data. To ensure that concurrent operations do not violate consistency, they have to use non-disjoint sets of replicas. We analytically derive the probability of concurrent operations including disjoint replica sets. By combining the simulation and analytical results, we show that the probability for a violation of data consistency is negligibly low for majority-based algorithms in DHTs.

1. Introduction

Peer-to-peer systems have gained tremendous popularity in recent years due to characteristics of scalability, fault-tolerance and self-management. Structured Overlay Networks (SONs) are a major class of these peer-to-peer system, examples of SONs include Chord [3], Chord# [6], Pastry [10] and DKS [7]. SONs provide lookup services for Internet-scale applications. Distributed Hash Tables (DHTs) use a SON's lookup service to provide a put/get interface for distributed systems with eventual consistency guarantees [11]. In contrast, many distributed systems require stronger consistency guarantees, relying on

*This research work is carried out under the SELFMAN project funded by the European Commission and the Network of Excellence CoreGRID funded by the European Commission.

†Royal Institute of Technology (KTH), $\{tallat, haridi\}(at)kth.se$
‡Zuse Institute Berlin (ZIB), $\{moser, schuett, ar\}(at)zib.de$
┤ Swedish Institute of Computer Science (SICS), $ali(at)sics.se$

services such as consensus [12] and atomic commits [13]. These services employ quorum techniques at their core to guarantee consistency.

Quorum based algorithms are not well-suited for DHTs. Quorum based techniques provide consistency guarantees as long as quorums overlap *i.e.* are never disjoint. On the contrary, the number of replicas of an item are not constant in a DHT. Hence, due to the extra replicas in a DHT, two quorums might not intersect, leading to inconsistent results.

Like most distributed systems, DHTs replicate a data item on different nodes in the SON to avoid loosing data. In DHTs, the number of replicas may become greater than the replication degree for two reasons: *lookup inconsistencies* and *partitions*. Consider a DHT with replication degree three and an item replicated on nodes $N1, N2$ and $N3$. Due to lookup inconsistencies in the underlying SON[1], another node $N4$ might think that it is also responsible for the data item and will replicate the item. In such a case, a majority-based quorum technique [16] will result in inconsistent data as there are disjoint majority sets *e.g.* $\{N1, N2\}$ and $\{N3, N4\}$.

DHTs tolerate partitions in the underlying network by creating multiple independent DHTs. Due to consistent hashing [14], new nodes take responsibilities of inaccessible nodes and replicate data items. Thus, in the aforementioned case, if a partition occurs such that $N1, N2$ (partition P1) are separated from $N3$ (partition P2), owing to consistent hashing, replacement node $N3'$ will replicate the item in P1, and replacement nodes $N1'$ and $N2'$ will replicate the item in P2. This will result in the two partitions to have disjoint majority sets which will lead to data inconsistency.

It has been proved that it is impossible for a web service to provide the following three guarantees at the same time: consistency, availability and partition-tolerance [9]. These three properties have also been proved to be impossible to guarantee by a DHT working in an asynchronous network such as the Internet [7]. Thus, choosing to provide guarantees for two properties will violate the guarantee for the third. Since lookups are always allowed in DHTs, this implies DHTs are always available, thus consistency cannot be guaranteed.

In this paper, we study the causes and frequency of occurrence of lookup inconsistency under different scenarios in a DHT. We focus solely on lookup inconsistency leaving scenarios where complete partitions can happen, resulting in creation of multiple separate DHTs. We discuss and evaluate techniques that can be used to decrease the effect of lookup inconsistencies. Based on our simulation results while considering lookup inconsistencies to be the only reason for creation of extra replicas, we give an analytical model that gives the probability under which a majority-based quorum technique works correctly.

[1] Informally, a lookup inconsistency means multiple nodes believe to be responsible for the same identifier. The term will be discussed in detail later.

Using techniques to decrease the effect of lookup inconsistency, we show that the probability of a quorum technique to produce consistent results is very high.

2. Background

Basics of a Ring-based SON. A SON makes use of an *identifier space*, which for our purposes is a range of integers from 0 to $N-1$, where N is the length of the identifier space and is a large, fixed and globally known integer. For ring-based SONs, this identifier space is perceived as a ring by arranging the integers in ascending order and wrapping around at $N-1$.

Every node in the system has a unique identifier drawn from the identifier space. Each node p has a pointer, *succ*, to its *successor*, which is the node immediately succeeding p, going in clockwise direction on the ring starting at p. Similarly, each node q has a pointer, *pred*, to its *predecessor*, which is the node immediately preceding q, going in anti-clockwise direction on the ring starting at q. To enhance routing performance, SONs also maintain additional routing pointers.

Handling Joins and Failures. Apart from *succ* and *pred* pointers, each node p also maintains a *successor-list* consisting of p's c immediate successors, where c is typically set to $\log_2(n)$, n being the network size.

Chord [3] handles joins and failures using a protocol called *periodic stabilization*. Each node p periodically checks to see if its *succ* and *pred* are alive. If *succ* is found to be dead, it is replaced by the closest alive successor in the successor-list. If *pred* is found to be dead, p sets $pred := nil$.

Joins are also handled periodically. A joining node makes a lookup to find its successor s on the ring, and sets $succ := s$. Each node periodically asks for its successor's *pred* pointer, and updates its *succ* pointer if it gets a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*. Hence, any joining node is eventually properly incorporated into the ring.

Failure Detectors. SONs provide a platform for Internet-scale systems, aimed at working on an asynchronous network. Informally, a network is asynchronous if there is no bound on message delay. Thus, no timing assumptions can be made in such a system. Due to the absence of timing restrictions in an asynchronous model, it is difficult to determine if a node has actually crashed or is very slow to respond. This gives rise to wrong suspicions of failure of nodes.

Failure detectors are modules used by a node to determine if its neighbors are alive or dead. Since we are working in an asynchronous model, a failure

detector can only provide probabilistic results about the failure of a node. Thus, we have failure detectors working probabilistically.

Failure detectors are defined based on two properties: *Completeness* and *Accuracy* [5]. In a crash-stop model, completeness is the property that requires a failure detector to eventually detect as dead a node which has actually crashed. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not. A perfect failure detector is accurate all the times, while the accuracy of an unreliable failure detector is defined by its probability of working correctly.

For our work, we use a failure detector similar to the baseline algorithm used by Zhuang et. al [4]. A node sends a ping to its neighbors at regular intervals. If it receives an acknowledgment within a timeout, the neighbor is considered alive. Not receiving an acknowledgment within the timeout implies the neighbor has crashed. The timeout is chosen to be much higher than the round-trip time between the two nodes.

3. Lookup and Responsibility Consistency

Data consistency is based on lookup consistency and responsibility consistency in the routing layer. We define these concepts and explain how a violation of these happens. The notion of a SON's configuration comprises the set of all nodes in the system and their pointers to neighboring nodes. A SON evolves by either changing a pointer, or adding/removing a node.

Lookup Consistency. *A lookup on a key is consistent, if lookups made for this key in a configuration from different nodes, return exactly the same node.*

Lookup consistency can be violated if some node's successor pointer does not reflect the current ring structure. Figure 1a illustrates a scenario, where lookups for key k can return inconsistent results. This configuration may occur if node $N1$ falsely suspected $N2$ as failed, while at the same time $N2$ falsely suspected $N3$ as failed. A lookup for key k ending at $N2$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N1$ would return $N3$.

Responsibility. *A node n is said to be* locally responsible *for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. We call a node* globally responsible *for a key, if it is the only node in the system that is* locally responsible *for it.*

The responsibility of a node changes whenever its predecessor is changed. If a node has an incorrect predecessor pointer, it might have an overlapping range of keys with another node. However, to have concurrent operations on an item i with key k working on two different physical copies i' and i'', the concurrent lookups should be inconsistent, and there should be an overlap of responsibility

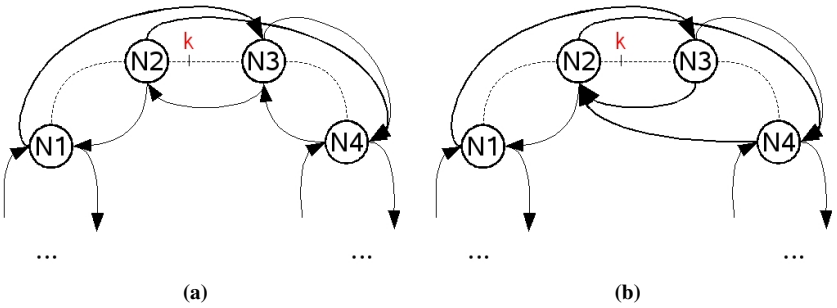


Figure 1: (a) *Lookup inconsistency* caused by wrong successor pointers. (b) *Responsibility inconsistency* caused by wrong successors and backlinks resulting in overlapping responsibilities.

for key k. Thus the following definition on responsibility consistency combines lookup consistency and global responsibility.

Responsibility Consistency. *The responsibility for a key is consistent if there is a globally responsible node for that key in the configuration.*

A situation where responsibility consistency for key k is violated is shown in Figure 1b. Here, lookup consistency for k cannot be guaranteed and both nodes, $N3$ and $N4$, are locally responsible for k. However, in Figure 1a, there is only one node $N3$ that is globally responsible despite lookup inconsistency. At node $N4$ the item is simply unavailable. The situation depicted in Figure 1b arises as the situation in Figure 1a with an additional wrong suspicion of node $N4$ about its predecessor $N3$.

As lookup consistency and responsibility consistency cannot be guaranteed in a SON it is impossible to ensure data consistency. However the violation of lookup consistency and responsibility consistency is a result of a combination of very infrequent events. In the following section we present simulation results that measure the probability of lookup inconsistencies and the probability of having an inconsistent responsibility, which turns out to be almost negligible.

4. Evaluation

In this section, we evaluate how often lookup inconsistencies and overlapping responsibilities occur. For our experiments, the measure of interest is the fraction of nodes that are correct, i.e. do not contribute to inconsistencies. The evaluations were done in a stochastic discrete event simulator in which we implemented Chord [3]. The simulator uses an exponential distribution for the inter-arrival time between events (joins and failures). To make the simulations scale, the simulator is not packet-level. The time to send a message from one node to another is an exponentially distributed random variable.

For our simulations, the level of unreliability of a failure detector is defined by its probability of working correctly. For the graphs, the probability of a *false positive* [2] is the probability of inaccuracy of failure detectors. Thus, a failure detector with a probability of false-positives equal to zero is a perfect failure detector.

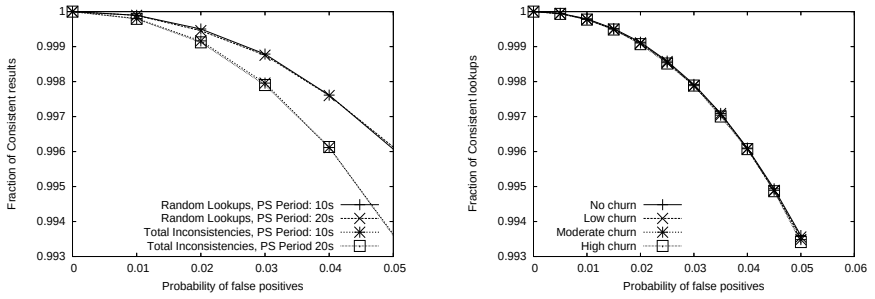
In our experiments, we implemented failure detectors in two styles, *independent* and *mutually-dependent* failure detectors. For independent failure detectors, two separate nodes falsely suspect the same node as dead independently. Thus, if a node n is a neighbor of both m and o, the probability of m detecting n as dead is independent of the probability of o detecting n as dead. For mutually-dependent failure detectors, if a node n is suspected dead, all nodes doing detection on n will detect n as dead with higher probability. This may be similar to a realistic scenario where due to n or the network link to n being slow, nodes do not receive ping replies from n thus detecting it as dead. In the afore-mentioned case, if n is suspected, both m and o will detect it dead with higher probability than the probability of false-positive. Henceforth, we use independent failure detectors unless specified.

For our simulations, we first evaluate lookup inconsistencies for different degrees of false-positives. Next, we evaluate overlapping responsibilities in a system with and without churn. Furthermore, we compare lookup inconsistency and overlapping responsibilities. Finally, we present the results with mutually dependent failure detectors.

Our simulation scenario has the following structure: Initially, we successively joined nodes into the system until we had a network with 1024 nodes. We then started to gather statistics by regularly taking snapshots (earlier defined as a configuration) of the system. In each snapshot, we counted the number of correct nodes i.e. do not contribute to lookup inconsistency and overlapping responsibilities. For the experiments with churn, we introduced node joins and failures between the snapshots. We varied the accuracy of the failure detectors from 95% to 100%, where 100% means a perfect failure detector. This range seems reasonable, since failure detectors deployed on the Internet are usually accurate 98% of the time [4]. The results presented in the graphs are averages of 1800 snapshots and 30 different seeds.

Lookup Inconsistency. Figure 2a illustrates the increasing lookup inconsistency when the failure detector becomes inaccurate. The plot denoted 'Total Inconsistencies' shows the maximum over all possible lookup inconsistencies in a snapshot, whereas 'Random Lookups' shows the number of consistent lookups when – for each snapshot – lookups are made for 20 random keys, where each lookup is made from 10 randomly chosen nodes. If all lookups for

[2] detect an alive node as dead



(a) **(b)**

Figure 2: (a) Evaluation of lookup inconsistency. (b) Evaluation of lookup inconsistency under churn with only node joins.

the same key result in the same node, the lookup is counted as consistent. As can be seen, changing the periodic stabilization rate does not effect the lookup inconsistency in this case. This is due to the fact that there is no churn in the system.

Next, we evaluated lookup inconsistencies in the presence of churn. We varied the churn rate with respect to the periodic stabilization (PS) rate of Chord. For our experiments, we defined churn as node session times, to be in tens of minutes [15]. Short session times produce 'high churn' while long session times produce 'low churn'. Figure 2b shows the results for our experiments. The Y-axis gives a count of the number of lookup inconsistencies per snapshot. As expected, churn does not effect lookup inconsistencies much. Though, even with a perfect failure detector (probability of false positive=0), there will be a non-zero though extremely low number of lookup inconsistencies given churn ($2.79\text{x}10^{-7}$ for a high churn system). The reason is that an inconsistency in such a scenario only happens if multiple nodes join between two old nodes m, n (where $m.succ = n$) before m updates its successor pointer by running PS.

This effect of churn is due to node joins on lookup inconsistency can be reduced to zero if we allow lookups to be generated only from nodes that are fully in the system. A node is said to be *fully in the system* after it is accessible from any node that is already in the system. Once a node is fully in the system, it is considered to be in the system until it crashes. We define the first node which creates the ring as fully in the system.

Responsibility Inconsistency. Next, we evaluate the effect of unreliable failure detectors and churn on the responsibility consistency. The results of our simulations are presented in Figure 3a which shows that responsibility consistency is not effected by churn. Figure 3b shows that even with a lookup inconsistency, the chances of overlapping responsibilities are decreased

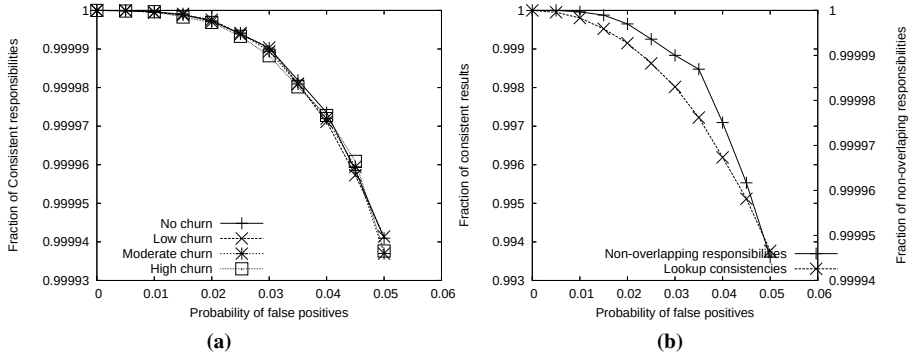


Figure 3: (a) Evaluation of overlapping responsibilities under churn with only node joins. (b) Comparison of lookup inconsistency and overlapping responsibilities. Lookup inconsistency is plotted against Y-axis while overlapping responsibilities is plotted against Y2-axis.

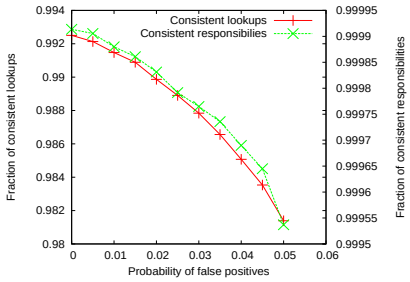


Figure 4: Evaluation of lookup inconsistency and overlapping responsibilities with mutually dependent failure detectors. Lookup inconsistency is plotted against Y-axis while overlapping responsibilities is plotted against Y2-axis.

roughly 100 times. This can be seen by the scale of the lookup inconsistency (Y-axis) and overlapping responsibility (Y2-axis).

Figure 4 shows results for a scenario without churn using mutually dependent failure detectors, where if a node n is suspected, the probability of nodes doing accurate detection on n drops to 0.7. In the scenario for the simulations, we suspect 32 random nodes. Compared to independent failure detectors, mutually dependent failure detectors produce higher lookup inconsistencies, but still low.

5. Data Consistency with Majority-Based Algorithms

To prevent loss of items stored in a SON, items are replicated on a set of nodes. The set of nodes that are responsible for the replicas, is determined by some replication scheme. Here we consider replication schemes that have a fixed replication factor r. An example for such a replication scheme is the DKS symmetric replication [2], where a globally known function determines

the set of keys under which replicas for an item are stored. The set of replicas for an item is called *replica set*.

In dynamic environments, some replicas might be temporary unavailable. However operations on an item should be able to succeed if they can access a subset of the replica set. Majority based algorithms require that at least a majority of replicas are available and tolerate the unavailability of the rest. Thus they are well suited for a SON. We refer to a set with a majority of replicas as a *majority set*. As each write operation includes such a majority set, two concurrent write operations have at least one replica in common, such that a conflict can be detected. It is crucial that the number of replicas in the system is never increased, otherwise one cannot guarantee that concurrent operations work on *non-disjoint majority sets*. However responsibility inconsistencies temporarily lead to an increase in the number of replicas. In the following, we analyze the probability of two concurrent operations working on *disjoint majority sets* given i inconsistencies in the replica set, to which we refer as *inconsistent replicas*.

Probability for Disjoint Majority Sets. In this section we model the probability that two operations work on disjoint majority sets in a given configuration. We assume that each responsibility inconsistency involves at most two nodes. More than two concurrent operations working on disjoint majority sets are not considered as the probability for it is considered as negligibly small.

The size of the smallest majority set is defined by $m = \lfloor \frac{r}{2} \rfloor + 1$. $T_{i,r}$, as shown in Equation (1), counts the number of all possible combinations for two majority sets, given $i > 0$ inconsistent replicas and the replication factor r. The formula takes into account the number of inconsistency j that are included in the majority sets. Each included inconsistency involves two possibilities to select a node that stores the replica.

$A_{i,r}$, in Equation (2), calculates the number of possible combinations for two disjoint majority sets, $m1$ and $m2$, given $i > 0$ inconsistent replicas in the replica set and a replication factor r. We compute $A_{i,r}$ by choosing set $m1$ and count all possible sets $m2$ that are disjoint to $m1$. Part a of $A_{i,r}$ counts all possibilities to choose $m1$, such that at least one inconsistency is included. Again, j denotes the number of included inconsistencies. The second majority set $m2$ shares k of the j inconsistencies (part b). For the remaining replicas of $m2$ we have to consider how many inconsistencies l it will include from those that are left (part c).

$$T_{i,r} = \Big(\sum_{j=max(m-(r-i),0)}^{min(i,m)} \binom{r-i}{m-j}\binom{i}{j} * 2^j\Big)^2 \quad (1)$$

$$A_{i,r} = \sum_{j=lb_j}^{ub_j}\sum_{k=lb_k}^{j}\sum_{l=lb_l}^{ub_l} \overbrace{\binom{r-i}{m-j}\binom{i}{j} * 2^j}^{a} * \underbrace{\binom{j}{k}}_{b} * \underbrace{\binom{(r-m)-(i-j)}{m-k-l}\binom{i-j}{l} * 2^l}_{c} \quad (2)$$

where (ub short for upper bound, lb short for lower bound)

$$lb_j = max(1, m-(r-i))$$
$$ub_j = min(i, m)$$
$$lb_k = max(1, m-(r-m))$$
$$lb_l = max(0, (m-k)-((r-m)-(i-j)))$$
$$ub_l = min(i-j, m-k)$$

$$pi_r = \sum_{i=1}^{r}(1-p)^{r-i} * p^i * \frac{A_{i,r}}{T_{i,r}} \quad (3)$$

pi_r calculates the overall probability in the system that two concurrent operations on one item operate on disjoint majority sets, where p is the probability of an inconsistency at a node as measured in our simulations.

Table 5 contains the probabilities for disjoint majority sets of two concurrent operations as calculated by $\frac{A_{i,r}}{T_{i,r}}$. As with an even replication factor the minimum number of replicas in common for two majority sets is two in contrast to one for an odd number of replicas, the probability for disjoint majorities is lower for an even number of replicas. In Figure 6 the results of the simulations are combined with Equation 3, where p denotes the simulated probability for an inconsistency. Depending on the failure detector accuracy it plots the probability to get non-disjoint majority sets. An even replication factor increases the probability of having non-disjoint majority sets, however less unavailable replicas can be tolerated.

6. Related Work

DHTs have been the subject of much research in recent years, with substantial amount of work on resilience of overlays to churn. While these studies show that overlays tolerate failures, they also show how lookups are effected by churn.

Rhea *et. al.* [1] have explored lookup inconsistencies for a real implementation under churn. Their approach differs from ours as they define a lookup to be consistent if a majority of nodes concurrently making a lookup for the same key get the same result. For our work, we require all results of making

r	$i=1$	$i=2$	$i=3$	$i=4$
1	0.5			
2	0	0.25		
3	0.16	0.31	0.42	
4	0	0.05	0.14	0.22
5	0.05	0.11	0.19	0.26

Figure 5: Probability for disjoint majority sets depending on the replication factor r and the number of inconsistencies i in the replica set.

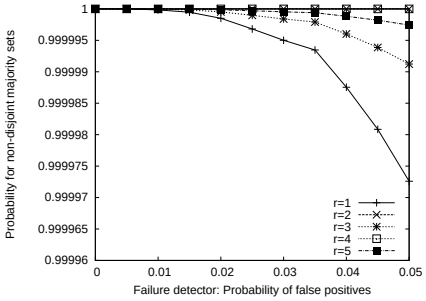


Figure 6: Probability for non-disjoint majority sets in a SON, depending on the accuracy of the failure detector

the lookup for the key to be the same. Furthermore, our work is extended to responsibility consistency. In their work, Rhea *et. al.* also study lookup inconsistency in an implementation of Pastry [10] called FreePastry [18], while we experiment with Chord.

Liben-Nowell *et. al.* [17] study the evolution of Chord under churn. Their study is based on a fail-stop model *i.e.* they assume perfect failure detection and reliable message delivery. Consequently, they ignore "false suspicions of failure", which is the main topic of our study as we observe that imperfect failure detectors are the main source of lookup inconsistencies.

Zhuang *et. al.* [4] studied various failure detection algorithms in Overlay Networks. They also use the same approach as Rhea *et. al.* [1] to define inconsistencies, which differs from our work.

7. Conclusion

This paper presents an evaluation of consistency in SONs. Data consistency cannot be achieved if responsibility consistency is violated. We describe why it is impossible to guarantee responsibility consistency in SONs. By simulating a Chord SON, we show that the probability of violating responsibility consistency is negligibly low.

We analytically derive the probability that majority-based operations are working on non-disjoint majority sets given an inconsistent responsibility for at least one replica. Operations that work on disjoint majority sets lead to inconsistent data. By combining the results from simulations and analysis, we show that the probability for getting inconsistent data when using majority based algorithms is significantly low. Furthermore, we conclude that since the accuracy of the failure detector greatly influences lookup and responsibility consistency, significant attention should be paid to the failure detection algorithm.

References

[1] S. Rhea, D. Geels, T. Roscoe and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of USENIX Annual Technical Conference*, 2004 Berkeley

[2] A. Ghodsi, L. Onana Alima and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *DBISP2P*, 2005, Trondheim, Norway

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. newblock *IEEE/ACM Transactions on Networking (TON)*, 11(1):17.32, 2003.

[4] S.Q. Zhuang, D. Geels, I. Stoica, R.H. Katz. On Failure Detection Algorithms in Overlay Networks. In *Proceedings of INFOCOM'05*, Miami, 2005

[5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43:2, 1996

[6] T. Schütt, F. Schintke and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *Proceedings of GP2PC'06*, 2006

[7] A. Ghodsi. Distributed *k*-ary System: Algorithms for Distributed Hash Tables. *PhD Dissertation*, KTH—Royal Institute of Technology Oct, 2006

[8] M. Moser, S. Haridi. Atomic Commitment in a Transactional DHT. In *Proceedings of the CoreGRID Symposium*, 2007

[9] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *SIGACT News*, 2002

[10] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of MIDDLEWARE.01*, volume 2218 of Lecture Notes in Computer Science (LNCS), Germany, 2001

[11] F. Dabek. A Distributed Hash Table. *Doctoral Dissertation,* MIT — Massachusetts Institute of Technology, 2005

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM Press, 1998, 16, 133-169

[13] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. Atomic Transactions. Morgan Kaufmann Publishers, 1994

[14] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, El Paso, 1997

[15] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of SOSP*, 2003.

[16] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of SOSP '79*, New York, USA, 1979

[17] D. Liben-Nowell, H. Balakrishnan, D. Karger Analysis of the Evolution of Peer-to-Peer Systems In *Proceedings of PODC '02*, USA, 2002

[18] Freepastry. `http://freepastry.rice.edu/`

A.15 Handling Network Partitions and Mergers in Structured Overlay Networks

Handling Network Partitions and Mergers in Structured Overlay Networks *

Tallat M. Shafaat† Ali Ghodsi‡ Seif Haridi†

†Royal Institute of Technology (KTH), ‡Swedish Institute of Computer Science (SICS)
{tallat,haridi}(at)kth.se ali(at)sics.se

Abstract

Structured overlay networks form a major class of peer-to-peer systems, which are touted for their abilities to scale, tolerate failures, and self-manage. Any long-lived Internet-scale distributed system is destined to face network partitions. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has hardly been studied in the context of structured peer-to-peer systems. These systems have mainly been studied under churn (frequent joins/failures), which as a side effect solves the problem of network partitions, as it is similar to massive node failures. Yet, the crucial aspect of network mergers has been ignored. In fact, it has been claimed that ring-based structured overlay networks, which constitute the majority of the structured overlays, are intrinsically ill-suited for merging rings. In this paper, we present an algorithm for merging multiple similar ring-based overlays when the underlying network merges. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something widely believed to be difficult or impossible. We evaluate the algorithm for various scenarios and show that even when falsely detecting a merger, the algorithm quickly terminates and does not clutter the network with many messages. The algorithm is flexible as the tradeoff between message complexity and time complexity can be adjusted by a parameter.

1 Introduction

Structured Overlay Networks (SONs)—such as Chord [29], Pastry [26], and SkipNet [13]—are touted for their ability to provide scalability, fault-tolerance, and self-management, making them well-suited for Internet-scale distributed applications. Such Internet-scale systems will always come across network partitions, especially if the system is long-lived. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has, with few exceptions, been ignored in the context of structured overlays. This is peculiar, as the importance of the problem has long been known in other problem domains, such as those of distributed databases [5] and distributed file systems [30].

It is our firm belief that a crucial requirement for practical SONs is that they should be able to deal with network partitions and mergers. As we show in Section 2, most SONs cope with network partitions, but not with network mergers. We believe that this is because a network partition, as seen from the perspective a single node, is identical to massive node failures. Since SONs have been designed to cope with churn, they can self-manage in the presence of such partitions. However, most SONs cannot cope with network mergers.

In fact, it has been claimed that ring-based structured overlays, which constitute the absolute majority of the SONs, are inherently a poor fit for dealing with network mergers. Datta *et al.* [4] focus on the merging of multiple SONs after a network partition ceases (network merger). They argue that ring-based SONs "cannot function at all until the whole merge process is complete". Birman [2] argues that ring-based SONs are inherently ill-suited for dealing with network partitions.

The merging of SONs gives rise to problems on two different levels: *routing level* and *data level*. The routing level is concerned with healing of the routing information after a partition merger.

The data level is concerned with the consistency of the data items stored in the SONs. The solutions to this problem might depend on the application and on the semantics of the data operations, e.g. immutable key/value pairs or monotonically increasing values. It is also known that it is impossible to achieve strong (atomic) data consistency, availability[1], and partition tolerance in SONs [11, 3, 9].

*This research has been funded by the European Project SELFMAN, VINNOVA 2005-02512 TRUST-DIS, and SICS Center for Networked Systems (CNS).

[1]By availability we mean that a get/put operation should eventually complete.

We focus on the problem of dealing with partition mergers at the routing level. Given a solution to the problem at the routing level, it is generally known how to achieve weaker types of data consistency, such as eventual consistency [30, 6].

In this paper, we present an algorithm for merging any number of similar structured overlays. We will limit ourselves to ring-based overlays, since they constitute the majority of the SONs. It is desirable that a solution to the problem of merging rings takes minimum amount of time to complete (time complexity). At the same time, it is desirable that the solution has a minimal bandwidth consumption (message and bit complexity). These two goals are conflicting, as shown by the following two extreme cases. On the one hand, it is possible to construct an algorithm that completes in minimal time by having all the nodes repeatedly spreading all their routing information to every other node through an overlay broadcast [7, 10, 9]. On the other hand, it is possible to construct an algorithm which tries to minimize the bandwidth consumption by passing a "merging" token around each of the rings. Hence, it is desirable to find an algorithm which strikes a balance between time, bit, and message complexity.

The contribution of this paper is a ring merging algorithm, which allows the system designer to adjust, through a *fanout* parameter, the tradeoff between message complexity and time complexity. Through experimental evaluation, we show typical fanout values for which our algorithm completes quickly, while keeping the bandwidth consumption at an acceptable level. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something widely believed to be difficult [2] or impossible [4]. We verify that the algorithm works efficiently even if only a single node detects the partition merger. We show that even with large rings with thousands of nodes, our solution is lean as it avoids positive-feedback cycles and, hence, avoids congesting the network.

Outline Section 2 serves as a background by motivating and defining our choice of ring-based SONs. Section 3 introduces the *simple ring unification algorithm*, as well as the *gossip-based ring unification algorithm*. Since the latter algorithm builds on the previous, we hope that this has a didactic value. Thereafter, Section 4 evaluates different aspects of the algorithms in various scenarios. Section 5 presents related work. Finally, Section 6 concludes.

2 Background

The rest of the paper focuses on ring-based structured overlay networks. Next, we motivate this choice, and thereafter briefly define ring-based SONs. Finally, we show how Chord deals with network partitions and failures.

Motivation for the Ring Geometry The reason for confining ourselves to ring-based SONs is twofold. First, ring-based SONs constitute a majority of the SONs, including Chord [29], Pastry [26], SkipNet [13], DKS [9], Koorde [16], Viceroy [23], Mercury [1], Symphony [24], Epi-Chord [17], and Accordion [18]. Second, Gummadi *et al.* [12] diligently compared the geometries of different SONs, and showed that the ring geometry is the one most resilient to failures, while it is just as good as the other geometries when it comes to proximity.

Our results apply to all ring-based SONs. Nevertheless, we assume a SON similar to Chord [29] to simplify the understanding of our algorithms.

A Model of a Ring-based SON A SON makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has a unique identifier from the identifier space. Node identifiers are typically assumed to be uniformly distributed on the identifier space. Each node keeps a pointer, $succ$, to its *successor* on the ring. The successor of a node with identifier p is the first node found going in clockwise direction on the ring starting at p. Similarly, every node also has a pointer, $pred$, to its *predecessor* on the ring. The predecessor of a node with identifier q is the first node met going in anti-clockwise direction on the ring starting at q. A *successor-list* is also maintained at every node r, which consists of r's c immediate successors, where c is typically set to $\log_2(\mathcal{N})$.

Ring-based SONs also maintain additional routing pointers on top of the ring to enhance routing. To keep things concrete, assume that these are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 < i < \log_2(\mathcal{N})$. Our results can easily be adapted to other schemes for placing these additional pointers.

Dealing with Partitions and Failures in Chord Chord handles joins and leaves using a protocol called *periodic stabilization*. Leaves are handled by having each node periodically check whether $pred$ is alive, and setting $pred := nil$ if it is found dead. Moreover, each node periodically checks to see if $succ$ is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Joins are also handled periodically. A joining node makes a lookup to find its successor s on the ring, and sets $succ := s$. Each node periodically asks for its successor's $pred$ pointer, and updates $succ$ if it finds a closer successor. Thereafter, the node notifies its current $succ$ about its own existence, such that the $succ$ node can update its $pred$

pointer if it finds that the notifying node is a closer predecessor than $pred$. Hence, any joining node is eventually properly incorporated into the ring.

As we mentioned previously, a single node cannot distinguish massive simultaneous node failures from a network partition. As periodic stabilization can handle massive failures [20], it also recovers from network partitions, making each component of the partition eventually form its own ring. Our simulation results confirm this, though they are omitted due to space constraints. The problem that remains unsolved, which is the focus of the rest of the paper, is how several independent rings efficiently can be merged.

3 Ring Merging

For two or more rings to be merged, at least one node needs to have knowledge about at least one node in another ring. This is facilitated by the use of *passive lists*. Whenever a node detects that another node has failed, it puts the failed node, with its routing information, in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is again alive. When this occurs, it starts a ring merging algorithm. Hence, a network partition will result in many nodes being placed in passive lists. When the underlying network merges, this will be detected and rectified through the execution of a ring merging algorithm.

A ring merging algorithm can also be invoked in other ways than described above. For example, it could occur that two SONs are created independently of each other, but later their administrators decide to merge them due to overlapping interests. It could also be that a network partition has lasted so long, that all nodes in the rings have been replaced, making the contents of the passive lists useless. In cases such as these, a system administrator can manually insert an alive node from another ring into the passive list of any of the nodes. The ring merger algorithm will take care of the rest.

The detection of an alive node in a passive list does not necessarily indicate the merger of a partition. It might be the case that a single node is incorrectly detected as failed due to a premature timeout of a failure detector. It might also be the case that a node with the same address and identifier as a failed node joins the ring. The ring merging algorithm should be able to cope with the first case, by trying to ensure that such false-positives will terminate the algorithm quickly. The latter case can be dealt with by associating with every node a globally unique random nonce, which is generated each time a node joins the network. Hence, a new node can always be differentiated from an old node with the same address.

3.1 Simple Ring Unification

In this section, we present the simple ring unification algorithm (Algorithm 1). As we later show, the algorithm will merge the rings in $O(N)$ time for a network size of N. Though we believe that the problem of dealing with network mergers is crucial, we think that such events happen more rarely. Hence, it might be justifiable in certain application scenarios that a slow paced algorithm runs in the background, consuming little resources, while ensuring that any potential problems with partitions will eventually be rectified. Later, we show how the algorithm can be improved to make it complete the merger in substantially less time.

Algorithm 1 Simple Ring Unification Algorithm

```
1:  every γ time units and detqueue ≠ ∅ at p
2:      q := detqueue.dequeue()
3:      sendto p : MLOOKUP(q)
4:      sendto q : MLOOKUP(p)
5:  end event

6:  receipt of MLOOKUP(id) from m at n
7:      if id ≠ n and id ≠ succ then
8:          if id ∈ (n, succ) then
9:              sendto id : TRYMERGE(n, succ)
10:         else if id ∈ (pred, n) then
11:             sendto id : TRYMERGE(pred, n)
12:         else
13:             sendto closestprecedingnode(id) : MLOOKUP(id)
14:         end if
15:     end if
16: end event

17: receipt of TRYMERGE(cpred, csucc) from m at n
18:     sendto n : MLOOKUP(csucc)
19:     if csucc ∈ (n, succ) then
20:         succ := csucc
21:     end if
22:     sendto n : MLOOKUP(cpred)
23:     if cpred ∈ (pred, n) then
24:         pred := cpred
25:     end if
26: end event
```

Algorithm 1 makes use of a queue called $detqueue$, which will contain any alive nodes found in the passive list. The queue is periodically checked by every node p, and if it is non-empty, the first node q in the list is picked to start a ring merger. Ideally, p and q will be on two different rings. But even so, the distance between p and q on the identifier space might be very large, as the passive list can contain any previously failed node. Hence, the event MLOOKUP(id) is used to get closer to id through a lookup. Once MLOOKUP(id) gets near its destination id, it triggers

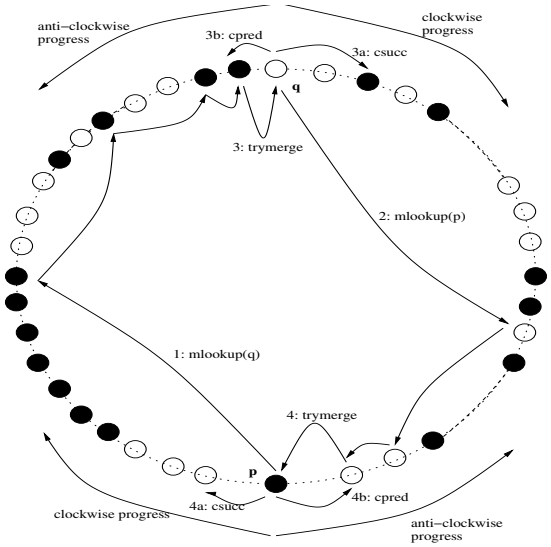


Figure 1: Filled circles belong to SON1 and empty circles belong to SON2. The algorithm starts when p detects q, p makes an MLOOKUP to q and asks q to make an MLOOKUP to p.

the event TRYMERGE($cpred, csucc$), which tries to do the actual merging by updating $succ$ and $pred$ pointers.

The event MLOOKUP(id) is similar to a Chord lookup, which tries to do a greedy search towards the destination id. One difference is that it terminates the lookup if it reaches the destination and locally finds that it cannot merge the rings. More precisely, this happens if MLOOKUP(id) is executed at id itself, or at a node whose successor is id. If an MLOOKUP(id) executed at n finds that id is between n and n's successor, it terminates the MLOOKUP and starts merging the rings by calling TRYMERGE. Another difference between MLOOKUP and an ordinary Chord lookup is that an MLOOKUP(id) executed at n also terminates and starts merging the rings if it finds that id is between n's predecessor and n. Thus, the merge will proceed in both clockwise and anti-clockwise direction.

The event TRYMERGE takes a candidate predecessor, $cpred$, and a candidate successor $csucc$, and attempts to update the current node's $pred$ and $succ$ pointers. It also makes two recursive calls to MLOOKUP, one towards $cpred$, and one towards $csucc$. This recursive call attempts to continue the merging in both directions. Figure 1 shows the working of the algorithm.

In summary, MLOOKUP closes in on the target area where a potential merger can happen, and TRYMERGE attempts to do local merging and advancing the merge process in both directions by triggering new MLOOKUPs.

3.2 Gossip-based Ring Unification

The simple ring unification presented in the previous section has two disadvantages. First, it is slow, as it takes $O(N)$ time to complete the ring unification. Second, it cannot recover from certain pathological scenarios. For example, assume two distinct rings in which every node points to its successor and predecessor in its own ring. Assume furthermore that the additional pointers of every node point to nodes in the other ring. In such a case, an *mlookup* will immediately leave the initiating node's ring, and hence terminate. We do not see how such a pathological scenario could occur due to a partition, but the *gossip-based ring unification algorithm* (Algorithm 2) rectifies both disadvantages of the simple ring unification algorithm. Also, the simple ring unification is less robust to churn, as we discuss in the evaluation section.

Algorithm 2 Gossip-based Ring Unification Algorithm

```
1:  every γ time units and detqueue ≠ ∅ at p
2:      ⟨q, f⟩ := detqueue.dequeue()
3:      sendto p : MLOOKUP(q, f)
4:      sendto q : MLOOKUP(p, f)
5:  end event

6:  receipt of MLOOKUP(id, f) from m at n
7:      if id ≠ n and id ≠ succ then
8:          if f > 1 then
9:              f := f − 1
10:             r := randomnodeinRT()
11:             at r : detqueue.enqueue(⟨id, f⟩)
12:         end if
13:         if id ∈ (n, succ) then
14:             sendto id : TRYMERGE(n, succ)
15:         else if id ∈ (pred, n) then
16:             sendto id : TRYMERGE(pred, n)
17:         else
18:             sendto closestprecedingnode(id) : MLOOKUP(id, f)
19:         end if
20:     end if
21: end event

22: receipt of TRYMERGE(cpred, csucc) from m at n
23:     sendto n : MLOOKUP(csucc, F)
24:     if csucc ∈ (n, succ) then
25:         succ := csucc
26:     end if
27:     sendto n : MLOOKUP(cpred, F)
28:     if cpred ∈ (pred, n) then
29:         pred := cpred
30:     end if
31: end event
```

Algorithm 2 is, as its name suggests, partly gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, but it starts multiple such mergers at

random places on the rings. The basic idea is to augment MLOOKUP(id), such that the current node randomly picks a node r in its current routing table and starts a ring merger between id and r. This change alone would, however, consume too much resources.

Two mechanisms are used to avoid the algorithm to consume too many messages, and therefore give rise to positive feedback cycles which congest the network. First, instead of immediately triggering an MLOOKUP at a random node, the event is placed in the corresponding node's $detqueue$, which only is checked periodically. Second, a constant number of random MLOOKUPs are created. This is regulated by a fanout parameter called F. Thus, the fanout is decreased each time a random node is picked, and the random process is only started if the fanout is larger than 1. The $detqueue$, therefore, holds tuples, which contain a node identifier and the current fanout parameter. Similarly, MLOOKUP takes the current fanout as a parameter.

4 Evaluation

In this section, we evaluate the two algorithms from various aspects and in different scenarios. There are two measures of interest: *message complexity*, and *time complexity*. We differentiate between the *completion* and *termination* of the algorithm. By completion we mean the time when the rings have merged. By termination we mean the time when the algorithm terminates sending any more messages. If not said otherwise, message complexity is until termination, while time complexity is until completion.

The evaluations are done in a stochastic discrete event simulator [28] in which we implemented Chord. The simulator uses an exponential distribution for the inter-arrival time between events (joins and failures). To make the simulations scale, the simulator is not packet-level. The time to send a message is an exponentially distributed random variable. The values in the graphs indicate averages of 18 runs with different random seeds.

We first evaluate the message and time complexity of the algorithms in the typical scenario where after merger, many nodes simultaneously detect alive nodes in their passive lists. A worst case scenario can be when only a single node detects the existence of another ring. Thereafter, we evaluate the performance of the algorithms while node joins and failures are taking place during the ring merging process. Finally, we evaluate message complexity of the algorithms when a node falsely believes that it has detected another ring.

Each simulation scenario had the following structure. Initially nodes join and fail. After a certain number of nodes are part of the system, we insert a partition event, on which the simulator divides the set of nodes into as many components as requested by the partition event. A partition event is implemented using lottery scheduling [32] to define the size of each partition. The simulator then drops all messages sent from nodes in one partition to nodes in another partition, thus simulating a network partition in the underlying network and therefore triggering the failure handling algorithms (see Section 2 and 3). Furthermore, Node join and fail events are triggered in each partitioned component. Thereafter, a network merger event simply again allows messages to reach other network components, triggering the detection of alive nodes in the passive lists, and hence starting the ring unification algorithms. For the simulations, time to send a message is exponentially distributed with mean 5 for periodic stabilization, and is 1 time unit for ring unification algorithms.

We simulated the simple ring unification algorithm and the gossip-based ring unification algorithm for partitions creating two components, and for fanout values from 1 to 7. For our simulation graphs, a fanout of 1 represents the simple ring unification algorithm.

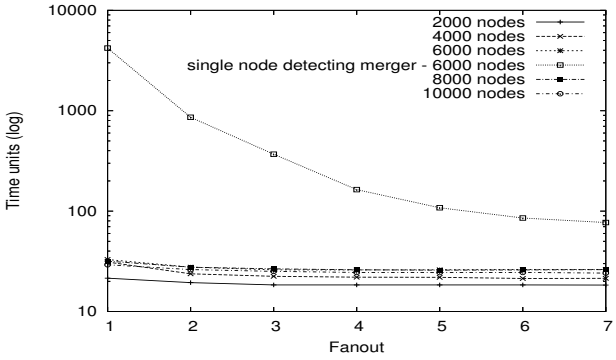


Figure 2: Evaluation of Time Complexity for a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

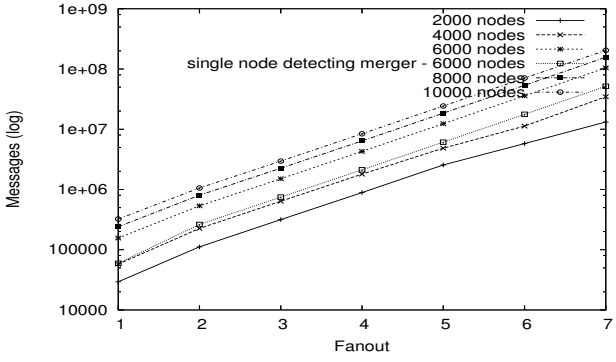


Figure 3: Evaluation of Message Complexity for a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

Figure 2 and 3 show the time and message complexity for a typical scenario where after a merger, multiple nodes

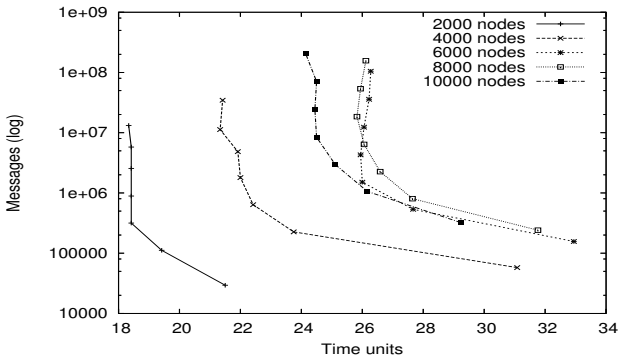


Figure 4: Comparing Time and Message complexity where multiple nodes detect the merger for various *network sizes* and *fanouts*.

detect the merger and thus start the ring-unification algorithm. The number of nodes detecting the merger depend on the scenario, in our simulations, it was 25-35% of the total nodes. The evaluation shows that while the time complexity is less for multiple nodes detecting compared to a single node detecting the merger, the message complexity is more when multiple nodes detect the merger.

As can be seen in Figures 2 and 3, the simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time. For higher values of F, the time complexity decreases while the message complexity increases. Increasing the fanout after a threshold value (around $3-4$ in this case) will not considerably decrease the time complexity, but will just generate many messages. Figure 4 shows a tradeoff between time complexity and message complexity. Choosing to have less time for completion of mergers will create more messages, and vice versa. As can be seen from Figure 3, the algorithm generates a lot of messages before termination, though the completion property might have been satisfied earlier. We would like to further explore optimizations to reduce the number of messages sent.

For the rest of the evaluations, we use a worst case scenario where only a single node detects the merger.

Next, we evaluate rings unification under churn, *i.e.* nodes join and fail during the merger. Since we are using a scenario where only one node detects the merger, with a very low probability, the algorithm may fail to complete and the merged overlay may not converge under churn, especially for simple ring unification and low fanouts. The reason being intuitive: for simple unification, the two MLOOKUPs generated by the node detecting the merger while traveling through the network may fail as the node forwarding the MLOOKUP may fail under churn. With higher values of F and in typical scenarios where multiple nodes detect the merger, the algorithm becomes more robust to churn as it creates multiple MLOOKUPs. The results presented in Figure 6 and 5 are only when the rings successfully converge. For simulation, after a merge event, we generate events of joins and fails until the unification algorithm terminates. With high churn, we mean that the inter-arrival time between events of joins and fails is less, thus representing highly dynamic conditions. Choosing a high inter-arrival time between events will create less joins and fails and thus churn will be less. For the simulations presented here, we choose inter-arrival time between events of joins and failures to be 30 units for high churn and 45 units for low churn, and an equal probability for a event to be a join or a fail. Figure 6 and 5 show how different values of F affect the convergence of the rings under different levels of churn, mainly showing the algorithm works under churn without effecting message and time complexity much.

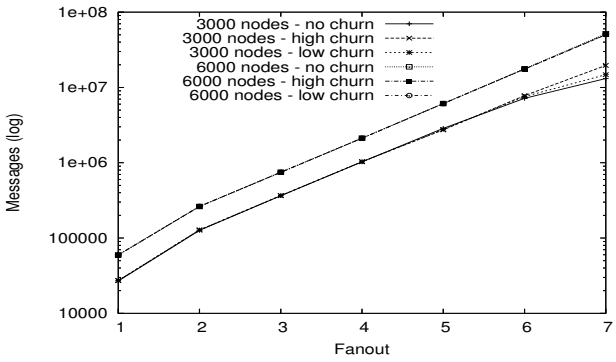


Figure 5: Evaluation of Time Complexity under churn

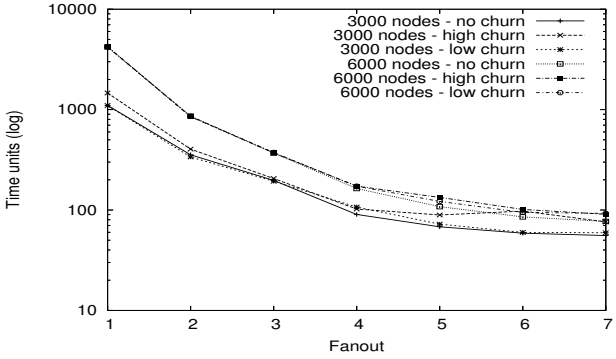


Figure 6: Evaluation of Message Complexity under churn

Finally, we evaluate the scenario where a node may falsely detect a merger. Figure 7 shows the message complexity of the algorithm in case of a false detection. As can be seen, for lower fanout values, the message complexity is less. Even for higher fanouts, the number of messages generated are acceptable, thus showing that the algorithm is lean. We believe this to be important as most SONs do not have perfect failure detectors, and hence can give rise to inaccurate suspicions.

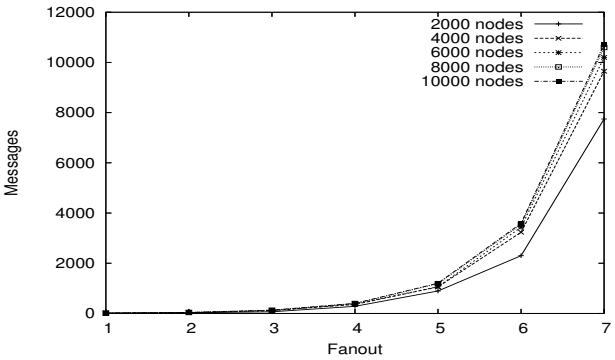


Figure 7: Evaluation of Message Complexity in case a node falsely detects a merger for various *network sizes* and *fanouts*.

Our simulations show that a fanout value of 4 is good for a system with several thousand nodes, even with respect to churn and false-positives.

5 Related Work

Much work has been done to study the effects of churn on a structured overlay network [22], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta *et al.* [4] have presented the challenges of merging two overlays, claiming that ring-based networks cannot operate until the merger operation completes. In contrast, we show how unification can work under churn while the merger operation is not complete. Birman [2] argued that ring-based SONs are inherently ill-suited for dealing with network partitions, while we show how ring-based SONs can be modified to deal with partitions.

The problem of constructing a SON from a random graph is, in some respects, similar to merging multiple SONs after a network merger, as the nodes may get randomly connected after a partition heals. Shaker *et al.* [27] have presented a ring-based algorithm for nodes in arbitrary state to converge into a directed ring topology. Their approach is different from ours, in that they provide a non-terminating algorithm which should be used to replace all join, leave, and failure handling of an existing SON. Replacing the topology maintenance algorithms of a SON may not always be feasible, as SONs may have intricate join and leave procedures to guarantee lookup consistency [21, 19, 9]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing SON.

Montresor *et al.* [25] show how Chord [29] can be created by a gossip-based protocol [14]. However, their algorithm depends on an underlying membership service like Cyclon [31], Scamp [8] or Newscast [15]. Thus the underlying membership service has to first cope with network mergers (a problem worth studying in its own right), whereafter T-Chord can form a Chord network. We believe one needs to investigate further how these protocols can be combined, and their epochs be synchronized, such that the topology provided by T-Chord is fed back to the SON when it has converged. Though the general performance of T-Chord has been evaluated, it is not known how it performs in the presence of network mergers when combined with various underlying membership services.

The problem of network partitions and mergers has been studied in other distributed systems like in distributed databases [5] and distributed file systems [30]. These studies focus on problems created by the partition and merger on the data level, while we focus on the routing level.

6 Conclusion

We have argued that the problem of partitions and mergers in structured peer-to-peer systems, when the underlying network partitions and recovers, is of crucial importance. We have presented a simple and a gossip-based algorithm for merging similar ring-based structured overlay networks after the underlying network merges. Our algorithm is quite fast compared to the basic linear solution presented by Datta *et al.* [4]. We have shown how the algorithm can be tuned to achieve a tradeoff between the number of messages consumed and the time before the overlay converges. We have evaluated our solution in realistic dynamic conditions, and showed that with high fanout values, the algorithm can converge quickly under churn. We have also shown that our solution generates few messages even if a node falsely starts the algorithm in an already converged SON.

We tried many variations of the algorithms before reaching those that are reported in this paper. Initially, we had an algorithm that was not gossip-based, i.e. was not periodic and did not have any randomization. Albeit the algorithm was quite fast, it heavily over-consumed messages, making it infeasible for a large scale network. For that reason, we added the fanout parameter, and made it run periodically. Without randomization, we could construct pathological scenarios, in which that algorithm would not be able to merge the rings.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, pages 353–366, Portland, OR, USA, March 2004. ACM Press.

[2] Ken Birman. Gossip Algorithms and Emergent Shape. Invited talk at the Workshop on Gossip-based Computer Networking at the Lorentz Center, Leiden, Netherlands, December 2006.

[3] E. Brewer. Towards Robust Distributed Systems, invited talk at the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00), 2000.

[4] A. Datta and K. Aberer. The Challenges of Merging Two Similar Structured Overlays: A Tale of Two Networks. In *Proceedings of the First International Workshop on Self-Organizing Systems (IWSOS'06)*, volume 4124 of *Lecture Notes in Computer Science (LNCS)*, pages 7–22. Springer-Verlag, 2006.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[7] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient Broadcast in Structured P2P Netwoks. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 304–314, Berkeley, CA, USA, 2003. Springer-Verlag.

[8] A. J. Ganesh, A.-M. Kermarrec, and L Massoulié. SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. In *Proceedings of the 3rd International Workshop on Networked Group Communication (NGC'01)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 44–55, London, UK, 2001. Springer-Verlag.

[9] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, December 2006.

[10] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *Proceedings of the 15th International Conference, Parallel and Distributed Computing and Systems*, Marina del Rey, CA, USA, November 2003.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Special Interest Group on Algorithms and Computation Theory News*, 33(2):51–59, 2002.

[12] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Symposium on Communication, Architecture, and Protocols*, pages 381–394, New York, NY, USA, 2003. ACM Press.

[13] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[14] M. Jelasity and Ö. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of 3rd Workshop on Engineering Self-Organising Systems (EOSA'05)*, volume 3910 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer-Verlag, 2005.

[15] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit, November 2003.

[16] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the 2nd Interational Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 98–107, Berkeley, CA, USA, 2003. Springer-Verlag.

[17] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004. IEEE Computer Society.

[18] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005. USENIX.

[19] X. Li, J. Misra, and C. G. Plaxton. Brief Announcement: Concurrent Maintenance of Rings. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, page 376, New York, NY, USA, 2004. ACM Press.

[20] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Observations on the Dynamic Evolution of Peer-to-Peer Networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2002.

[21] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 295–305, London, UK, 2002. Springer-Verlag.

[22] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 21–32, Berkeley, CA, USA, 2003. Springer-Verlag.

[23] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, New York, NY, USA, 2002. ACM Press.

[24] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[25] A. Montresor, M. Jelasity, and Ö. Babaoglu. Chord on Demand. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*. IEEE Computer Society, August 2005.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 2nd ACM/IFIP International Conference on Middleware (MIDDLEWARE'01)*, volume 2218 of *Lecture Notes in Computer Science (LNCS)*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[27] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*, pages 39–46. IEEE Computer Society, August 2005.

[28] SicsSim, 2007. http://dks.sics.se/p2p07partition/.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[30] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 172–183. ACM Press, December 1995.

[31] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

[32] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11. USENIX, November 1994.

A.16 Reliable Dynamic Reconfiguration of Component-Based Systems

Reliable dynamic reconfigurations in the Fractal component model*

Marc Léger
France Telecom R&D
28, chemin du Vieux Chêne
38243 Meylan, France
marc.leger@orange-ftgroup.com

Thomas Ledoux
OBASCO Group, EMN/INRIA, LINA
Ecole des Mines de Nantes
4, rue Alfred Kastler
44307 Nantes, France
thomas.ledoux@emn.fr

Thierry Coupaye
France Telecom R&D
28, chemin du Vieux Chêne
38243 Meylan, France
thierry.coupaye@orange-ftgroup.com

ABSTRACT

This article is an analysis based on our experience with the Fractal component model of the need of reliability for dynamic reconfigurations in component based systems. We make a proposal to ensure this reliability for concurrent and distributed Fractal applications. We started from the definition of ACID properties in the context of dynamic reconfigurations in component models and we propose to use integrity constraints to define system consistency and transactions for guaranteeing the respect of these constraints at runtime. Moreover we manage concurrency between reconfigurations by avoiding potential conflicts between reconfiguration operations. Finally, a recovery mechanism has been conceived to deal with failures.

Keywords

reflexive architectures, reliability, Fractal component model, dynamic reconfigurations

1. INTRODUCTION

Dynamic reconfigurations in component-based software applications [12] are central to promising approaches like autonomic computing [9]. There are many motivations to introduce modifications in a system at runtime: correction of security flaws or functional bugs, improvement of systems (e.g., performance optimizations), or adaptations to execution context changes.

Thanks to properties of component models like modularity and the causal connection of reflexive architectures, dynamic reconfigurations can rely on component-based architectures [16]. The Fractal component model [4] provides these good properties to reconfigure applications. In this model, reconfigurations can be **structural** (e.g., addition or removal of components) or **behavioral** (e.g., lifecycle modifications). However, runtime modifications may let the system in an inconsistent state i.e. no more available or usable from a functional point of view. Indeed the architecture of the system once reconfigured can be not in conformity with the component model or possibly system specific constraints (e.g. architectural invariants) anymore. For instance adding a component *C1* in another component *C2* could be an invalid reconfiguration if it creates a cycle in the component hierarchy (i.e. if *C2* contains *C1*) which is forbidden by the Fractal model.

Thus reliability as a key concept of dependability is a main problem in systems subject to dynamic reconfigurations, especially with open models like Fractal because reconfigurations can be unanticipated at compile time and consequently, static analysis of reconfigurations is difficult. The extensibility of the model must also be taken into account because extensions should be defined in a reliable way without breaking the core specification.

Our proposal aims to ensure the reliability of dynamic reconfigurations in the Fractal component model (but it can be generalized to other models), and can be applied to concurrent and distributed reconfigurations. We will show how ACID properties in the context of reconfigurations can improve reliability by making systems fault-tolerant i.e., compliant with the specification in spite of faults due to dynamic reconfigurations. These properties are unifying concepts of transactions for distributed computation [17] used for supporting concurrency, recovery, and guaranteeing system consistency. To benefit from these properties, each reconfiguration, defined as a composition of introspection and intercession operations, is executed as a separated transaction called a reconfiguration transaction.

This paper is organized as follows. Section 2 is an overview of dynamic reconfigurations in the Fractal component model and it shows what problems it raises regarding reliability. Then section 3 describes how transactions combined with integrity constraints can be a solution to these problems. Finally section 4 presents some related works before concluding in section 5.

*This research is supported by the French RNTL project SELFWARE (`http://www.rntl.org/projet/resume2005/selfware.htm`) and the European IST project SELFMAN (`http://www.ist-selfman.org/wiki/index.php/SELFMAN_Project`).

ARM 97 Newport Beach, California, USA

2. DYNAMIC RECONFIGURATIONS IN THE FRACTAL COMPONENT MODEL

2.1 The Fractal model and associated tools

Fractal is a hierarchical component model with sharing and reflexive control. It is based on classic concepts of component (as a runtime entity), interface (an interaction point between components expressing provided and required services) and binding (a communication channel between component interfaces) as it can be seen on the simple *ClientServer* example in figure 1. A component consists of a membrane which can show and control a causally connected representation of its encapsulated content. The content is either directly an implementation in case of a primitive component (the Client and Server components in the example) or sub-components for composite components (e.g., the ClientServer component).

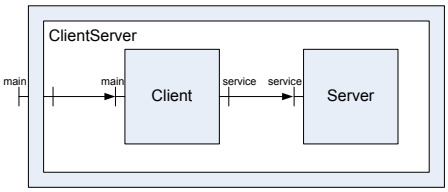


Figure 1: A simple client-server Fractal application

There are several implementations of the model in different languages, the Julia [3] implementation in Java has been chosen for our experimentations. Julia is an execution framework for Fractal-based application which allows to program component membranes with interceptors. Moreover, an Architecture Description Language (Fractal ADL [7]) is used to specify and deploy component configurations with an extensible XML-based syntax. A basic ADL specification of the *ClientServer* example in figure 1 could be the following:

```
<definition name="ClientServer">
 <interface name="main" role="server"
        signature="java.lang.Runnable" />
 <component name="client">
  <interface name="main" role="server"
        signature="java.lang.Runnable" />
  <interface name="service" role="client"
        signature="Service" />
  <content class="ClientImpl" />
 </component>
 <component name="server">
  <interface name="service" role="server"
        signature="Service" />
  <content class="ServerImpl" />
 </component>
 <binding client="this.main" server="client.main" />
 <binding client="client.service" server="server.service" />
</definition>
```

In addition to the ADL, there is another Domain-Specific Language (DSL), FPath [6], which is a query language for runtime Fractal applications with a XPath-like syntax. It is restricted to the introspection of architectures, navigating inside them to locate elements of interest by their properties or location in the architecture. It is based on a representation of Fractal architectures using directed labeled graphs (cf. figure 2): core entities (components, interfaces and attributes) in Fractal are represented by nodes, and their relations by directed arcs (e.g., the binding relation between component interfaces). For instance, to select all subcomponents of the *ClientServer* component (i.e the Client and Server components), we can use the following FPath expression: `$cs/child::*` where *cs* is a variable initialized with the node representing the ClientServer component.

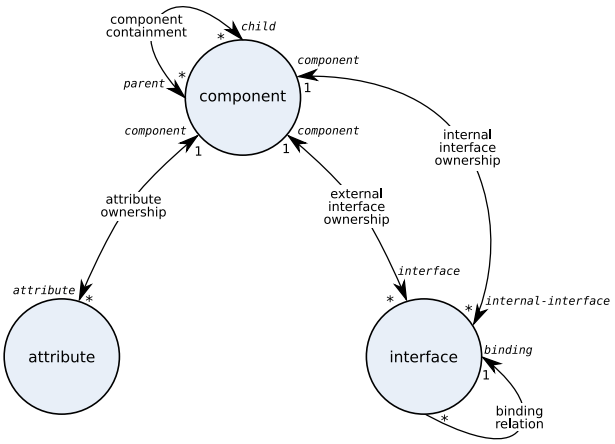


Figure 2: FPath representation of the Fractal component model

2.2 The reliability problems of dynamic reconfigurations

Dynamic reconfigurations allow to modify a part of a system during its execution without stopping it entirely. They may involve every manageable element defined in the component model provided they are reified at runtime. Some specific interfaces in Fractal called controllers concern reconfigurations. Some controllers are defined by the Fractal API but, as the model is extensible, other controllers can be user-defined:

- **BindingController**: to bind and unbind component interfaces, and to introspect bindings.
- **ContentController**: to add or remove sub-component for composite components, and to introspect component hierarchy.
- **SuperController**: to get parent components for a given component.
- **LifeCycleController**: to manage component life cycle, basically with *stop* and *start* operations.
- **AttributeController**: to set and get attribute values.
- **NameController**: to set and get component names.

Operations in Fractal controllers deal with non-functional concerns and constitute primitive reconfiguration operations. We distinguish two kinds of operations belonging to reconfiguration operations (N.B. they are not separated in Fractal controllers):

- **Introspection operations** are without side effects (e.g., the *lookupFc* method in *BindingController* to retrieve the server interfaces which are bound to a given client interface).

- **Intercession operations** modify the system (e.g., the *bindFc* method in *BindingController* to bind two component interfaces).

Reconfiguration can be composed as sequences of intercession operations with conditions expressed by means of introspection operations in component configurations. A classic example of reconfiguration is component hot-swap, i.e replacing an old version of a component by a new one to update an application. In Fractal, this reconfiguration can be implemented with a sequence of several primitive reconfiguration operations of the Fractal API, it implies to *stop* the component, *unbind* all its interfaces, *remove* it from all its parents, *add* the new instantiated component in all the parents, *bind* its interfaces and *start* it (a state transfer operation may be needed if the component is stateful). It is noticeable that another means to express such a reconfiguration in Fractal is the FScript language [6] which is a DSL dedicated to reconfigurations in the Fractal model.

A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functional execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hot-swap example with a stateful component, calls on the old component must be blocked until a "quiescent state" [10] is reached, then the state must be transfered, finally previous calls are forwarded towards the new component. This synchronization currently relies on the implementation of the life cycle management of components in Julia.

We will focus on another problem at the non-functional level about consistency violation by reconfigurations. Component models should define what a consistent system is, for instance in terms of structural assemblies, and dynamic reconfigurations should not break this consistency. In addition to the consistency defined by the Fractal component model, we want to be able to add application specific constraints. For instance, we may want to add a structural constraint about the number of subcomponents of a composite component. Then we must ensure the conformity of the system to the model and constraints after reconfigurations.

3. A TRANSACTIONAL APPROACH TO ENSURE RELIABLE RECONFIGURATIONS

We think that well-defined transactions associated with the verification of structural and behavioral constraints is a means to guarantee the reliability of reconfigurations i.e., the system stays consistent after reconfigurations. The following definition of the ACID properties in the context of dynamic reconfigurations in component-based systems is given:

Atomicity.

Either the system is reconfigured and the reconfiguration transaction commits (all the operations forming the transaction are executed) or it is not and the transaction aborts. If a reconfiguration transaction fails, the system comes back in a previous consistent state as if the transaction never started.

Consistency.

A reconfiguration transaction is a valid transformation of the system state i.e. it takes the considered system from a consistent state to another consistent state. A system is a consistent state if and only if it conforms to our consistency criteria: it does not violate the component model and possibly more specific constraints.

Isolation.

Reconfiguration transactions are executed as if they were independent. Results of reconfiguration operations inside a non-committed reconfiguration transaction are not visible from other transactions until the transaction commits ; or never if the transaction aborts.

Durability.

The results of a committed reconfiguration transaction are permanent: once a reconfiguration transaction commits, the new state of the system (both the architecture description and the component state) is persisted so that it can be recovered in case of major failures (e.g., hardware failures).

Regarding the transaction model, flat transactions appear sufficient for short-lived reconfiguration transactions we consider here. The Two-Phase-Commit protocol [17] is used to coordinate distributed participants in transactions. Each ACID property and associated mechanisms will now be detailed more precisely.

3.1 Atomicity of reconfigurations

By default, as any reconfiguration could lead the system to an inconsistent state, the execution model of reconfiguration transactions is that every reconfiguration operation must be included inside a transaction: if an operation is executed outside a transaction, a new transaction is automatically created to include this operation. However, this policy can be configured for each reconfiguration operation so that it is not executed in a transaction but then no guarantee is provided that the system will be recovered properly. This last possibility is notably useful when there is no concurrent reconfiguration in the system, so introspection operations cannot do dirty reads and consequently they don't need to be isolated. An explicit language demarcation is necessary when reconfigurations are programmed directly in Java thanks to calls to a *TransactionManager*, otherwise every primitive reconfiguration operation which is not demarcated constitutes itself a single transaction.

In our approach, to ensure atomicity of reconfiguration transactions, operations performed in transactions which abort are undone. Actually, only intercession operations need to be taken into account. When operations are reversible, undoing a reconfiguration transaction is equivalent to sequentially undo primitive intercession operations in reverse order. For every transaction, demarcation and intercession operations are logged in a journal so that it can be undone in case of rollback.

An extensible library of wrappers in Java for primitive operations from the Fractal API is proposed (cf. figure 3) where each intercession operation is associated to its reverse intercession operation in the semantics we choose (typically *bindFc* and *unbindFc* for two component interfaces are inverse operations). Moreover, an operation can keep a state so as to be undone (e.g., changing the value of an attribute requires to keep the old value of this attribute so that the operation can be undone). On the other hand, it is also possible to define non-reversible intercession operations but they need a customized treatment of the rollback (by capturing a *RollbackException* or by listening to transaction events if the transaction participant implements a *TransactionLis-*

tener interface) so as to execute some compensation operations.

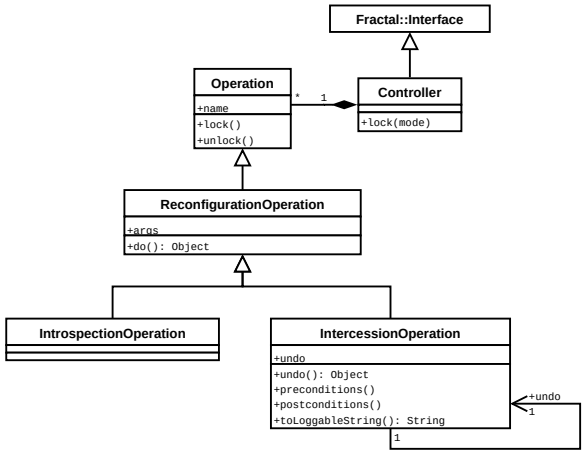


Figure 3: Reconfiguration operation model.

3.2 Integrity constraints to ensure system consistency

In our proposal, system consistency relies on integrity constraints and we want to express these constraints both on the component model and on applications. An *integrity constraint* is a predicate on assemblies of architectural elements and component state [11]. Therefore, a reconfiguration transaction can be committed only if the resulting system is consistent i.e., if all integrity constraints on the system are satisfied. To express constraints including invariants, preconditions and postconditions, we use FPath as a constraint language "à la OCL" [15]. Some advantages of FPath are that it can navigate and select Fractal elements at runtime with just introspection capacities. Moreover it is already based on a graph representation of the system during execution. We distinguish three different levels of constraint specification (cf. figure 4).

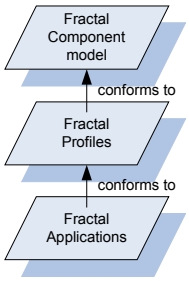


Figure 4: The three levels of integrity constraint.

The model level.

This a set of generic constraints associated to the component model. These constraints apply to all instances of some elements of the Fractal model. Examples of such constraints are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid recursion). We specified the semantics of reconfiguration operations in the model with preconditions and postconditions which should never be violated. An example of such a precondition for the remove operation is to check that all interfaces of the component to remove are unbound:

```
Fractal API: void removeSubComponent(Component child);
// preconditions:
not($child/interface::*[bound(.)]);
...
```

The profile level.

This is a set of generic constraints to refine the component model for a family of applications. The profile level conforms to the model level. A profile may for instance forbid component sharing in applications with the constraint `size(./parent::*)<=1` which says that every component cannot have more that one super-component.

The application level.

Application constraints are specific to a given architecture and apply directly to instances of Fractal elements designed by their names. The application level conforms to the profile and to the model. Invariants can concern cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc. Application constraints are specified in Fractal ADL with the *constraint* tag. For instance, we may want to add a structural invariant on the component *ClientServer* saying that it always contain only one component providing the *service* interface:

```
<definition name="ClientServer">
 ...
 <constraint value="size(./child::*[./interface::service])=1" />
</definition>
```

Constraints must be checked both at compile time on the component static configuration and at runtime and only new application constraints can be dynamically added or removed by means of a *ConstraintController*. FPath expressions are parsed once when they are added to a component. Pre/post-conditions of primitive intercession operations are checked at each operation execution whereas invariants are only checked at commit of transactions. That is to say a system can temporarily violate invariants during a transaction but it must be in a correct state after commit. When it is detected, a constraint violation makes the transaction involved rollback.

3.3 Isolation of reconfigurations to support concurrency

We make the hypothesis that several administrators may want to reconfigure the same system (potentially distributed) at the same time or a single reconfiguration is itself composed of parallel reconfigurations to optimize the reconfiguration process. Furthermore, reconfiguration initiators are either humans (interactive reconfigurations) or the system itself is able to auto-reconfigure (automatic reconfigurations). There are two available scheduling policies adapted to the level of concurrency for reconfigurations in Fractal applications: either reconfigurations are concurrently executed and accesses to Fractal elements in the system must be synchronized or reconfigurations are serially executed, i.e., only one reconfiguration is executed at a time while others are simply queued.

For concurrency management, we adopt a pessimistic approach with strict Two-Phase locking [17] to provide strong concurrency (corresponding to *serializable* isolation level): locks acquired during a transaction are only released when the transaction is committed to avoid *phantoms* (update

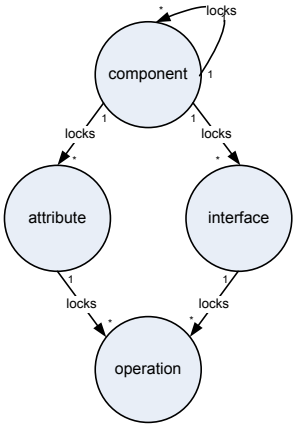


Figure 5: Hierarchical locking model

lost). When a deadlock is detected in the waiting graph of transactions (i.e. a cycle), a transaction is automatically rollbacked and possibly retried automatically several times with a delay. Several elements in the model corresponding to FPath nodes are lockable with a hierarchical algorithm (cf. figure 5): for instance, a lock acquisition on a component for instance will also lock all its interfaces. Locks are reentrant (a transaction can acquire several times the same lock) and there are basically two types of locks: read (or shared) locks, and write (or exclusive) locks. A *LockController* is provided with every component to lock its sub-elements, locks can be also acquired by means of an FPath Expression. For example, we can read lock all subcomponent of the component *ClientServer*: `lock(READ, $cs/child::*);`

We use this locking model at a finer granularity to avoid inconsistent state for reconfiguration operations based on their semantics. Basically, an introspection operation in a transaction will take read locks on other reconfiguration operations to avoid *dirty reads* and *fuzzy reads* (non repeatable reads): it should not see modifications in the system due to other transactions which have not been committed yet. However, two operations in two different transactions can introspect the same element in the system by sharing a read lock. On the other hand, intercession operations on a given element will take write locks to avoid conflicts between state modifications: other transactions can neither introspect the same element nor modify it. For example, if we want to add the component *Server* in the component *ClientServer*, it will write locks operations in the *ContentController* of the parent and operations in the *SuperController* of the child (*Server*).

```
protected void lock() throw DeadLockException {
  lock(WRITE, "$parent/interface::content-controller");
  lock(WRITE, "$child/interface::super-controller");
}
```

3.4 Durability of reconfigurations for recovery

The journal of a reconfiguration transaction is kept both in memory and persisted on disk and in case of failure, the journal is redone: all transactions which are not committed are canceled and all committed transactions since the last checkpoint are redone. The FPath syntax is used in the log format and Fractal elements in arguments of operations are designed with one absolute path (path is not unique due to sharing) in the system architecture.

In addition to the journalization of transactions, the system state is periodically checkpointed. We consider here the state of a component-base system as its architecture description and the set of its component state. Chekpointing allows to reboot any component in its last know consistent state resulting from a previous successful reconfiguration. The frequency of checkpointing can be adjusted in terms of a number of reconfiguration transactions, it can be basically after each transaction commit.

A component state is checkpointed with Fractal ADL dumps and attribute values are saved either in file or in database. The ADL dumper is extensible so as to consider eventual extensions of the ADL language. State persistence is the responsibility of the *StateController* for each component. Attribute values which are of primitive types are automatically saved in ADL dumps. There are two implementations for other attribute values which are Java Object: either a Java serialization in a file named with an absolute path of the component or it can be saved in a database if an ORM is defined for the attributes of the component (our implementation uses Hibernate [1]). Attributes are tagged as persistent in an extension of the attribute module in Fractal ADL.

```
<component name="server">
 <attributes signature="ServiceAttributes">
  <attribute name="header" persistent="true" />
  <attribute name="count" value="0" />
 </attributes>
 <persistence file="log/state.save"/>
</component>
```

4. RELATED WORK

Many works on ADLs follow a static approach to check consistency of component-based architectures by compilation [13] but only a few are interested in dynamic analysis of this consistency. One interesting works which supports dynamic reconfigurations while still offering strong guarantees is ArchJava [1], which extends Java with component-based concepts. ArchJava architectures can be reconfigured dynamically and the language guarantees communication integrity during execution. However, these guarantees are only possible because the reconfigurations are hard-coded in the program. So we will focus on more recent component models which are reflexive and allow non anticipated (also called ad-hoc) reconfigurations.

FORMAware [14] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is our integrity constraints are more flexible than styles as they can be applied at the model level or directly to specific instances with pre/post-conditions and invariants and they can be applied to every element of our component model. Moreover our definition of the transactional semantics of reconfiguration seems to be more precise.

Plastik [2] is the integration of the OpenCOM[5] component model and the ACME/Armani ADL [8]. As in our solution, architectural invariants can be checked at runtime and constraints are expressed at two levels (style and instance). However we define more formally reconfiguration operations to identify conflicts between them so that our locking algorithm is more precise than a simple lock on components and it also considers introspection operations.

[1] http://www.hibernate.com

5. CONCLUSION

Dynamic reconfiguration in component-based systems raises reliability problems, especially in open systems in which they are not anticipated. In this article, we identified the following global problems based on our experience with the Fractal component model: maintaining the consistency regarding component and application models, synchronizing reconfiguration operations between them and recovering applications in case of failure.

We propose to use integrity constraints at several levels (model, profile and application) to define consistency for dynamic reconfigurations and to include these reconfigurations in transactions. We use a graph representation of our application at runtime thanks to the reflexivity of the Fractal component model and apply a constraint language on this graph. To manage concurrency between reconfigurations, we use a locking approach based on the semantics of reconfiguration operations. To recover Fractal applications in case of failure, transaction are journalized and components are checkpointed as architecture description and value of attributes.

We plan notably for future work to do a better check of conflicts between constraints so as to be able to guarantee constraint compatibility. As for concurrency management, we are studying the possibility to use an optimistic approach with validations to avoid the cost of locking especially when they are few concurrent reconfigurations in the system. We also aim to integrate our work in an autonomic platform as a reliable concurrent and distributed solution for dynamic reconfigurations of components.

6. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *International Conference on Software Engineering, ICSE 2002*, Orlando, Florida, USA, May 2002.

[2] T. V. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In R. Morrison and F. Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[4] r. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[5] G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, Nov. 2004.

[6] P.-C. David and T. Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[7] Fractal ADL. *http://fractal.objectweb.org/fractaladl*.

[8] D. Garlan, R. T. Monroe, and D. Wile. Acme: architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of component-based systems*, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[9] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[10] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[11] M. Léger, T. Coupaye, and T. Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In R. Rousseau, C. Urtado, and S. Vauttier, editors, *LMO*, pages 21–36. Hermès Lavoisier, 2006.

[12] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[13] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), Jan. 2000.

[14] R. S. Moreira, G. S. Blair, and E. Carrapatoso. Supporting adaptable distributed systems with formaware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[15] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[16] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[17] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.

A.17 Language Support for Navigation and Reliable Reconfiguration of Component-Based Architectures

FPath & FScript: Language Support for Navigation and Reliable Reconfiguration of Fractal Architectures*

Pierre-Charles David[⋆] Thomas Ledoux[⋆] Marc Léger[⋆,†]
Thierry Coupaye[†]

[⋆] OBASCO Group, EMN/INRIA, LINA
Ecole des Mines de Nantes
4 rue Alfred Kastler
44307 Nantes, France

[†] France Telecom R&D
28 chemin du Vieux Chêne
38243 Meylan, France

Abstract

Component-based systems must support dynamic reconfigurations to adapt to their execution context, but not at the cost of reliability. Fractal provides intrinsic support for dynamic reconfiguration, but its definition in terms of low-level APIs make it complex to write reconfigurations and to ensure their reliability. This article presents a language-based approach to solve these issues: direct and focused language support for architecture navigation and reconfiguration make it easier both to write the reconfigurations and to ensure their reliability. Concretely, this article presents two languages: *(i)* FPath, a DSL which provides a concise yet powerful notation to navigate inside and query Fractal architectures, and *(ii)* FScript, a scripting language which embeds FPath and supports the definition of complex reconfigurations. FScript ensures the reliability of these reconfigurations thanks to sophisticated run-time control, which provides transactional semantics (ACID properties) to the reconfigurations.

I Introduction

Component-based software engineering (CBSE) is a powerful approach to build complex software [35]. Its explicit representation of dependencies between components make it easier to reuse building blocks, to assess the validity of an architecture, and simply to understand complex systems.

In practice, a system's architecture is rarely static: the rapidly changing contexts in which applications are used today mean that systems must be adapted, or even adapt themselves [21], to changes during their lifetime. For component-based systems, this means they must support dynamic reconfigurations, including unanticipated ones [32]. This must not come at the price of the system's reliability however: in a world of interconnected services, users (human or not) rely on services to be always available. As much as possible, reconfigurations need to happen at run-time and in a way that does not jeopardize the system's dependability.

The Fractal component model [7] provides a good base to build such adaptable and autonomic systems. Indeed, one of the most interesting and powerful features of the Fractal component

*This research is supported by the French RNTL project Selfware (`http://www.rntl.org/projet/resume2005/selfware.htm`) and the IST project Selfman (`http://www.ist-selfman.org`).

model is its thorough support for reflection: it is possible both to *discover* the structure of a Fractal application (introspection) and to *modify* it (intercession) at run-time. However, Fractal is defined in terms of a relatively low-level API [8]. This has two main consequences in our context:

1. This makes it relatively *difficult to write dynamic reconfigurations*. Because Fractal does not extend the underlying language (contrary to ArchJava [2] for example), the resulting code mixes different levels of abstractions and concepts (for example language-level interfaces and component interfaces). This is amplified by the minimalist nature of the API and its heavy use of exceptions for error signaling, which quickly lead to verbose and convoluted programs.

2. It makes it even more difficult to write *correct dynamic reconfigurations*. Because Fractal allows almost every aspect of an application to be reconfigured at run-time, even the slightest error in the reconfiguration program can make the system unusable. The Fractal model and its implementations provide some guarantees on the architectural soundness of reconfigurations (for example, all mandatory client interfaces of a component must be bound before it can be started), but this only applies to individual operations at the API level, not to reconfigurations as a whole. In addition, most concrete applications have specific architectural invariants to verify which go much beyond what the core Fractal model can provide.

Our proposal uses a *language-based approach* to overcome both these issues. Offering direct language support for manipulating Fractal architectures, makes it much *easier* to discover and modify them: programs are more concise, have a higher level of abstraction by directly manipulating the relevant concepts, and do not mix these concepts with lower-level, "technical" ones. This approach also allows us to ensure the *reliability* of the reconfigurations. First, focusing the language on the single domain of Fractal architectures and simply not allowing other concerns to be expressed in the language, makes it possible to ensure *by construction* that reconfiguration programs will only be able to manipulate the architectural concepts exposed in the language. Furthermore, because with the complete control on the language's implementation, it is possible to integrate sophisticated run-time techniques to control the reconfigurations' execution, and even – thanks to the language's restricted power of expression – static ("compile-time") analyses to validate the reconfiguration before their actual execution, both in a completely transparent way from the programmer's point of view.

Concretely, this article presents two complementary languages:

- *FPath* is a Domain-Specific Language (DSL) [26] for querying Fractal architectures. Its domain is restricted to the *introspection* of architectures, navigating inside them to locate elements of interest by their properties or location in the architecture. This focused domain allows FPath to offer a very concise yet powerful and readable syntax.

- *FScript* is a scripting language which allows for the definition of complex reconfigurations of Fractal architectures, and nothing else. FScript integrates FPath seamlessly in its syntax, FPath queries being used to select the elements to reconfigure. The restricted power of the language ensures that FScript programs can only talk about Fractal architectures, and can not execute "dangerous" and difficult to control code constructs (infinite loops, IO, etc.) as would be the case in a general-purpose scripting language. To ensure the reliability of its reconfigurations, FScript considers them as *transactions*, with all the usual ACID properties (adapted to our context). The FScript interpreter integrates a back-end system which implements this transactional semantics on top of the Fractal model.

Both languages can be used either from Java through a simple API to interact with their interpreters, or using an shell-like console for interactive exploration and interaction.

The article is structured in two main parts. Section II presents FPath, and show how it can be used by itself as a query language for Fractal architectures. Section III then describes the full FScript language (which embeds FPath), including the reliability guarantees it offers on dynamic reconfiguration of Fractal systems (Section III.3). Finally, section IV discusses some related work and section V concludes with some hints on future work.

II Querying Fractal Architectures With FPath

FPath is a Domain-Specific Language [26] to query and introspect Fractal-based systems. FPath expressions can be used to select architectural elements in a target system according to a combination of their properties (e.g. the state of a component) and their relations to others (e.g. the sub-components of a composite). FPath has been originally designed as a part of FScript, where it is used to select the elements to reconfigure. However, it is also usable by itself as a general querying/addressing language for Fractal systems, which is why it is described in its own section.

Both the syntax and the execution model of FPath are inspired by the XPath [38] language. XPath is the standard query language for XML documents defined by the W3C, used by many other XML technologies (XLink, XSLT, XQuery...). The use of XPath as a model for FPath was motivated by the following features:

- XPath does not depend on the syntax of XML documents, but only on an abstract graph model. This makes the approach suitable other graph-like models, in particular component architectures.
- Although XPath defines a fixed set of node types and relations suitable for XML documents, the syntax is open-ended and supports the definition of new kinds of nodes and relations between them without changing the language. This is important in our case to support the same level of extensibility than the Fractal model itself. If a new Fractal extension is defined which introduces new architectural elements and/or new relations between components (for example an aspect weaving relationship [30]), it should be possible to use it in FPath without changing the syntax of the language.
- XPath expressions can have varying degrees of precision. This make it possible in FPath to write very precise expressions, which will locate an element at a specific location in an architecture (for example "the component which implements service S for the direct child of composite C_1 named C_2"), but also more generic and less brittle expressions which will work on a wider range of architectures (for example "any component which provides service S and is contained, directly or not, in C_1").
- The syntax is reasonably concise and readable, and the execution model is simple to understand for users while still allowing different implementation strategies.

It should be noted that although FPath is *inspired* by XPath, it is not *implemented* using XPath, and does not use any XML representation of Fractal architectures: XPath implementations are too closely tied to XML, and going back and forth between Fractal components and XML documents at run-time would be too inefficient.

II.1 Language Description

This section describes the FPath language itself. The first part (Sec. II.1.1) describes the conceptual model used by FPath to represent Fractal architectures. Then, section II.1.2 shows how path expressions on this model can be used to query these architectures.

II.1.1 Modeling Fractal Architectures as Directed Graphs

The Fractal component model is defined in relatively technical terms using a language-independent API. As a result, the different concepts it defines and their relations are not all represented in a uniform way. Some concepts have a well-defined language-level representation, like interfaces which are reified as `Interface` objects; others are defined through conventions, like configuration attributes which appear only as matching pairs of getter and setter methods.

To keep the FPath language simple and uniform, the language is based on an alternative representation of Fractal architectures using a very general model: *directed labeled graphs*. In this model, core entities in Fractal are represented by *nodes*, and their relations by directed *arcs*, the arc's *label* indicating what kind of relationship is modeled. One important feature of this approach is that, like Fractal, it is easily extensible in an homogeneous way: modeling an extended version of Fractal with new concepts not defined in the standard specification is simply a matter of adding new kinds of nodes and new labels. The FPath language itself supports this generality and extensibility, and even its syntax does not need to change to work on extensions or alternative representations of the model.

The three main Fractal concepts which are directly reified as FPath nodes are *components*, *interfaces* and *attributes*:

- Each component in a Fractal architecture, be it primitive or composite, is represented by exactly one **component node** in the corresponding graph. Although it is used in practice to identify components, the standard `Component` controller is an *interface* of the same nature that, for example, `BindingController`. Its use both for service discovery and component identification is, in a sense, purely conventional. As a concept however, the notion of *component* is obviously at the heart of Fractal, and components have thus been made first-class entities as nodes in our graph representation;

- Each interface of each component, be it internal or external, is also represented by exactly one **interface node**. This includes the `Component` interfaces, which are present in the graph as interface nodes *in addition* to the corresponding component node.

- Finally, each configuration attribute of each component is represented as an **attribute node**. Configuration attributes are discovered by introspection on the set of methods defined by concrete `attribute-controller` interfaces. In the Fractal API, attributes are only present implicitly in the naming conventions of the methods of a component's `attribute-controller`. Because they are used a lot in practice to represent configuration parameters of interest for FPath (discovering how a system is configured) and FScript (tuning or reconfiguring the system), individual attributes are represented as nodes in our graph model.

Although they are a central concept in Fractal, *primitive bindings* are not reified as nodes in our representation. This choice was made for several reasons: *(i)* bindings represent a *relation* between interfaces, and are more naturally modeled as an arc in the graph; *(ii)* representing them as nodes would add an indirection level in the graph and in every FPath query using them, for no benefit; *(iii)* in most cases, when one wants to talk about the bindings themselves one

can identify them with the client-side interface (the server side is easy to obtain from there in FPath).

Each kind of node defines a set of properties to further describe the architectural elements they represent. These properties are primitive values (for example strings or booleans), and are accessible at the language-level through unary functions on nodes named after the property. For example, each node has a *name* property, available using the `name()` function (for component nodes, the name is defined by the component's `namecontroller`). Other properties include the `state()` of components (`STARTED` or `STOPPED`), the `value()` of the attributes, and all the properties of Fractal interfaces (whether they are `client()`, or `server()`, `mandatory()` or `optional()`, etc.). The examples below will illustrate the use of many of these.

To complete the graph representation of Fractal architectures, the nodes described above are connected using directed and labelled arcs which model the structure of the architecture. The arcs are used both to connect together the different nodes which represent a given component (component nodes with their interfaces and attributes) and to model the whole architecture as relations between components and interfaces.

Figure 1 summarizes all the primitives axes defined in FPath between the different kinds of nodes. For example, it shows that component nodes have outgoing arcs labelled `attribute` to the nodes which represent their attributes ("attribute ownership" on the figure). Conversely, the attribute nodes themselves are connected to their unique owner component with an arc labelled `component`. The other primitive axes include:

- `child` (resp. `parent`), which connect component to their direct sub-components (resp. direct super-components);
- `binding`, which connects client interface nodes to the server interface they are bound to;
- and finally `interface` (resp. `internal-interface`) which connects components to their external (resp. internal) interfaces. A reverse arc labelled `component` also connects the interfaces to their owner component.

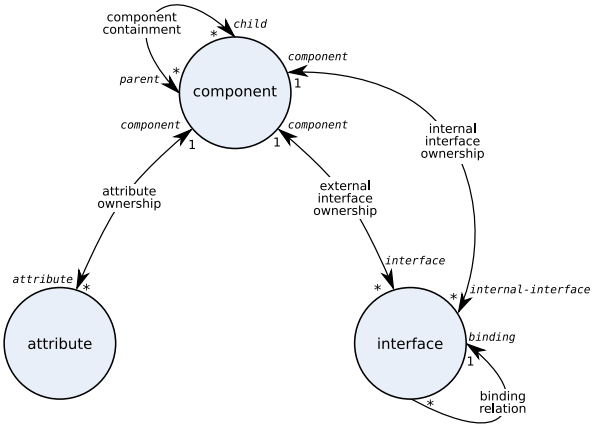


Figure 1: Default nodes and primitive axes defined in FPath.

These axes are called *primitive* because they represent all the core relationships between the elements of the model. FPath also defines *derived axes* (defined in terms of the primitive axes), which are accessible in the exact same way to the user, but enable more powerful queries:

- `sibling`, which connects component nodes to all the other component nodes which share at least one parent with them (this is useful to locate components with which one can create a connection);
- `descendant` (reps. `ancestor`) connects a component node to *all* of its sub-components (resp. super-components), direct of indirect. Formally, the relation defined by this axis between the graph nodes is the transitive closure of the one defined by `child` (resp. `parent`).
- Finally, all these derived axes have a variant named with the suffix `-or-self`, which defines the same connections as its base axis, but also connects each node to itself. This corresponds to the *reflective* versions of the relations defined by the other axes. For example, given an initial component node representing the root of an application, the `descendant-or-self` can be used to select in one step *all* the components comprising the application (including the root itself).

To conclude this section, figure 2 shows part of the architecture of the Comanche web server[1] (used later in the examples) and its representation as a graph. In order to keep the graph readable, only the first two levels of Comanche components are shown. The figure also omits interface nodes corresponding to control interfaces (showing only service interfaces), the `component` axis (which exist between every node and its owner component), and all derived axes.

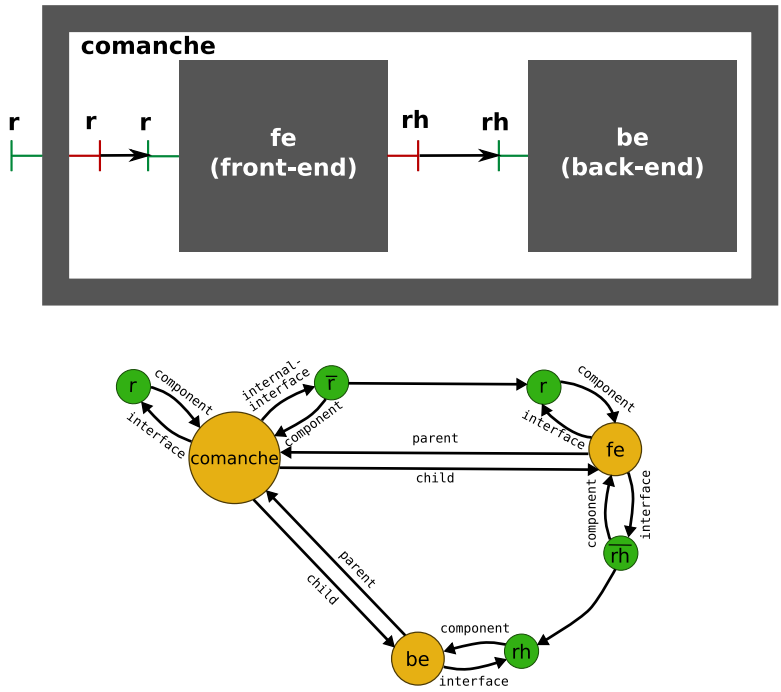


Figure 2: The architecture of Comanche and the equivalent FPath graph.

II.1.2 FPath Expressions

Given the graph representation described above, FPath expressions can be used to "walk" in the graph, starting from a initial node (or set of nodes), and successively following specific axes

[1] See `http://fractal.objectweb.org/tutorial/index.html`.

inside the graph to attain other nodes. The general syntax of such an expression is a sequence of *steps* separated by slashes:

```
step1/step2/.../stepN
```

Each step indicates the *axis* to follow, the *name* of the nodes to look for, and an optional list of predicates to select nodes more precisely:

```
axis::name[predicate(.)]
```

An example of a simple path expression would be:

```
$comanche/child::fe
```

The first part of the expression, `$comanche`, refers to the value of an FPath variable and is used as the initial node-set for the query. In this case it is assumed to denote the component node representing the root of the Comanche web server. The second part, after the slash character, is a *step* expression: `child::fe`. It tells FPath which axis to follow in the graph (here `child`), and which of the resulting nodes to selects (here, the nodes named `fe`). This simple, one-step path expression thus selects the sub-component named `"fe"` of Comanche (the server front-end).

One can also write `$comanche/child::*` to tell FPath to return *all* the sub-components, whatever their name is. In this case the value of the expression would be a node-set containing both Comanche's front-end and back-end (`fe` and `be`).

Of course, a path expression can be made of several steps, and each step can use a different axis. If one wants to find all the components which have an (external) interface named `r` (the `Runnable` interfaces in Comanche), one can write:

```
$comanche/descendant-or-self::*/interface::r/component::*
```

The first step uses the `descendant-or-self` axis, which is the transitive and reflective closure of `child`. Because the expression uses a star (`*`), this will select *all* the components in Comanche, including the root (the `-or-self` variants of derived axes always include the initial node in the result). This node-set is used as input for the second step, `interface::r`. From each of the initial nodes (here, all the components in Comanche), this will try to follow the `interface` axis to find an interface node named `r`. For some of the component, there is no such interface. This simply means they do not contribute anything to this step. For the others, the nodes representing the matching interfaces will be grouped in a single node-set. At this point, the intermediate result for the first two steps is the set of all the external interfaces named `r` present in Comanche, but what we want it the set of *components* which have such an interface. This is easily done in the third step using the `component` axis, to find all the owners of the interfaces named `r` (the name used for `Runnable` interfaces in Comanche).

Path expressions have one last feature which make them very powerful: at each step in a path, it is possible to filter the intermediate result before it is fed to the next. The filtering is done using an optional list of predicates, which can access nodes properties and can even be complete path expressions themselves.

For example, the query

```
$comanche/descendant-or-self::*[stopped(.)]
```

finds all the components in Comanche which are stopped. The first part of the query is the same as before, but with a predicate expression in brackets added at the end of the step. Here, the predicate simply invokes the `stopped()` function to each node in the intermediate result (the dot "`.`" denotes the implicit argument passed to the predicate). Instead of returning all the components in Comanche, this will return only those which are stopped.

Figure 3 summarizes the syntax of path expressions.

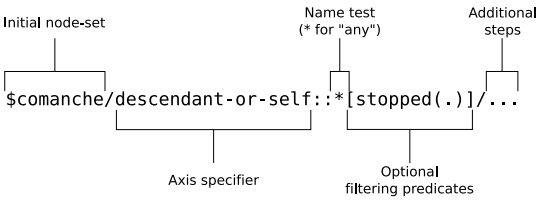


Figure 3: Path expressions syntax.

II.2 Examples of FPath Queries

This section describes more complex FPath expressions which show the power of the language as a general query language for Fractal-based systems.

Finding configuration attributes. Configuration attributes exposed by `attribute-controller` interfaces provide a convenient way to customize the behaviour of components. Complex architectures with dozens of components can expose lots of these attributes, scattered in the many composition levels of the system, and it is not always obvious what configuration parameters are available to tune an application. However, with a very simple FPath query it is possible to discover *all* the configuration attributes supported by a system:

```
$root/descendant-or-self::*/attribute::*
```

This query, where `$root` denotes the top-level component of the system (or a sub-system), will return a set of attribute nodes, each representing one of the configuration attributes of any component inside `$root` (directly and indirectly).

Checking architectural invariants. Fractal is a very flexible model, and makes it possible to reconfigure applications dynamically. However it also imposes some constraints to ensure that the resulting architectures are consistent. For example, before a component can be started, all its clients interfaces which are not optional must be bound to a compatible server interface. Such a constraint can be checked easily using the following FPath query, which will find all the components violating the constraint (and can thus not be started immediately):

```
$root/descendant-or-self::*/interface::*[client(.)][mandatory(.)][not(./binding::*)]
```

This query illustrates the use of multiple predicates. It also show how it is possible to access the properties of nodes using pre-defined functions. Here the expression uses the functions `client()` and `mandatory()` to select only client interfaces which *must* be bound (i.e. are not optional). The last predicate uses en embedded path expression to select the server interface bound to the candidate client. The predicate will be *true* if and only if the client interface is not currently bound (`./binding::*` returns an empty set, considered *false*, which is transformed in *true* by the `not()` function).

Shared components. Fractal supports component sharing, i.e. components which have multiple direct super-components. This is a powerful feature, but it can make an architecture more difficult to understand, as each of the parents try to control the shared component, requiring some coordination. Once again, FPath makes it very easy to identify all the shared components in a system which will need to be checked further:

```
$root/descendant-or-self::*[size(./parent::*) > 1]
```

This query can almost be read as is: "Starting from the system's root component, select all its descendants in the architecture, but keep only those which have strictly more than one parent." For comparison, the Java code equivalent to this simple and readable expression takes 25 lines of code.

III Programming Reliable Architectural Reconfigurations With FScript

This section describes the FScript language, which embeds the FPath query language presented above. While FPath only supports querying Fractal architectures (i.e. pure introspection without modification), FScript enables the definition of complex reconfiguration scripts which can modify the structure and configuration of a Fractal application. In FScript, FPath is used as an embedded sub-language to select the elements on which to apply the reconfigurations.

The overall design objective of FScript is to be a scripting language to express reliable architectural reconfigurations of Fractal-based systems. More precisely:

- FScript is *focused* on the manipulation of architecture-level concepts. It provides complete control of the architecture and configuration of Fractal based systems, to the full extent of what is defined by the model, but nothing else. This means all the concepts defined by the Fractal model are available in the language, both for introspection and, where supported by the specification, modification. Because its domain is limited to the architecture level ("wiring" and configuration), it also means that FScript is explicitly not designed to *implement* Fractal components. It is not a complete programming language, and does not deal with business logic. For example, although the service interfaces of components are visible and their bindings controllable, the language itself does not support the invocation of business methods.

- The second important objective is to ensure the *reliability* of the reconfigurations. When a system is dynamically reconfigured, it is critical to ensure that reconfigurations will not result in inconsistent systems. In FScript, this is guaranteed by giving a *transactional semantics* to the reconfigurations, with the standard ACID properties. Implementing our own language instead of reusing an existing one, makes it possible to integrate the runtime controls required to offer these guarantees in a completely transparent way for the FScript programmer. The restricted power of expression of the language compared to general-purpose languages also allows in the future the use of static analyses to validate reconfigurations before they are actually executed.

- Finally, a secondary but important objective for FScript is to be able to support any extension to the Fractal model seamlessly, i.e. without requiring changes to the core language. This constraint is already visible in the design of the FPath model and syntax, which supports the definition of new kinds of nodes and axes.

The rest of this section first describes the different constructs of the language (Section III.1). More involved reconfigurations are then described in section III.2. Finally, the section concludes with a description of the reliability guarantees offered by FScript, and an overview of the techniques used to implement them (Section III.3).

III.1 Language Description

This section describes the different constructs of the language. The language's structure is that of an imperative, scripting-like language with a syntax mostly taken from the C family of languages, except for FPath expressions. This syntax was chosen for its simplicity and familiarity, making it easier for most users to learn the language.

The next sub-sections describe how FScript programs are structured (Section III.1.1), then show the different control structures available to program the reconfigurations (Section III.1.2), and finally which primitive (pre-defined) reconfiguration actions are available (Section III.1.3).

III.1.1 Procedures Definitions

An FScript program is made of a sequence of top-level procedure definitions, which can be either *functions* or *actions*. Moreover, FScript has a dynamic type checking system where the types of procedure arguments and return values are checked at invocation time relating to the procedure signatures. Functions can only use introspection features from FPath and are hence guaranteed to be side-effect free. They can be used in FPath expression (for example in predicates). Actions on the other hand are allowed to modify the target architecture. Concretely, the body of a function can use only other functions (be they predefined FPath functions or user-supplied), while actions can make use of functions and other actions (primitives or user-defined).

The only syntactic difference between functions and actions appears in their definitions. Functions are introduced by the keyword `function` while actions use the keyword `action`:

```
/* Locate the request scheduler in a Comanche instance. */
function scheduler(comanche) {
  return $comanche/child::fe/child::s;
}

/* Replace the request scheduler of a Comanche instance. */
action replace-scheduler(comanche, newSched) {
  prev = scheduler($comanche); // Invoke user-defined function.
  replace($prev, $newSched);   // replace() action defined below.
  return $prev;
}
```

The first procedure is a function named `scheduler()`; it takes a single argument named `comanche` which should represent the top-level component of a Comanche instance. The function simply evaluates and returns the value of an FPath expression. As FPath can only introspect components but can not modify them, the whole procedure can be declared as a safe `function`. Once loaded in an FScript interpreter, it will then be usable in other procedures (both functions and actions), or in FPath queries.

The second procedure, `replace-scheduler()`, is declared as an action, and can thus use any other procedure, including functions (as in the first line) and other actions (as in the second line). Here, the action first locates the current scheduler component of the Comanche instance passed in parameter. It then invokes a generic reconfiguration action named `replace()` to do the actual replacement.

An important property of reliable reconfiguration is that they should always terminate in finite time. One of the ways FScript guarantees this is by forbidding recursive definitions of user-defined procedures (either direct or indirect).

III.1.2 Control Structures

FScript procedures support a limited set of control structures so that it is possible to ensure that they will eventually terminate. In addition to simple sequencing of instructions (terminated by semi-colons ";"), the control structures are:

Variables Assignment. New local variables can be created inside the body simply by assigning them an initial value. Their values can be changed by re-assigning them. For example the function `bound-to()` defined below introduces a new local variable named `$servers` to make the code easier to follow. The second function, `bindings-to()`, also introduces a local variable `$itfs` but then changes its value before returning it.

```
/* Tests whether the client interface $itf is bound to
 * (an interface of) component $comp.
 */
function bound-to(itf, comp) {
  servers = $itf/binding::*/component::*;
  return size(intersection($servers, $comp/component::*)) > 0;
}

/* Finds all the client interfaces which are bound to the
 * component $comp.
 */
function bindings-to(comp) {
  itfs = $comp/parent::*/internal-interface::*[bound-to(., $comp)];
  itfs = union($itfs, $comp/sibling::*/interface::*[bound-to(., $comp)]);
  return $itfs;
}
```

Conditionals. Conditional execution is supported using the standard C syntax. For example, the following action uses the life-cycle state of a component, accessible through the `started()` *function* to make a choice:

```
/* Makes sure that $dest is in the same lifecycle state as $src.
 * Assumes the standard lifecycle (STARTED, STOPPED).
 */
action copy-lc-state(src, dest) {
  if (started($src)) {
    start($dest);
  } else {
    stop($dest);
  }
}
```

Iteration. FScript supports a restricted form of iteration, which ensures that the execution will always terminate. The `for` loop executes a given block repeatedly with a local iteration variable successively bound to each element in a node-set (which is always finite):

```
/* Copies the value of all the attributes of component $src
 * to the attributes of the same name in component $dest.
 */
action copy-attributes(src, dest) {
  for oldAttr : $src/attribute::* {
    newAttr = $dest/attribute::*[name(.) == name($oldAttr)];
```

```
    set-value($newAttr, value($oldAttr));
  }
}
```

The iteration expression (here `$src/attribute::*`) must evaluate to a node-set.

Explicit return. At any point during its execution, a procedure can stop its execution and immediately return to the caller (optionally yielding a value) using a `return` statement.

III.1.3 Primitive Actions

All the standard reconfiguration operations defined in the Fractal specification are available in FScript as primitive, pre-defined actions:

Content. The operations defined in the standard `contentcontroller` interface are available in FScript as actions `add()` and `remove()`. They correspond respectively to the `addFcSubComponent()` and `removeFcSubComponent()` methods.

Because FScript is imperative and not object-oriented, each of these actions take two arguments instead of one. For example, the FScript code `add($parent, $child)` adds the component denoted by the `$child` (a component node) into the composite denoted by `$parent` (also a component node). It corresponds to the following Java code

```
Fractal.getContentController(parent).addFcSubComponent(child);
```

where `parent` and `child` are objects of type `Component`. The `remove()` action is similar, except that `remove($parent, $child)` invokes the `removeSubComponent()` control method.

Bindings. Bindings between interfaces can be controlled in FScript using the `bind()` and `unbind()` actions. They correspond respectively to the `bindFc()` and `unbindFc()` methods of the `bindingcontroller`.

`bind()` takes as arguments the client and server interfaces to connect, in this order. This is slightly different than in the Java API where the client interface is referenced by name. In Java, the receiver object of the `bindFc()` method and the interface name are used to identify the client interface, but FScript uses directly the interface node. `unbind()` takes only the client interface to disconnect as an argument.

Life-cycle. The default component life-cycle defined by the default `lifecyclecontroller` can be controlled using the actions `start()`, `stop()`. Each take a single argument denoting the component to start or stop.

Some reconfiguration operations in Fractal require that the components implied are stopped. When such an action is called in an FScript reconfiguration, the FScript interpreter makes sure they are stopped before executing the action, without needing an explicit call to `stop()`. They are restarted automatically at the end of the reconfiguration. This feature simplifies a lot the task of the programmer, who does not have to test the components' state at each step and keep track of which must be restarted.

Attributes. The values of configuration attributes defined in `attributecontroller` can also be changed in FScript, using the `set-value()` primitive action. It takes two arguments: the attribute itself, as an FPath attribute node, and the new value to set.

Name. The name associated to Fractal components through their `namecontroller` can be modified using the `set-name()` primitive action: `set-name($component, "newName")`.

Component creation. Finally, FScript also supports the creation of new components through the action called `new()`. It corresponds to the `newComponent()` method of Fractal ADL's `Factory` interface. In its simplest form it takes a single string argument, which must be the full name of a Fractal ADL component definition. Some component definitions in Fractal ADL are parametrized, and the parameter values must be supplied to instantiate them. This is also supported by a second form of `new()`, where parameter names and values are supplied as additional parameters. Because FScript does not support complex data structures like arrays of dictionaries, the pairs of parameter names and values are simply put in sequence.

```
// Simple form
comanche = new("comanche.Comanche");
// Second form with parameters.
adl = new("org.objectweb.fractal.adl.BasicFactory", // Template name
          "fractaladl.backend",                     // Parameter name
          "org.objectweb.fractal.adl.JavaBackend"); // Parameter value
```

In addition to these pre-defined actions, new primitive actions can be easily added to reflect new customized Fractal operations and controllers. For instance, to deal with Fractal RMI bindings, two new primitive actions have been developed to bind and unbind component references in a Fractal RMI registry.

III.2 Sample Reconfiguration Scripts

This section describes in more details two complex FScript programs, which illustrate all the constructs of the language and how they can be combined to create complete reconfiguration scripts.

III.2.1 Adding a Cache to Comanche

The Comanche web server already presented above is designed to be very simple. In particular, it does not include a page cache, and needs to re-read files' content on each request. Because of its component-based architecture however, it is easy to create a new cache component and to introduce it at the right place in the architecture. Thanks to Fractal's dynamicity, it is even possible to do so while the server is running.

The following FScript program does just this. It is split in two actions:

- `get-or-create-cache()` takes in argument the Comanche request handler (a sub-component of the server's backend). This is the composite which includes the different handler components (for example the file and error handler) and a dispatcher which sends the requests to all the available handlers in turn until one accepts it. The action first looks inside the handler composite to see if an instance of the cache component (named `"cfh"`) is already there. If it is found, it is returned immediately. Otherwise, a new cache component in instantiated using the `new()` action, renamed `"cfh"`, and placed inside the handler.

- The `enable-cache()` action itself first tests whether the cache is already installed by following a binding on the dispatcher component. If the cache is not installed, it removes the direct connection from the dispatcher to the file handler, uses the `get-or-create-cache()` to obtain a cache, and inserts it between the dispatcher and the file handler. One this is done, the next requests sent by the dispatcher will be intercepted by the cache, which can

either answer them directly if it has the file content in memory, or use its connection to the original file handler component to read (and store) the content otherwise.

```
action get-or-create-cache(handler) {
  if ($handler/child::cfh) {
    return $handler/child::cfh;
  } else {
    cache = new("comanche.CachingFileRequestHandler");
    set-name($cache, "cfh");
    add($handler, $cache);
    return $cache;
  }
}

action enable-cache(handler) {
  dispatcher = $handler/child::rd;
  if (not($dispatcher/interface::h0/binding::*/component::cfh)) {
    unbind($dispatcher/interface::h0);
    file-handler = $handler/child::frh;
    cache = get-or-create-cache($handler);
    bind($dispatcher/interface::h0, $cache/interface::request-handler);
    bind($cache/interface::handler, $file-handler/interface::rh);
    start($cache);
  }
}
```

III.2.2 Replacing a Component (hot-swapping)

The last example action is a more complex – and useful – one, which reuses some of the procedures defined earlier. It takes two components as arguments, and replaces the first one with the second everywhere it appears in the architecture. It also reproduces the configuration (attributes) and state (started or stopped) of the original into the replacement component.

The structure of the action is easy to follow:

1. First, it adds the new component inside all the parents of the old one. This step must be done before the next ones to avoid the creation of non-local bindings.

2. Next, it updates all the bindings which involved the old component with an equivalent binding with the new one. Updating the bindings *from* the old component (step 2.1 in the code) is trivial, using a simple FPath query to loop on the client interfaces which are bound. The action simply has to remember to take a reference on the original server interface (the `server` variable in the code) before destroying the original binding. Updating the bindings which pointed *to* the original component (step 2.2) is a little more involved since it is not possible to directly follow bindings "backwards". Finding all the relevant client interfaces is done in the `bindings-to()` function which was explained earlier. Thanks to the power of FPath, this was done in 3 lines. Once this task is abstracted away in a function, using the resulting set of client interfaces in `replace()` is actually even simpler than updating the bindings from the old component.

3. Then, the actions reproduces the configuration and state of the old component into the new one using previously defined actions `copy-attributes()` and `copy-lc-state()`.

4. Finally, the action can completely remove the old component from the architecture, since it is not connected to it anymore.

```
/* Replaces $oldComp with $newComp everywhere in the architecture. */
action replace(oldComp, newComp) {
  // 1. Add the new component everywhere the old one is present.
  for p : $oldComp/parent::* {
    add($p, $newComp);
  }
  // 2. Update bindings involving oldComp to use newComp instead:
  // 2.1. Bindings *from* oldComp
  for client : $oldComp/interface::*[client(.)][bound(.)] {
    itfName = name($client);
    server = $client/binding::*;
    unbind($client);
    bind($newComp/interface::$itfName, $server);
  }
  // 2.2. Bindings *to* oldComp (uses bindings-to() defined above)
  for client : bindings-to($oldComp) {
    itfName = name($client/binding::*); // Match names on the server side
    unbind($client);
    bind($client, $newComp/interface::$itfName);
  }
  // 3. Reproduce configuration and state
  copy-attributes($oldComp, $newComp);
  copy-lc-state($oldComp, $newComp);
  // 4. Remove the old component
  for p : $oldComp/parent::* {
    remove($p, $oldComp);
  }
}
```

Comparison with Java. FScript as a domain specific language has not only a more readable syntax but it is also more compact than a general purpose language like Java for programming reconfigurations in Fractal. The complete definition of the `replace()` action above, including the support procedures defined earlier (`bound-to()`, `bindings-to()`, `copy-lc-state()` and `copy-attributes()`) and comments takes 68 lines. The equivalent Java code takes 114 lines, an increase of about 67%. The difference may not seem that large, except for the fact that the Java version uses a helper method to deal with attributes in a generic way[2], and most importantly the total of 114 lines does not include *any* error handling: exceptions are not simply caught and ignored, they are not considered at all.

Not surprisingly, the biggest gain in terms of size is found in the `bindings-to()` equivalent, which uses relatively sophisticated FPath expressions. The FScript version is 5 lines long and could easily be made shorter, even to the point of simple one-liner. The Java version however takes 26 lines (an increase of 420%, again with no error handling), and includes up to 5 levels of imbrication, 3 of which are `for` loops, making the code very difficult to understand.

Table I gives the detail, procedure-by-procedure. The number of lines in the table do not include comment lines.

There is no explicit error handling construct in FScript, but this is not because errors are ignored. On the contrary, it is because error detection and correction is handled completely automatically and transparently to ensure the reliability of the reconfigurations and of the target applications.

[2]The `AttributesHelper` class (provided by FScript) is about 450 lines long, although only part of the features are used here.

Procedure	LOC in FScript	LOC in Java	Increase
`bound-to`	4	7	×1.4
`bindings-to`	5	26	×5.2
`copy-lc-state`	7	10	×1.42
`copy-attributes`	6	7	×1.16
`replace`	21	37	×1.76
Total	43	86	×2

Table I: Code size comparison between FScript and Java.

The next section shows how this is done using a powerful and uniform approach which handles several kinds of errors in a completely transparent way for the FScript programmer.

III.3 Ensuring The Reliability of Reconfigurations

Reliability (a key concept of dependability [5]) is a main problem in systems subject to dynamic reconfigurations, especially with open models like Fractal because reconfigurations can be unanticipated at compile time and consequently, static analysis of reconfigurations is difficult. Indeed dynamic reconfigurations may be invalid and leave a system in an inconsistent state, i.e. no more available/usable from a functional point of view. FScript aims to guarantee the reliability of reconfigurations with automatic and transparent error detection and correction thanks to a transactional semantics for reconfigurations.

III.3.1 FScript Reconfigurations as Transactions

A reconfiguration script must be a consistent transformation of the system, however it is possible to specify invalid actions in FScript. For instance, the following action is invalid:

```
action strictly-invalid(c1, c2) {
  add($c1, $c2);
  add($c2, $c1);
}
```

Whichever application it is applied to, it will fail because this would create a cycle (*$c1* would end up containing itself), which is forbidden by the Fractal model. Some actions can be valid only on some specific system architectures, i.e. with some preconditions on components of the system:

```
action conditionally-valid(c1, c2) {
  add($c1, $c2);
  bind($c1/internal-interface::foo, $c2/interface::bar);
}
```

To be valid, this action requires that:

- component *$c1* is a composite component
- *$c1* and *$c2* have respectively a *foo* and a *bar* interface
- *foo* and *bar* interfaces are type compatible

Executing these invalid actions in a system would violate its consistency, so we use transactions to be able to recover the system from failed reconfigurations. The execution backend of the FScript interpreter aims to ensure the reliability of dynamic, distributed and concurrent reconfigurations in the Fractal component model. This backend consists essentially on an extension of the Julia [7] implementation of the Fractal model in Java associated to a transaction manager dedicated to dynamic reconfigurations. We use a flat transaction model for managing reconfiguration transactions. Several more complex transaction models have been defined for specific applicative domains [37] but flat transactions have proven to be sufficient and efficient in applications such as databases where transactions are relatively short-lived and the number of concurrent transactions is relatively small and the transactional processing system is centralized. Dynamic reconfigurations we consider appear to satisfy these hypotheses. Indeed, reconfiguration are essentially short-lived FScript actions, and the level of concurrency for reconfigurations should be moderate: the system is not constantly reconfigured, otherwise it would reduce its availability. Moreover, even if reconfigurations can be distributed, our transaction processing system is rather centralized. This model appears sufficient for short-lived reconfiguration transactions and a the moderate level of concurency we consider here. FScript reconfigurations are automatically demarcated by the FScript frontend, then each top-level FScript action and function is executed as a separated transaction (sub-actions/functions are inlined as explain in the sequel).

ACID properties in the context of reconfigurations can improve reliability by making systems fault-tolerant i.e., compliant with the specification in spite of faults due to dynamic reconfigurations. These properties are unifying concepts of transactions for distributed computation [36] used for supporting concurrency, recovery, and guaranteeing system consistency. The definition of the ACID properties in the context of dynamic reconfigurations in component-based systems is as follows:

- **Atomicity**. Either the system is reconfigured and the reconfiguration transaction commits (all the operations forming the transaction are executed) or it is not and the transaction aborts. If a reconfiguration transaction fails, the system comes back in a previous consistent state as if the transaction never started.

- **Consistency**. A reconfiguration transaction is a valid transformation of the system state i.e. it takes the considered system from a consistent state to another consistent state. A system is a consistent state if and only if it conforms to our consistency criteria: it does not violate the component model and possibly more specific constraints.

- **Isolation**. Reconfiguration transactions are executed as if they were independent. Results of reconfiguration operations inside a non-committed reconfiguration transaction are not visible from other transactions until the transaction commits ; or never if the transaction aborts. It must be noticed that synchronization between the functional and the non-functional level to reach a “quiescent state" (i.e. a stable state) [22] currently relies on the implementation of the *LifeCycleController* as in Julia [7].

- **Durability**. The results of a committed reconfiguration transaction are permanent: once a reconfiguration transaction commits, the new state of the system (both the architecture description and the component state) is persisted so that it can be recovered in case of major failures (e.g., hardware failures).

Each ACID property and associated mechanisms will be detailed more precisely in the next sections.

III.3.2 Atomicity of Reconfiguration Transactions

The execution of a reconfiguration transaction, which corresponds to a top-level FScript action or function, boils down to the execution of primitive reconfiguration operations (in Fractal controllers) i.e. *introspection operations* without side effects, and *intercession operations* that modify the system. In our approach, to ensure atomicity of reconfiguration transactions, operations performed in transactions which abort are undone. Actually, only intercession operations need to be taken into account as they are the only operations modifying the system, functions are considered as read-only actions which must be isolated. When operations are reversible, undoing a reconfiguration transaction is equivalent to sequentially undo primitive intercession operations (in reverse order) in a compensation transaction. The undo model is linear and there is no permutation between operations and then the commutativity of operations is not required. On the other hand, it is possible to define non-reversible intercession operations but they need a customized treatment during the transaction rollback. Compensation operations can be associated to these operations but without any guarantee on the atomicity since the resulting state of a rollback can be not exactly the same as the initial one, but the system consistency is ensured.

An extensible library of primitive operations from the Fractal API is proposed where each intercession operation is associated to its inverse intercession operation (cf Figure 4) in the semantics we choose (typically *bindFc* and *unbindFc* for two component interfaces are inverse operations). Moreover, an operation can keep a state so as to be undone (e.g., changing the value of an attribute requires to keep the old value of this attribute). When a transaction rollback occurs, a notification is send by means of a *RollbackException* caught by the reconfiguration client.

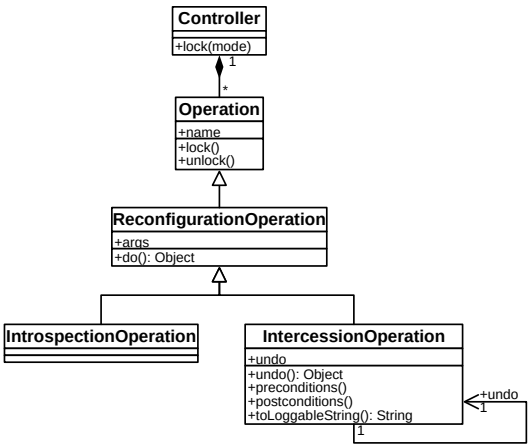


Figure 4: Reconfiguration operation model.

The target component architecture is managed as a single transactional resource with only one centralized transaction manager dedicated to dynamic reconfigurations. Then the transaction manager can use internally a One-Phase-Commit protocol [1] whitout a voting phase since there is no need of synchronization between several transactional resources and the manager has only to coordinate its own modifications, even if the system architecture is distributed. Nevertheless, the transaction manager also supports the Two-Phase-Commit protocol [36] so as to be able to participate in global transactions involving several transactional resources, then it can be coordinated as a distributed transaction participant by another transaction manager.

By default, as any operation could potentially lead the system to an inconsistent state,

they must always be included in transactions (if there is no transaction, a new one is created). Transaction demarcation is automatic for FScript actions and functions (no explicit language demarcation is needed), they are always included inside a transaction. As there is no nested transaction, two nested actions or functions are always executed in the same transaction. A top level action is always composed of a single transaction and nested actions are then linearised. It corresponds to the FScript semantics in which actions encapsulated in other actions are simply inlined in the top-level actions.

III.3.3 Integrity Constraints to Ensure System Consistency

In our proposal, system consistency relies on integrity constraints and we want to express these constraints both on the component model and on applications. An *integrity constraint* is a predicate on assemblies of architectural elements and component state [23]. Therefore, a reconfiguration transaction can be committed only if the resulting system is consistent i.e., if all integrity constraints on the system are satisfied. To express constraints including invariants, preconditions and postconditions, we use FPath as a constraint language "à la OCL" [28]. Some advantages of FPath are that:

- it can navigate and select Fractal elements at runtime with just introspection capacities,
- it is based on a simple graph representation of the system during execution which can be easily extended with new concerns (e.g., new relations can be added between Fractal elements),
- its powerful syntax allows to specify both generic constraints for all elements in the system by using the star token and recursivity, and specific constraint where Fractal elements are designated by their name.

We distinguish three different levels of constraint specification (cf. figure 5) corresponding to three different abstraction levels.

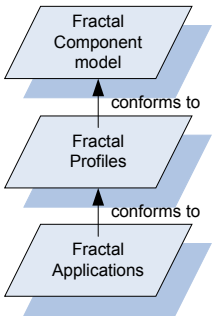


Figure 5: The three levels of integrity constraint.

The model level This a set of generic constraints associated to the component model. These constraints apply to all instances of some elements of the Fractal model. Examples of such constraints are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid recursion). We specified the semantics of reconfiguration operations in the model with preconditions and postconditions which should never be violated. An example of such a precondition for the remove operation is to check that all interfaces of the component to remove are unbound:

```
Fractal API: void removeSubComponent(Component child);
// preconditions:
not(\$child/interface::*[bound(.)]);
...
```

The profile level This is a set of generic constraints to refine the component model for a family of applications. The profile level conforms to the model level. A profile may for instance forbid component sharing in applications with the constraint `size(./parent::*)<=1` which says that every component cannot have more that one super-component.

The application level Application constraints are specific to a given architecture and apply directly to instances of Fractal elements designed by their names. The application level conforms to the profile and to the model. Invariants can concern cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc. Application constraints are specified in Fractal ADL with the *constraint* tag. For instance, we may want to add a structural invariant on a simple component *ClientServer* (Figure 6) saying that it must always contain only one component providing the *service* interface, it will be specified in the ADL definition of the component as follows:

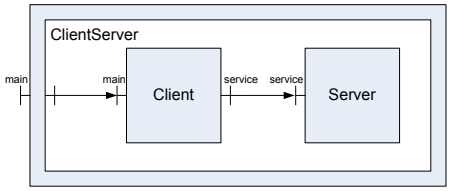


Figure 6: A simple client-server Fractal application

```
<definition name="ClientServer">
 <interface name="main" role="server" signature="java.lang.Runnable" />
 <component name="client">
  <interface name="main" role="server" signature="java.lang.Runnable" />
  <interface name="service" role="client" signature="Service" />
  <content class="ClientImpl" />
 </component>
 <component name="server">
  <interface name="service" role="server" signature="Service" />
  <content class="ServerImpl" />
 </component>
 <binding client="this.main" server="client.main" />
 <binding client="client.service" server="server.service" />
 <constraint value="size(./child::*[./interface::service])=1" />
</definition>
```

Constraints must be checked both at compile time on the component static configuration and at runtime and only new application constraints can be dynamically added or removed by means of a *ConstraintController*. There is currently no check of conflicts between constraints. Pre/post-conditions of primitive intercession operations are checked at each operation execution whereas invariants are only checked at commit of transactions. That is to say a system can temporarily violate invariants during a transaction but it must be in a correct state after commit. When it is detected, a constraint violation makes the transaction involved rollback.

III.3.4 Isolation of Reconfigurations to Support Concurrency

Several administrators (humans or machines) may want to reconfigure the same system at the same time or a single reconfiguration may be composed of parallel reconfigurations to optimize the reconfiguration process. There are two available scheduling policies adapted to the level of concurrency for reconfigurations in Fractal applications: either reconfigurations are concurrently executed and accesses to Fractal elements in the system must be synchronized or reconfigurations are serially executed, i.e., only one reconfiguration is executed at a time while others are simply queued.

For concurrency management, we adopt a pessimistic approach with strict Two-Phase locking (2PL) [36] to provide strong concurrency (corresponding to *serializable* ANSI isolation level). In 2PL, locks acquired during a transaction are only released when the transaction is committed to avoid *phantoms* (lost updates). Several elements in the model corresponding to FPath nodes are lockable with a hierarchical algorithm: for instance, a lock acquisition on a component for instance will also lock all its interfaces. Locks are re-entrant (a transaction can acquire several times the same lock) and there are basically two types of locks: read (or shared) locks, and write (or exclusive) locks. A *LockController* is provided with every component to lock its sub-elements, locks can be also acquired by means of an FPath expression.

We use this locking model at a finer granularity to avoid inconsistent state for reconfiguration operations based on their semantics. Basically, an introspection operation in a transaction will take read locks on other reconfiguration operations to avoid *dirty reads* and *fuzzy reads* (non repeatable reads): it should not see modifications in the system due to other transactions which have not been committed yet. However, two operations in two different transactions can introspect the same element in the system by sharing a read lock. On the other hand, intercession operations on a given element will take write locks to avoid conflicts between state modifications: other transactions can neither introspect the same element nor modify it. For example, if we want to add a component B in a component A, it will write lock the following operations in the *ContentController* of the parent (A): *addFcSubComponent*, *getFcSubComponents* and *removeFcSubComponent*. The *getFcSuperComponents* operation in the *SuperController* of the child (B) will also be write locked.

III.3.5 Durability of Reconfigurations to Support Recovery

Transactions are units of recovery, so for every transaction, demarcation and primitive intercession operations are logged in a journal so that it can be undone in case of rollback. The journal is kept both in memory and persisted on disk and it can be redone in case of software or hardware failure. We use WAL (Write Ahead Logging) [36] with checkpointing which means that all modifications are written to the log before they are applied in the system and both redo and undo information is stored in the log. In case of major failure such as a hardware crash, the Undo/Redo recovery protocol is applied by the Transaction Manager: all transactions which are not committed are canceled and all committed transactions since the last checkpoint are redone. The log format currently follows the FPath syntax and Fractal elements in arguments of operations are designed with one absolute path in the system architecture (path is not unique due to sharing).

In addition to the journalization of transactions, the system state is periodically checkpointed. The state of a component-base system which is considered here is its architectural description and the set of its component state. Checkpointing at commit time allows to recover any component in its last known consistent state resulting from the last successful reconfiguration. A component architectural state is checkpointed by a Persistence Manager with Fractal ADL dumps, the ADL dumper is extensible so as to consider eventual extensions of the ADL language. Thus some

new state information can be included in the ADL such as for instance the lifecycle state of components. The functional state which is considered for a component is basically the set of its attribute values of components. Attribute values which are of primitive types are automatically saved in ADL dumps. There are two implementations for other attribute values which are Java Object: either a Java serialization in a file named with an absolute path of the component or they can be saved in a database if an Object Relational Mapping is defined for the component attributes (our implementation uses Hibernate [3]). In case of recovery, the component state will be updated automatically with its last available persisted state. An extension of the attribute module in Fractal ADL allows to define which attributes are persistent. Regarding the functional state persistence, state transfert is the responsibility of the *StateController* for each component.

III.3.6 An Instrumented Fractal Implementation for Reconfiguration Transactions

The Julia implementation of the Fractal model has been extended to provide a transactional semantics to Fractal reconfiguration operations. It must be noted that this implementation does not depends on FScript so that reconfigurations which are programmed only with the Fractal API can also benefit of transactional properties by means of an explicit transaction demarcation. New controllers have been added to the standard Fractal controllers to implement the ACID properties:

- The *ConstraintController* is used to set and get integrity contraints to specify the component consistency. It is bound to a *Consistency Manager* component for constraint checking before validating a transaction.
- The *LockController* is used for concurency management to lock the component, its interfaces and operations. It interacts with a *Concurrency Manager* component which implements the locking protocol and notably keeps a waiting graph of transactions.
- The *StateController* allows to get and set the functional state of the component, i.e its architecture description and its functional state. A *Persistence Manager* component is responsible to get and serialize the architectural and functional state of components involved in a transaction.
- The *TransactionController* checks preconditions and postconditions of operations and registers the invoked operations in the journal of the *Transaction Manager*.

Moreover, an interceptor is added to each standard reconfiguration controller so as to notify the transaction manager when an operation is invoked on the control interface. It acts as a proxy for each operation of the controller and it reports any error which could occur during the execution of the operation and lead to a cancellation of the transaction. For example, a reconfiguration operation m() is intercepted before and after its execution while exceptions are caught and registered in the Transaction Manager:

```
void m () {
  try {
    checkPreconditions();
    register(m, PRE);
    impl.m();
    checkPostconditions();
    register(m, POST);
```

[3] http://www.hibernate.com

```
  } catch(Exception e) {
     register(m, FAILED, e);
  }
}
```

When an exception is thrown by the code of the Fractal operation or if the check of pre/postconditions fails, the reconfiguration is rollbacked and the FScript interpretor is notified of the error.

IV Related Work

Navigation and query languages. OCL [28] (Object Constraint Language) is a standardised language used by the OMG in UML and related technologies. It is used primarily to annotate UML models with model-specific constraints (for example pre and post-conditions on operations), but can used for navigation and query. OCL provides a rather concise and readable syntax, and has good support to handle collections of elements (filtering, set operations...). However, the language can not navigate along transitive relations like FPath does with derived axes (`descendant` for example). This means the "depth" of a query must be fixed and hard-coded, limiting the usefulness of queries in large or unknown architectures.

OQL [9] (Object Query Language) is a declarative language derived from SQL92 and defined by the ODMG. It includes several extensions to SQL to support Object Databases, including complex objects with attributes and references to each other, path expressions, operation invocation and a rich collection of data structures (sets, bags...). Contrary to OCL, it supports navigation to arbitrary depth using a built-in function `reachables()` which finds all the objects reachable from a given source. However, this operation is not very selective and returns all the reachables at once. Compared to FPath, it would correspond to following *all* the axes at the same time, and letting the user write the appropriate predicates to restrict the result afterwards. Compared to FPath, OQL is a much more powerful (and complex) language, but its very generality makes it less adapted to our specific domain (Fractal architectures), where FPath provides tailored support to the relevant concepts.

Logic programming, in the form of Prolog or Datalog [10] or languages inspired by them like [19], can also be used as query languages. Graph structures like the one used by FPath are encoded naturally in logic terms. This family of languages have the advantage of begin formally defined and based on rigorous mathematical theories. As for SQL-derived languages however, their syntactic structure can make them difficult to integrate with other languages. We are currently using Datalog as part of our work on formalising the semantics of FPath and FScript. As Datalog is more powerful than FPath, this may lead to extensions in future versions of FPath.

The version of FPath presented in this article was inspired by XPath 1.0 [38], but an updated version of XPath [39] has recently been standardized. Some of the new features, like existential and universal quantifiers, might be interesting to add to FPath, especially when FPath is used to express architectural constraints (as for example in section III.3.3 to check for consistency).

[3] discussed infrastructure support in Fractal for sophisticated querying of the system architectures. Its scope was larger than FPath's, as it also included the querying of component repositories in addition to run-time components. The underlying model was also based on a directed graph representation of Fractal components which is very similar to the one used in FPath. However, this work was more concerned with infrastructure issues (how to build a query service), and less on the language aspect of queries. To our knowledge, the query language proposed in the paper was never fully implemented.

Component configuration and reconfiguration. Several component models support dynamic reconfiguration but most of them however do not provide direct language support for these reconfigurations and rely directly on the implementation programming language instead (OpenCOM [11], K-Component [16]). Also, few of them take into account the reliability of the reconfigurations. Instead, most work on reliability and validation of component-based architectures is done in the context of static Architecture Description Languages (ADLs) [25]. Wright [4] use a process algebra to ensure that there will be no deadlocks in components interactions. C2 [24] can check the conformance of an architecture description relative to *style constraints*. However, as identified by [29], ADLs are not enough. To support dynamic architectures one also need what the author calls an *Architecture Modification Language* (AML) to describe modification operations and an *Architecture Constraint Language* (ACL) to describe the constraints under which architectural modifications must be performed. In the Fractal ecosystem, Fractal ADL[4] already fills the role of a classic (although extensible) ADL to describe initial/static configurations, while FScript and FPath complement it by filling these two additional roles (AML and ACL, respectively).

One interesting work which supports dynamic reconfigurations while still offering strong guarantees is ArchJava [2], which extends Java with component-based concepts. ArchJava architectures can be reconfigured dynamically and the language guarantees communication integrity during execution. However, these guarantees are only possible because the reconfigurations are hard-coded in the program. This is an important limitation because in practice adaptations required during the lifetime of an application can rarely be predicated at the time it is built. Mae [33] proposes an architecture evolution environment using xADL [12], it has another approach than ours to reconfigure applications as it is goal-based oriented: an ADL configuration is given as an objective and the difference with the current configuration is automatically performed, the resulting patch is applied on the system.

More recently, component models relying on reflexive architectures to allow non anticipated (also called ad-hoc) reconfigurations while supporting some kinds of guarantees have appeared. FORMAware [27] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is that our integrity constraints are more flexible than styles as they can be applied at the model level or directly to specific instances with pre/post-conditions and invariants. Plastik [6] is the integration of the OpenCOM component model and the ACME/Armani ADL [18]. As in our approach, architectural invariants can be checked at run time and constraints are expressed at two levels (style and instance). However we propose a full support of concurrency for reconfiguration. We define more formally reconfiguration operations to identify conflicts between them so that our locking algorithm is finer than simple locking components, moreover it also considers introspection operations with shared locks, notably so as to avoid dirty reads.

V Conclusion & Future Work

This paper has presented two related (and complementary) languages which make it easier to manage Fractal architectures.

FPath is a general query language for Fractal applications inspired by XPath [38]. It provides a convenient notation to navigate inside Fractal architectures and select elements based on their properties or location in the architecture. FPath is based on a uniform and extensible representation of Fractal architectures as directed graphs, and although its syntax is relatively simple it

[4] `http://fractal.objectweb.org/fractaladl/index.html`

supports sophisticated queries. FPath was originally designed as a part of FScript, but can also be used by itself as a general addressing language for Fractal. FPath has been used in [14] and [20] to write the condition part of Event-Condition-Action rules in these autonomic extensions to Fractal. It is also used in [30] as a point-cut language for an Aspect-Oriented extension of the Fractal model, and in [34] to define and verify architectural invariants on Fractal architectures.

FScript relies on FPath for its addressing needs, but supports the definition of architectural reconfigurations of Fractal architectures. It is modeled as a simple scripting language with an imperative flavor to make it easy to learn by potential users. FScript supports all the reconfiguration features defined in the standard Fractal model, including structural (bindings and containment), state (attributes) and life-cycle reconfigurations. Like FPath, it is extensible along the same ways than the Fractal model itself. Beyond the convenience of a custom syntax, the main feature of FScript is that its constructs and run-time semantics have been designed to make reconfigurations *reliable*: its restricted (focused) power of expression enable us to guarantee that FScript reconfigurations terminates and have a *transactional semantics*, and hence keep the target system in a consistent, usable state. This makes it possible to apply unanticipated reconfigurations to running systems in a way that minimizes disruption and maximizes the continuity of service. FScript [5] has been used as part of the Safran extension to Fractal [13, 14] to program the reaction part of Safran's autonomic rules which can automatically adapt a Fractal architecture to context changes. [20] also uses FScript for similar purposes. Finally, [31] has shown how FScript can be used with the Think [17] implementation of Fractal (targeted at Operating System kernels) to take advantage of Think's reconfiguration features.

A *formal specification* of FPath/FScript is currently being defined. This should enable to provide *static analyses* of FScript programs in order to detect some kinds of errors earlier [15]. This would allow us to reject certain kinds of invalid reconfigurations before actually modifying the system. This formal definition will also be used as a basis for a new, more sophisticated implementation of FPath and FScript, which will include optimisations (something completely missing from the current interpreter).

References

[1] Abdallah, M., Guerraoui, R., and Pucheral, P. One-phase commit: Does it make sense? In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 1998), IEEE Computer Society, p. 182.

[2] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting software architecture to implementation. In *International Conference on Software Engineering, ICSE 2002* (Orlando, Florida, USA, May 2002).

[3] Alia, M., Lenglet, R., Coupaye, T., and Lefebvre, A. Querying on reflexive component-based architectures. In *Proceedings of the 30th EUROMICRO Conference* (Rennes, France, Sept. 2004).

[4] Allen, R. J. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. Technical Report Number: CMU-CS-97-144.

[5] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 01*, 1 (2004), 11–33.

[5] The FScript interpreter is available for downloading at http://fractal.objectweb.org/fscript/

[6] Batista, T., Joolia, A., and Coulson, G. Managing dynamic reconfiguration in component-based systems. In *2nd European Workshop on Software Architectures (EWSA 2005)* (2005).

[7] Bruneton, r., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36*, 11-12 (2006), 1257–1284.

[8] Bruneton, r., Coupaye, T., and Stéfani, J.-B. The Fractal component model. Tech. rep., The ObjectWeb Consortium, Sept. 2003. version 2.0.

[9] Cattell, R., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F., Eds. *The Object Data Standard – ODMG 3.0*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Apr. 2000.

[10] Ceri, S., Gottlob, G., and Tanca, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering 1*, 1 (Mar. 1989), 146–166.

[11] Coulson, G., Blair, G. S., Grace, P., Joolia, A., Lee, K., and Ueyama, J. A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)* (Cambridge, MA, USA, Nov. 2004).

[12] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)* (Amsterdam, Netherlands, 2001).

[13] David, P.-C. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Université de Nantes / École des Mines de Nantes, July 2005.

[14] David, P.-C., and Ledoux, T. An aspect-oriented approach for developing self-adaptive Fractal components. In *5th International Symposium on Software Composition (SC'06)* (Vienna, Austria, Mar. 2006).

[15] David, P.-C., Léger, M., Grall, H., Ledoux, T., and Coupaye, T. A multi-stage approach for reliable dynamic reconfigurations of component-based systems. In *Proceedings of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)* (Oslo, Norway, June 2008), LNCS, Springer Verlag.

[16] Dowling, J., and Cahill, V. The K-Component architecture meta-model for self-adaptive software. In *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan* (Sept. 2001), A. Yonezawa and S. Matsuoka, Eds., vol. 2192 of *Lecture Notes in Computer Science*, AITO, Springer-Verlag, pp. 81–88.

[17] Fassino, J.-P., Stefani, J.-B., Lawall, J., and Muller, G. THINK: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference* (2002).

[18] Garlan, D., Monroe, R. T., and Wile, D. Acme: architectural description of component-based systems. In *Foundations of component-based systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, New York, NY, USA, 2000, pp. 47–67.

[19] Hajiyev, E. CodeQuest - source code querying with datalog. Msc thesis, St. Anne's College, University of Oxford, Oxford University, Sept. 2005.

[20] Jayaprakash, N., Coupaye, T., Collet, C., and Rivierre, N. Des règles actives au sein d'une infrastructure logicielle autonomique. In *Journées Composants 2005, 4ème Conférence Francophone autour des Composants Logiciels* (Le Croisic, France, Apr. 2005).

[21] Kephart, J., and Chess, D. M. The vision of autonomic computing. *IEEE Computer 36*, 1 (Jan. 2003), 41–50.

[22] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (1990), 1293–1306.

[23] Léger, M., Coupaye, T., and Ledoux, T. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In *LMO* (2006), R. Rousseau, C. Urtado, and S. Vauttier, Eds., Hermès Lavoisier, pp. 21–36.

[24] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the ACM SIGSOFT'96 Fourth Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA, Oct. 1996), ACM SIGSOFT, pp. 24–32.

[25] Medvidovic, N., and Taylor, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26*, 1 (Jan. 2000).

[26] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys 37*, 4 (Dec. 2005), 316–344.

[27] Moreira, R. S., Blair, G. S., and Carrapatoso, E. Supporting adaptable distributed systems with FORMAware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 320–325.

[28] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[29] Oreizy, P. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science University of California, Irvine, Aug. 1996.

[30] Pessemier, N., Seinturier, L., Coupaye, T., and Duchien, L. A model for developing component-based and aspect-oriented systems. In *Software Composition, 5th International Symposium, SC 2006* (Vienna, Austria, Mar. 2006), W. Löwe and M. Südholt, Eds., vol. 4089 of *Lecture Notes in Computer Science*, Springer, pp. 259–274. Revised Papers.

[31] Polakovic, J., Mazaré, S., Stefani, J.-B., and David, P.-C. Experience with implementing safe reconfigurations in component-based embedded systems. In *The 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)* (Boston, MA, USA, July 2007), Lecture Notes in Computer Science, ACM, Springer Verlag.

[32] Redmond, B., and Cahill, V. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of ECOOP 2002* (Malaga, Spain, May 2002), vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 205–230.

[33] Roshandel, R., Hoek, A. V. D., Mikic-Rakic, M., and Medvidovic, N. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering Methodologies 13*, 2 (2004), 240–276.

[34] Rouvoy, R. *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions.* PhD thesis, Université des Sciences et Technologies de Lille, Lille, France, Dec. 2006.

[35] Szyperski, C. *Component Software.* ACM Press, New York, 1997.

[36] Traiger, I. L., Gray, J., Galtieri, C. A., and Lindsay, B. G. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst. 7*, 3 (1982), 323–342.

[37] Weikum, G., and Schek, H.-J. Concepts and applications of multilevel transactions and open nested transactions. 515–553.

[38] World Wide Web Consortium. XML path language (XPath) version 1.0. W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xpath/`.

[39] World Wide Web Consortium. XML path language (XPath) version 2.0. W3C Recommendation, Jan. 2007. `http://www.w3.org/TR/xpath20/`.

A.18 A Multi-staged Approach to Enable Reliable Dynamic Reconfiguration of Component-Based Systems

A Multi-stage Approach for Reliable Dynamic Reconfigurations of Component-Based Systems*

Pierre-Charles David[1], Marc Léger[2], Hervé Grall[1],
Thomas Ledoux[1], and Thierry Coupaye[2]

[1] OBASCO Group, EMN / INRIA, Lina
École des Mines de Nantes
4 rue Alfred Kastler
F-44307 Nantes CEDEX 3
[2] France Télécom, Recherche & Développement
28, chemin du vieux chêne
F-38243 Meylan

Abstract. In this paper we present an end-to-end solution to define and execute reliable dynamic reconfigurations of open component-based systems while guaranteeing their continuity of service. It uses a multi-stage approach in order to deal with the different kinds of possible errors in the most appropriate way; in particular, the goal is to detect errors as early as possible to minimize their impact on the target system. Reconfigurations are expressed in a restricted, domain-specific language in order to allow different levels of static and dynamic validation, thus detecting errors before executing the reconfiguration where possible. For errors that can not be detected early (including software and hardware faults), a runtime environment provides transactional semantics to the reconfigurations.

1 Introduction

Complex software systems must be modified/maintained during their lifetime, for example to fix bugs or include new functionalities. It is often not practical – or even possible – to stop the system in order to perform these changes. Instead, the changes must be applied dynamically to keep the running system available.

There are two conflicting forces that make evolution especially challenging. On the one hand, the evolutions that will be applied to a system cannot be precisely anticipated at the time it is initially built and deployed. This means the system must be kept open and flexible to accommodate future needs. On the other hand, modifying production systems that are often business-critical is very risky, and we need to ensure that these changes will cause the minimum possible disruption, even though we do not know ahead of time the actual changes that will be made. In short, we need a way to provide *reliable dynamic reconfigurations*.

* This work is partially funded by the Selfware RNTL project (`http://sardes.inrialpes.fr/selfware`) and the Selfman IST project (`http://www.ist-selfman.org/`).

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 106–111, 2008.

By *reliable* we mean: *(i)* reducing as much as possible the occurrence of errors (fault prevention), *(ii)* when errors that could not be prevented actually happen, minimize the damage they cause to the system (fault tolerance).

This paper introduces a modular validation chain to support *reliable* dynamic reconfigurations on top of general-purpose component models like Fractal [1]. The chain is based on a decomposition of the life-cycle of individual reconfigurations in multiple stages, from their definition to their actual execution on the target system. As reconfiguration scripts go through these successive stages, different techniques are used to "weed out" incorrect reconfigurations and handle errors that could not be prevented. The different stages of the validation chain complement each other to offer strong reliability guarantees. At the same time, the chain stays modular and can be customized to support different tradeoffs between performance and guarantees depending on the domain.

In the rest of this paper, we first present the overall architecture of our approach (Sect. 2), detailing the different kinds of errors our validation chain handles thanks to its multi-stage architecture. Sections 3 to 5 then give more details on each successive stage of the chain. We conclude (Sect. 6) with an overview of the current status of these different modules and of the future work.

2 Overview of the Validation Chain

The goal of the proposed validation chain is to ensure the reliability of the dynamic reconfigurations of software architectures (e.g. component replacement, reconnection of component bindings). It relies on the use of a dynamic component model that supports unanticipated reconfigurations, typically thanks to reflective features. In our case we use Fractal [1] for its flexibility and extensibility.

The main idea behind our proposal is to handle the different kinds of errors at different points in the life-cycle of a reconfiguration. Accordingly, the validation chain is organized with three main stages as illustrated in figure 1, each stage corresponding to a different step in the life-cycle of the scripts that describe the reconfigurations to be executed. At each stage, if an error is detected, the reconfiguration is immediately rejected. Hence the whole chain acts as a sequence of increasingly specific sieves that scripts must pass through, from basic sanity checks to a full-blown managed execution of the reconfigurations as transactions.

Loading. First, the specification of a reconfiguration is loaded into the validation chain, in the form of a *reconfiguration script*. Such a script can be executed many times in its lifetime, with different target architectures, but it is loaded only once. For example, a generic component replacement script can be reused with different parameters each time a component must be updated to a new version. At this time, the possible target architecture are only specified by the architecture model, which defines some rules that the architectures under consideration satisfy. With these informations, the possible validations include various levels of *static analyses* filtering out reconfiguration scripts that could cause errors when applied to a concrete architecture.

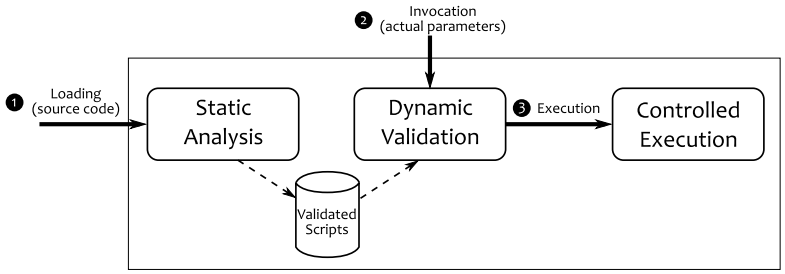


Fig. 1. The validation chain's architecture

Invocation. After loading, a user may want to actually execute the reconfiguration on a particular target architecture. Some additional validations can now be performed: this stage filters out scripts that are incompatible with the given target architecture.

Execution. Finally, the reconfiguration is executed on the target architecture. If the previous steps have been precise enough, most erroneous reconfigurations have already been rejected at this point. However, some kinds of errors are either impossible to predict (e.g. hardware faults) or too costly to detect. To handle these errors, the execution stage uses a runtime environment providing transactional properties to reconfigurations in the Fractal model. Although it can actually handle all the errors detected by earlier stages, this choice may make the architecture not available during a too long time.

The different stages of the validation chain work together providing an integrated whole. At the same time, the chain stays modular, and some of the stages can be disabled or replaced. As the different analysis techniques have different costs, the validation chain can be customized depending on the target architectures: critical systems will require more complex static analyses in the earlier stages, and may even include a test run on a replica system whereas the cost of these steps may be redhibitory in other contexts. The rest of the paper gives more details on each of the successive chain stages.

3 Static Analysis with Respect to the Architecture Model

The first stage in the validation chain loads the source code of the reconfiguration script into the chain. Its goal is to verify the validity of the reconfiguration with respect to the underlying architecture model. The component model defines some rules to be satisfied by the architectures under consideration. At this point, the actual architectures to which the script will be applied are unknown.

The reconfiguration scripts are written in a domain-specific language named FScript [2] that we have defined for this purpose. This language not only allows reconfigurations to be easily expressed, but also ensures some safety properties: for instance, any well-formed script terminates.

When a script is well-formed, a semantic analysis introduces an axiomatic definition of the script execution. This analysis is parameterized by a selection of the rules of the architecture model: this allows us to easily support variants, at the infrastructure or application levels, like different architectural styles. Note that some rules can be discarded because they are too costly to analyze. The analysis defines Hoare's correctness formulas $\{P\}S\{Q\}$ where S is the script and P and Q are properties describing the architecture to be reconfigured (expressed in first-order logic). Such a formula means that any architecture satisfying the precondition P will satisfy the postcondition Q after the completion of the script S. The aim of the semantic analysis is to determine a precondition P that does not lead to an error state where the architecture violates some invariant rules under consideration: only the architectures that satisfy the precondition P will be reconfigured by the script. Therefore, if the precondition P is *false*, it means that the script is not useful according to the analysis and should be rejected. Otherwise, the script is considered as potentially valid, and passed to the next stage of the validation chain along with the computed precondition. Note that the semantic analysis can be more or less precise: the precondition P is sufficient to ensure the absence of errors with respect to the selected rules, but not necessary.

4 Validation with Respect to the Target Architecture

The second stage in the validation chain is triggered each time the user requests the invocation of a reconfiguration script by giving a target architecture and actual parameters. This stage performs additional validations thanks to this information, but without actually modifying the target architecture.

At this point, the script has already passed the first stage of the chain, and has a pre-condition associated to it. The first step is thus a simple *compatibility check*, which consists in evaluating the pre-condition on the target architecture and the actual parameters. This can be done easily, and only requires to introspect the target architecture, without modifications.

If the compatibility check has succeeded, an optional second step can be included, which consists in a *simulation* of the script's execution. This step uses a virtual implementation of the target architecture, on which the reconfiguration script is executed using the script interpreter. The virtual architecture is initialized with the initial state of the target system, but implements "copy-on-write" semantics: operations are applied to the virtual copy, and do not modify the actual target system. If any of the component model invariant rules of the architectural model are violated during the simulation, the invocation is rejected.

One advantage of the simulation is that it can be more precise (and thus can catch more errors) that the static analyses, which may be restricted by a selection of the invariant rules to be preserved. Also, by instrumenting the virtual architecture to be reconfigured, it can generate the exact trace of the reconfiguration performed by the script, which can be "replayed" with very little overhead to reproduce its effect on the actual target system [3]. The only drawback is that this step can increase the latency of the reconfiguration.

5 Execution of Reconfigurations as Transactions

The final stage of the chain is the actual application of the reconfiguration script on the target architecture. Depending on how the previous stage was configured, it uses either a compiled form of the script, or the specialized trace of reconfiguration operations that was generated during the simulation.

Because the overall objective of the validation chain is to guarantee the reliability of the reconfiguration, this step must either apply the complete reconfiguration script without errors, or, in case of errors, restore the system to the last consistent state before the execution of the script by rollbacking the failed reconfiguration: it must be fault tolerant. The failures that happen during the actual execution of the reconfiguration include software failures (e.g. violation of the architecture model) that were not detected earlier, and some errors that are fundamentally impossible to predict (e.g. hardware crashes). In all cases, the resulting architecture must be in a consistent state according to the definition of the underlying architecture model.

These objectives call for the use of transaction management techniques, as they closely match the standard ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions in distributed computing [4]. In order to execute reconfiguration scripts inside global transactions with automatic demarcation, we use an extended version of the Fractal component model [5], which provides transactional semantics for Fractal architectures. Therefore, reconfigurations can benefit from ACID properties to support concurrency, recovery, and to guarantee system consistency.

6 Conclusion and Future Work

The objective of this work is to make runtime reconfigurations of open software architectures reliable while maximizing their availability. We specially target reflexive component-based architectures for their suitable adaptability property, as exemplified by the use of the Fractal model in our current implementation. Our solution relies on a multi-stage validation chain with two main dependability methods: fault prevention and fault tolerance. Fault prevention notably includes the use of static analysis on a dedicated reconfiguration language in order to detect invalid reconfigurations with respect to the architecture model, and an additional simulation stage on the target architecture. Fault-tolerance is ensured by a transactional runtime for the actual execution of reconfigurations.

Although several component models support open dynamic reconfigurations, they do no take into account the reliability of reconfigurations. On the contrary, most work on reliability and validation for component-based architectures deal with Architecture Description Languages [6,7] only include static validations and do not support unanticipated reconfigurations. Recent component models, like FORMAware [8] and Plastik [9], rely on reflexive architectures to allow unanticipated reconfigurations while supporting some kinds of guarantees checked at runtime. Our work differs in that we provide a multi-stage architecture that

integrates different complementary validation techniques in a consistent whole. Depending on the domain requirements, the focus of the validation chain can be put on the static validation, the controlled execution, or both, for instance for critical systems.

Currently the overall architecture of the validation chain is in place, and the whole system is usable although some of the individual stages are not yet complete: the simulation and execution of reconfiguration programs is fully functional, including transactional guarantees. Our current focus is on the earlier steps, and in particular the definition and implementation of the static analysis of reconfiguration scripts, which requires a formal definition of both the FScript language and the Fractal model. Once we have a fully implemented validation chain for Fractal, we plan to extend it to support other component models.

References

1. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and its Support in Java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11-12), 1257–1284 (2006)
2. David, P.C., Ledoux, T.: Safe dynamic reconfigurations of Fractal architectures with FScript. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, Springer, Heidelberg (2006)
3. Polakovic, J., Mazaré, S., Stefani, J.B., David, P.C.: Experience with implementing safe reconfigurations in component-based embedded systems. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, Springer, Heidelberg (2007)
4. Traiger, I.L., Gray, J., Galtieri, C.A., Lindsay, B.G.: Transactions and consistency in distributed database systems. ACM Trans. Database Syst. 7(3), 323–342 (1982)
5. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in the Fractal component model. In: Proceedings of the 6th workshop on Adaptive and reflective middleware (ARM 2007), p. 6. ACM, New York (2007)
6. Allen, R.J.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon University Technical Report Number: CMU-CS-97-144 (May 1997)
7. Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: Using object-oriented typing to support architectural design in the C2 style. In: Proceedings of the ACM SIGSOFT 1996 Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, ACM SIGSOFT, October 1996, pp. 24–32 (1996)
8. Moreira, R.S., Blair, G.S., Carrapatoso, E.: Supporting adaptable distributed systems with FORMAware. In: ICDCSW 2004: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops, Washington, DC, USA, pp. 320–325. IEEE Computer Society Press, Los Alamitos (2004)
9. Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, Springer, Heidelberg (2005)

A.19 Security Issues in SmallWorld Network Routing

The paper has been accepted by the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008).

Security Issues in Small World Network Routing

Felix Halim, Yongzheng Wu, Roland Yap
School of Computing National University of Singapore
Law Link, Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

Abstract

Small World Network (SWN) have been shown to be navigable in the sense that a short route can be found using efficiently using decentralized algorithms. This routing relies on nodes having a position to guide the routing such as its coordinates. Even in the absence of positional information such as node coordinates, by using local self-reorganization, it is possible to reconstruct a proxy for the node coordinates which still allows for efficient routing. This paper shows that in the presence of malicious nodes, the self-reorganization mechanism breaks down. We investigate what self protection mechanisms for such SWNs. Preliminary results using a simple restart mechanism for self tuning can mitigate much of the effect of malicious nodes.

1. Introduction

Structured Overlay Networks such as Distributed Hash Tables (DHT) provide a self organization network layer which can be used for communication and storage. There has been extensive work on DHTs such as Chord, Pastry, CAN, Kademlia, DKS, etc [1]. However, DHTs have to deal with the maintenance costs of dealing with dynaminism (churn) and be attacked in a variety of ways. Perhaps the worst problem is that in P2P settings, it is difficult to defend against Sybil attacks [3].

Small World Networks (SWN) presents an interesting alternative to self organizing networks. SWN are networks characterized by "small world phenomena". Perhaps, the best example is the idea of "six degrees of separation", that any two persons can be linked through a chain of acquaintances whose path length is at most six. Two important properties of SWN are navigatibility, being able to route or find information, and low diameter (which accounts for the six in six degrees). The success of social networking websites such as Facebook, LinkedIn, MySpace, etc. are possibly driven also by such small world properties.

There are many graphs proposed to model SWN, the Watts-Strogatz [5] model gives some simple insights. They model a SWN by starting with a network with regular connections (each node is connected to its k-nearest neighbors) and then some of the edges of a node are rewired with probability p to a random node. This results in graphs which are partly between regular graphs ($p = 0$) and random graphs ($p = 1$). The "rewired" edges act like shortcuts in the network to make the diameter small.

One question is given a SWN, how to efficiently navigate the graph, i.e. how to efficient route a message between any two nodes in the graph. Kleinberg [2] proposes a SWN model which is inspired by the Watts-Strogatz model. Starting with a graph which is a lattice, e.g. a 2-D grid, add for each node, a constant number of shortcut edges to other edges with probability proportional to $d(u, v)^{-r}$ where $d()$ is the distance function giving the Manhattan distance between nodes in the lattice. Note that the distribution of shortcut nodes follows a power law distribution. Kleinberg shows that greedy routing, thus using only local information, is efficient with an expected route length of $O(log^2 n)$ hops where n is the number of nodes in the graph.

In a P2P setting, efficient decentralized routing algorithms using only local information is attractive. However, the assumption in the Kleinberg model that nodes have knowledge of their own location (e.g. lattice coordinates) is too strong. It is not reasonable that nodes or peers know position information which can be thought of as global rather than local information. This would suggest routing difficulties even with a SWN and a low graph diameter.

Sandberg [4] proposed a way around this difficulty. Nodes have a position but this may be incorrect, so a continuous self-tuning approach is used to correct the position. Each node performs a fixed length random walk. The nodes at both ends of the random walk can decide to switch their positions to maximize the maximum likelihood that the node positions follow the edge distribution in the Kleinberg model. This can be achieved by minimizing the product of edge distances. The self tuning employed is somewhat analogous to self stabilization in DHTs but it can be thought of as shuffling nodes around the graph.

We show that routing and self-tuning break down when there are malicious nodes which can infect nodes with invalid positions. We discuss attacks by malicious nodes and investigate self-protection strategies. Preliminary results show that decentralized local reset of node position has potential in reducing the effect of malicious nodes.

2. Malicious Nodes and Self-Protection

The self-tuning in Sandberg assumes all nodes are good. If there are malicious nodes in the network, they can attack self-tuning by lying about their neighbors positions. This results in a malicious node being able to disseminate false position information which in turn affects the self-tuning and routing algorithms. Malicious nodes can also collude which makes it harder to detect such cheating.

After switching position with a good node, the malicious node has the good node's position and can reset it to a particular position X. After a while, there will be many good nodes having position X. However, duplicate positions can be detected once the nodes contact each other. One way to recover from duplicate position is to reset to a random position once duplicate is detected.

Another attack is to reset to a position close to X, so that, after a while, many good nodes are concentrated near X. This has two effects: firstly, nodes which are not near X will have high load; secondly, routing success rate will drop. Fig. 1 shows the success rate drops below 20%.

A decentralized self-protection strategy which doesn't need global information is for nodes to reset/restart their position with a certain probability. For simplicity, we can consider that the node restart decision occurs in rounds, a round is the time unit when all nodes decided whether or not they switch with another node. The restart probability can be calculated from different measures observed locally on each round (such as the density of the position distribution, the successful routes percentage, the switch percentage).

A malicious node succeeds when it switches with a good node. Thus it *poisons* a good node, *infect* good node with a false position. Fig. 1 shows that without self-protection (*Without Restart*), the entire network can be poisoned.

3 Experimental Results

We give some initial experiments to demonstrate the effect of malicious nodes and the effectiveness of the restart strategy. Our initial experiments focus on malicious nodes which attack by actively perform random walks to good nodes to infect them. Fig. 1 shows experimental results with 10^5 nodes in the SWN, the random walk length is 8 (same as [4]). The graphs show the percentage of successful route, switch, and infected nodes over time. Without self-protection, all nodes will eventually become infected (*Without Restart % Infected*) and number of successful routes drop significantly (*% Success Route*). With the restart strategy, successful routing can be maintained around 80% (without malicious nodes, convergence is rapid and routing success is $> 90\%$ after ~ 100 rounds), the infection rate is contained to about 10%.

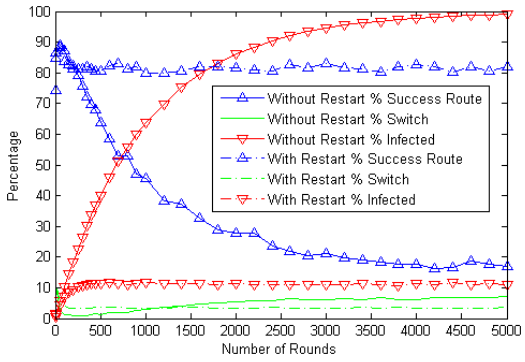


Figure 1. SWN with 0.1% malicious nodes: no self-protection (solid lines); self-healing protection strategy with reset probability 0.008 (dotted lines).

4. Discussion

We show that the self-tuning which allows a small world network to function with decentralized routing fails when there are malicious nodes. Thus, self-protection mechanisms are necessary. Our preliminary experiments using self-protection using a self-healing strategy with partial restart shows that simple decentralized security mechanisms have promise. We observe that: i) without protection, all nodes will be eventually infected by even a single malicious node; ii) a small number of malicious nodes require a small restart probability, while more malicious nodes require larger restart probability. For more information and experimental results, see: `http://www.comp.nus.edu.sg/~halim/drswn`

References

[1] S. El-Ansary and S. Haridi. An overview of structured overlay networks. In *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*. 2005.

[2] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. *ACM STOC*, 2000.

[3] B. Levine, C. Shields, and N. Margolin. A survey of solutions to the sybil attack. Technical Report 2006-052, U. of Massachusetts Amherst, 2006.

[4] O. Sandberg. Distributed routing in small-world networks. *Workshop on Algorithm Engineering and Experiments*, 2006.

[5] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, pages 440–442, 1998.

A.20 A Transactional Scalable Distributed Data Store: Wikipedia on a DHT

A Transactional Scalable Distributed Data Store: Wikipedia on a DHT*

T. Schütt M. Moser S. Plantikow F. Schintke A. Reinefeld

Zuse Institute Berlin (ZIB)

E-mail: {schuett, moser, plantikow, schintke, reinefeld}@zib.de

Abstract

We present a scalable Wikipedia on top of a transactional distributed data store. It delivers 1500 pages per second running on ten cluster nodes, where the Wikipedia render farm delivers in peak 2000 pages per second using about 200 servers. Our distributed key-value store is based on a DHT which guarantees atomicity, consistency, isolation and durability of distributed updates operations.

1. The Challenge: Fast Transactions on a Global DHT

To extend P2P applications beyond the typical file sharing, support for *transactions* on DHTs is the most important and yet missing feature. It will allow a wide variety of new applications to benefit from the inherent scalability and fault tolerance of P2P systems. It could be applied to globally distributed services, but also to locally deployed, business critical hosting solutions running on commodity hardware. In many applications, e.g. online-stores [1], scalability and a quick response are of key importance for overall customer satisfaction.

Traditional P2P file sharing supports only write-once, read-often access to data. Modifications and write access are not supported. Files could be replaced by newer versions, but no guarantees on consistency and atomicity are given.

Recent advances in query expressiveness [5] and improved consistency guarantees [4] for DHTs allow to build scalable, geographically distributed read-write stores.

As an application, we present a highly scalable P2P database backend for Wikis. Wikis became increasingly popular in the recent years. As many users concurrently access pages a proper concurrency control scheme is needed for conflict detection. We think that Wikis are of particular interest for this scalability challenge as they have to cope with a high load, which is exponentially increasing.

Current Wiki implementations use a centralized hosting approach. However, a P2P approach fits the general idea of sharing in two aspects: sharing the data and additionally sharing the resources.

For our P2P-Wiki, we extended Chord$^{\#}$ [5] by a data access layer that executes all data operations inside transactions. Thereby our transactional scalable distributed data store provides:

Scalability: DHTs are inherently scalable and can run on geographically distributed nodes.

Replication: Symmetric replication [2] stores each item on several nodes to ensure data availability when nodes crash.

Data Consistency: Our transaction layer provides consistent access to the stored data despite node failures.

Load-Balancing: If the query load is unevenly distributed over the nodes, lightly loaded nodes take over a part of the data from overloaded nodes [3].

Self-*: DHTs are designed for harsh environments where node failures are common. Their maintenance protocols include self-repairing resp. self-healing functions. Self-optimization is partly covered by the load balancing.

2. Wikipedia

Wikipedia[1] is a "free encyclopedia, that anyone can edit". All content is created by users, the infrastructure is funded by donations, and according to Alexa (www.alexa.com), a web information company, it belongs to the ten most frequently visited web-sites.

Beyond providing dumps of the database (download.wikimedia.org), Wikipedia is also open about their hosting architecture. It consists of three tiers: (a) web proxies, (b) HTML renderers, and (c) databases, see Fig. 1.

*Part of this work was carried out under the SELFMAN and XtreemOS projects funded by the European Commission.

[1] www.wikipedia.org

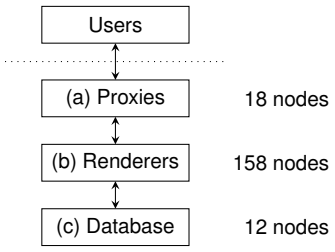


Figure 1. Wikipedia architecture of main datacenter in Tampa, Fl (Jan. 2008).

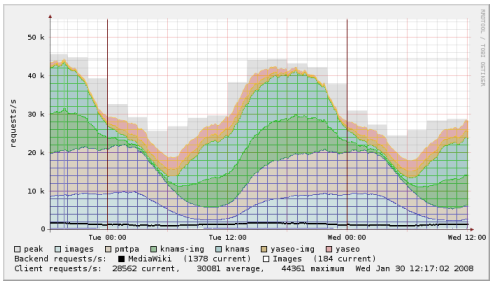


Figure 2. wikipedia.org access: 45k reads / s (43.6 k/s cached, ca. 2 k/s backend)

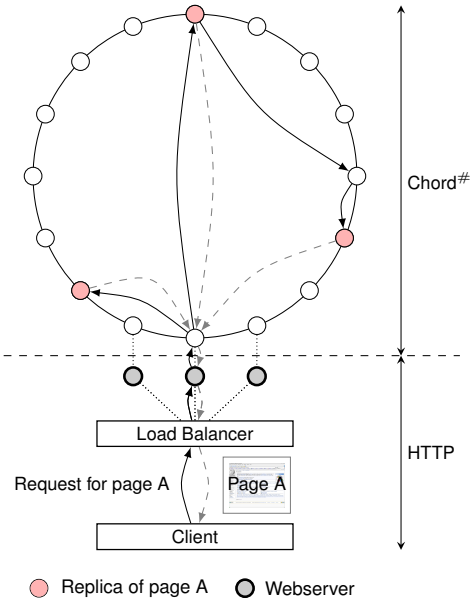


Figure 3. Architecture of a distributed Wikipedia that uses a data store based on Chord# with transactions

It currently runs on about 240 servers in 3 data centers (Tampa (USA), Amsterdam (NL), Seoul (Kr)) which serve in peak 45k requests per second (see Fig. 2)[2]. 95 % of all requests are served by a few squid proxies. Only about 2000 requests/s hit the backends.

3. Architecture and Algorithms

Data Store = Chord# + Transactions

To build our data store, we used the concepts of Chord# [5] *and* transactional DHTs [4], Fig. 3. In the following paragraphs, we discuss how we implemented Wikipedia features on the data store.

3.1. Editing Pages

The most simple data model for storing Wiki pages maps directly onto DHTs. Unique page titles are used as keys and the pages' current content is used as value. To ensure that users do not accidentally overwrite each other's changes, write operations may only be performed on a page if no concurrent change has occurred since the page was originally read for editing. Most Wikis achieve this by assigning version numbers to pages and comparing them atomically during writing of new versions. If a conflict is detected, the write operation is aborted and the user is presented with detailed information about the conflicting versions.

This operation requires the use of transactions to guarantee the atomic execution of compare-version-and-write-or-abort in our DHT scenario. Proper version numbers are generated as part of the transaction protocol (increment on commit).

3.2. Additional Wiki Functionality

Modern Wikis, like Mediawiki, the software behind Wikipedia, provide a huge set of sophisticated features that allow users to search, structure, organize, and create pages. This includes keeping track of page history and global changes, page search functionality, categories, and page backlinks.

To implement these *navigational aids* efficiently, additional per-page data and indices have to be stored in the DHT. Page content and all affected data structures can only be updated consistently using transactions.

Page History. Reviewing changes, that have been made to a page, allows to understand how its content evolved over

[2] source http://hemlock.knams.wikimedia.org/~leon/stats/reqstats/

time. Additionally, storing previous versions allows to revert vandalism and user errors.

For our implementation, we store each page with its whole history as a single item in the DHT. While this allows for quick updates, it does not allow to implement efficient queries for recently changed pages.

Categories. Categories allow grouping of pages and thus provide a bottom-up approach to directory building (folksonomy). Therefore, each page lists the categories it belongs to. Users want to list all pages of a category, which requires the construction of a reverse index.

For each category we create an item with the *category-key* and a value that stores all keys of pages that are part of a category. Whenever the set of categories a page belongs to changes, the corresponding category item is updated consistently using a transaction.

Back links. The back links of a page are all pages that refer to it. Showing them can help discovering the context and related content of a given page. We have not yet implemented this.

Full-Text Index. Many Wikis implement full text search. However, it is currently unclear how such a functionality could be implemented on top of a DHT without introducing a large overhead. At this time, we therefore decided to leave the indexing to external search engines like Google and their dedicated infrastructure.

4. Transactions

Our data store provides transactions and therefore strong data consistency [4]. To tolerate node failures, data items are replicated according to symmetric replication [2]. Operations on items are implemented with a quorum voting scheme and thus tolerate unavailability of a number of replicas.

Because the system's focus is on *strong* data consistency, we have to sacrifice availability during network-partitioning. It has been shown that it is impossible to guarantee consistency *and* availability while tolerating network partitions [7].

Each item is assigned a version number in order to determine the latest value. Read operations select the item with the highest version number from a majority of replicas, they read from. A single read operation on an item can be performed by parallel lookups on the replicas. Thus a read operation is executed within a single communication phase.

Write operations and transactions are coordinated by the use of Paxos atomic commit [8]. Paxos atomic commit is a non-blocking protocol. It tolerates the failure of a minority of so-called acceptors which act as the transaction manager. Write operations and transactions need three phases, including a phase to determine the nodes that take part in the atomic commit protocol.

5. Implementation

Our distributed data store consists of an implementation of Chord$^{\#}$ [5], as the underlying routing structure, and transactional storage mechanisms on top of it. The implementation is written mainly in Erlang. We use our implementation for various experiments on PlanetLab and for the overlay substrate in the EU projects SELFMAN and XtreemOS.

We replaced the simple put/get interface of DHTs with an enhanced API to access items stored on the ring. Applications are able to perform reads, writes and transactions on the items. The interface hides details like replication and node failures from the application. A replication mechanism, added to Chord$^{\#}$, creates a fixed number of replicas for each item and stores them under their particular replica key.

The Wikipedia backend implementation uses the enhanced API of Chord$^{\#}$to store pages and categories as described in Section 3 and is strictly separated from the presentation layer code.

The web presentation layer is implemented as a Java servlet and runs in a Java servlet container deployed on some of the nodes (see Fig. 3). Our web frontend utilizes plog4u[3] to convert Wiki-text to HTML and communicates with the Chord$^{\#}$ node running on the same host using *jinterface* (part of erlang).

The DHT-based Wikipedia was populated using dumps stored in a Wikipedia-specific XML vocabulary. To read the dumps from Erlang, we use XSLT to transform the XML into proper Erlang terms suitable for the Wiki. This approach turned out to be a simple method for reading large amounts of complex XML data into Erlang.

6. Performance

PlanetLab Setup. Fig. 4 shows the nodes used for our initial measurements – ca. 30 PlanetLab nodes distributed over four continents. As a test dataset we used the complete Bavarian Wikipedia with about 5000 articles. Depending on the location of replicas the median access time for 10 requests varies between 926 and 1479 ms. For the demonstration we plan to show a larger deployment on hundreds of PlanetLab nodes.

[3] plog4u.org/index.php

Figure 4. Hosts used for PlanetLab tests

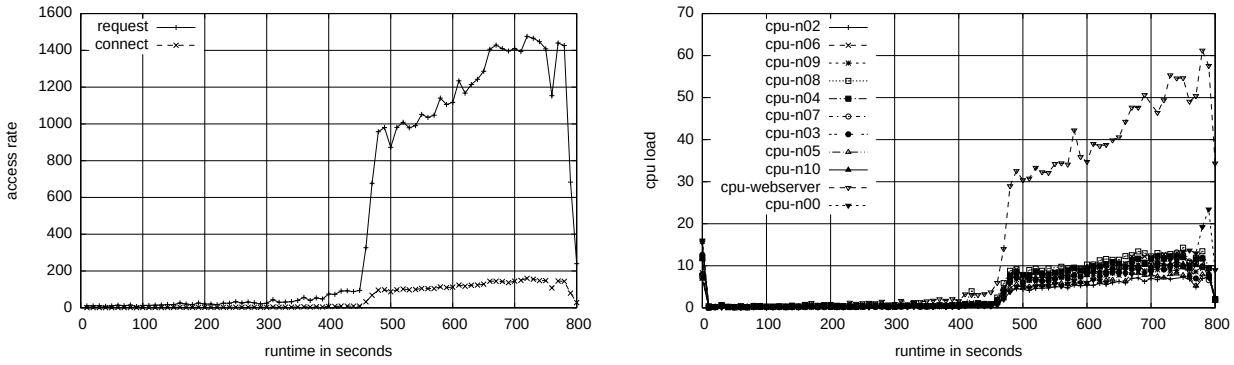


Figure 5. Throughput and CPU-load of cluster setup on increasing access rate over time

Cluster Setup. For additional throughput measurements, we used a Chord# ring on 10 nodes in a local Linux cluster with the same Bavarian Wikipedia dataset as above. We measured the number of requests/s that can be served by increasing the request rate until the system cannot handle additional requests any more. The results are presented in Fig. 5. The left graph shows the throughput over the duration of the experiment – the peak is at about 1500 pages/s. The right hand side shows the CPU load on the involved nodes. It can be seen that the load on the Chord# nodes is moderate while the (single) webserver was a bottleneck. For the demonstration we plan to show a larger cluster installation with the load distributed over several webserver nodes. We expect a higher throughput then.

Cluster Setup with new Frontend. We reimplemented the Java frontend using the Jetty webserver and ran additional throughput tests on the aforementioned cluster but

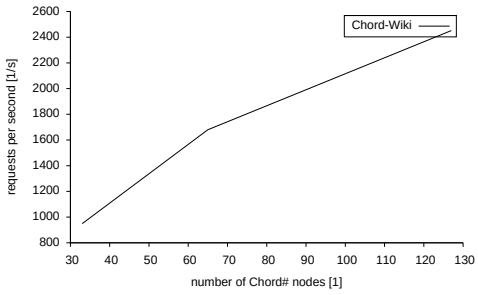


Figure 6. Jetty Tests

with several webservers instead of just one. Fig. 6 shows the scaling of the throughput with increasing numbers of Chord# nodes. As Chord# nodes are essentially single-threaded, adding more nodes is the only way to benefit from additional CPUs. For the tests about 2 nodes shared one

CPU.

The major problem with the scalability tests were again the webservers and the load generator. When we tried to go beyond 2500 requests/s the webservers became overloaded and started to drop requests, while the Chord$^{\#}$ was mostly idle. We will investigate other webservers which can better handle the load.

7. Demonstration

We will present the system running on PlanetLab and a different installation running on the cluster (for a screenshot see Fig. 7). For both we show browsing the Wiki and editing the pages. Images are not included in the available dumps and therefore not rendered in our system. The audience will be invited to access, browse and edit pages.

As dataset we will use the dump of Wikipedia in *simple English*, which currently consist of 24,680 articles.

During the demonstration we will show:

- browsing of the Wiki
- editing pages
- killing of nodes during runtime
- adding of new nodes to the system
- live statistics of data distributions and ring structure
- throughput tests

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, Oct. 2007.

[2] A. Ghodsi, L. Alima, S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *DBISP2P*, Aug. 2005.

[3] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. *IPTPS 2004*, Feb. 2004.

[4] M. Moser, S. Haridi. Atomic Commitment in Transactional DHTs. 1st CoreGRID Symposium, Aug. 2007.

[5] T. Schütt, F. Schintke, and A. Reinefeld. Structured overlay without consistent hashing: Empirical results. *GP2PC'06*, May. 2006.

[6] T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *EuroPar'07*, Aug. 2007.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *SIGACT News*, 2002.

[8] J. Gray and L. Lamport. Consensus on transaction commit. In *ACM Trans. Database Syst.*, ACM Press, 2006, 31, 133-160.

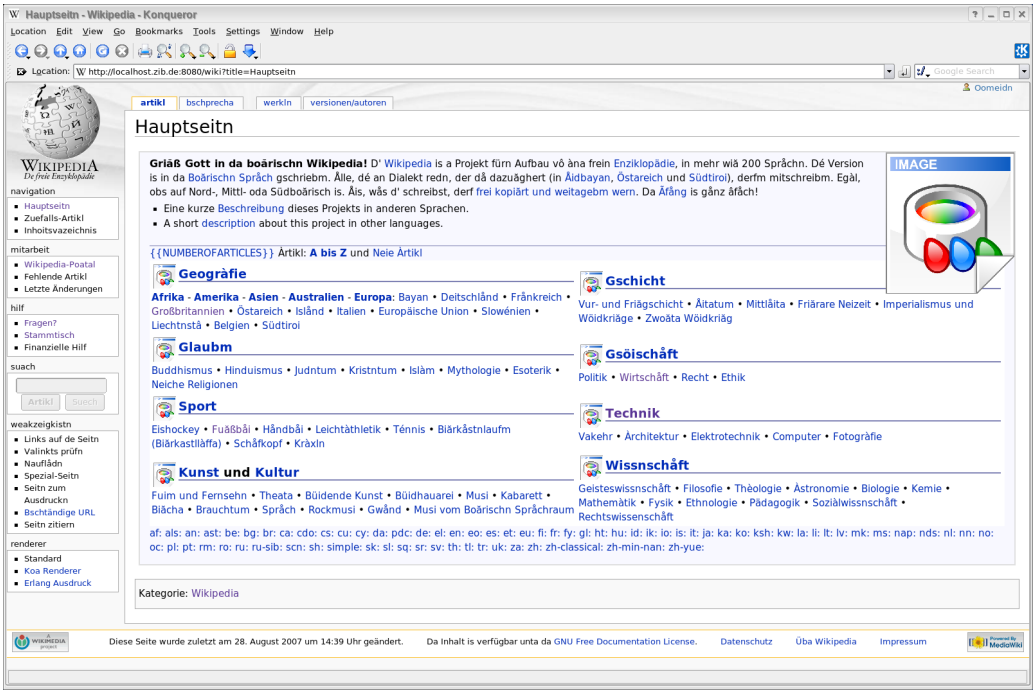


Figure 7. Screenshot of our P2P-Wikipedia serving the Bavarian Wikipedia

Bibliography

[1] Alloy Analyzer Web Site. Accessible at: http://alloy.mit.edu/.

[2] http://www.peerialism.com.

[3] http://www.slf4j.org.

[4] Julia. http://fractal.objectweb.org/julia, 1999-2008.

[5] INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA. IEEE, 2005.

[6] T. Abdellatif, J. Kornas, and J.B. Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In Component Deployment, 3rd International Working Conference, CD 2005, volume 3798 of Lecture Notes in Computer Science. Springer, 2005.

[7] A. Akkerman, A. Totok, and V. Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In Component Deployment, Third International Working Conference, CD 2005, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[8] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control. In 17th Conference on Systems Administration (LISA 2003). USENIX, 2003.

[9] Cosmin Arad and Seif Haridi. Kompics. http://kompics.sics.se, 2008.

[10] J. Armstrong. Making reliable distributed systems in the presence of software errors. PhD thesis, Swedish Institute of Computer Science (SICS), 2003.

[11] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. Software Quality Journal, 15(3), 2007.

[12] Marco Avvenuti and Alessio Vecchio. Application-level network emulation: the emusocket toolkit. J. Network and Computer Applications, 29(4):343–360, 2006.

[13] L.S. Barbosa, S. Meng, B.K. Aichernig, and N. Rodrigues. On the Semantic of Componentware: A Coalgebraic Perspective, volume 2 of Component-Based Software Development, chapter 3. World Scientific, 2006.

[14] J. Barwise and L. Moss. Vicious Circles, volume 60 of CSLI Lecture Notes. CSLI Publications – Center for the Study of Language and Information, Stanford, California, 1996.

[15] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony. Columbia Report 2004, 2004.

[16] M. C. Bastarrica, A. A. Shvartsman, and S. A. Demurjian. A Binary Integer Programming Model for Optimal Object Distribution. In 2nd Int. Conf. On Principles Of Distributed Systems. OPODIS 98. Hermes, 1999.

[17] Tom Berson. Skype security evaluation. 2005.

[18] D P Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. Computational Optimization and Applications, (1):7–66, 1992.

[19] Dimitri P. Bertsekas. Network Optimization: Continuous and Discrete Models (Optimization, Computation, and Control). Athena Scientific, 1998.

[20] Ashwin R. Bharambe, Cormac Herley, and Venkata N. Padmanabhan. Analyzing and improving a bittorrent networks performance mechanisms. In INFOCOM, 2006.

[21] Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing skype traffic: When randomness plays with you. SIGCOMM 2007, 2007.

[22] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, N. De Palma, V. Quéma, and J.B. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005). IEEE Computer Society, 2005.

[23] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2006.

[24] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004. ACM, 2004.

[25] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. Software - Practice and Experience, 36(11-12), 2006.

[26] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model. ObjectWeb Consortium, 2.0.3 edition, 2004.

[27] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper., 36(11-12):1257–1284, 2006.

[28] D. Caromel, A. di Costanzo, and C. Delbé. Peer-to-Peer and fault-tolerance: Towards deployment-based technical services. Future Generation Computer Systems, 23(7), 2007.

[29] Bruno Carton and Valentin Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, Euro-Par, volume 3149 of Lecture Notes in Computer Science, pages 1060–1067. Springer, 2004.

[30] A. Chazalet and P. Lalanda. A Meta-Model Approach for the Deployment of Services-oriented Applications. In IEEE International Conference on Services Computing (SCC 2007). IEEE Computer Society, 2007.

[31] Chyouhwa Chen and Kun-Cheng Tsai. The server reassignment problem for load balancing in structured p2p systems. IEEE Trans. Parallel Distrib. Syst., 19(2):234–246, 2008.

[32] S. Cheshire and M. Krochmal. Nat port mapping protocol (NAT-PMP). Internet-Draft http://files.dns-sd.org/draft-cheshire-nat-pmp.txt, April 2008.

[33] B. Claudel, N. De Palma, R. Lachaize, and D. Hagimont. Self-protection for Distributed Component-Based Applications. In Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, number 4280 in Lecture Notes in Computer Science. Springer, 2006.

[34] Raphaël Collet. The Limits of Network Transparency in a Distributed Programming Language. PhD thesis, Dept. of Computing Science and Engineering, Université catholique de Louvain, December 2007.

[35] G. Coulson, G. S. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. ACM Trans. Comput. Syst., 26(1), 2008.

[36] T. Coupaye and J. Estublier. Foundations of enterprise software deployment. In Conference on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, 2000.

[37] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007. ACM, 2007.

[38] Anwitaman Datta, Roman Schmidt, and Karl Aberer. Query-load balancing in structured overlays. In Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07), 2007.

[39] P.C. David. Développement de composants Fractal adaptatifs: un langage dédié à l'aspects d'adaptation. PhD thesis, Université de Nantes, France, 2005.

[40] P.C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In Software Composition, 5th International Symposium, SC 2006, number 4089 in Lecture Notes in Computer Science. Springer, 2006.

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, SOSP, pages 205–220. ACM, 2007.

[42] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. S. Gokhale. DAnCE: A QoS-Enabled Component Deployment and Configuration Engine. In Component Deployment, 3rd Int. Working Conference, CD 2005, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[43] Bruno Dillenseger. Test management and load testing with salome-tmf and clif. http://www.ow2.org/view/Events/OW2QuarterlyMeetingGrenobleFrance, May 2008. OW2 Tech Day, Grenoble (France).

[44] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. http://p2ps.info.ucl.ac.be, 2008.

[45] DistOz Group. Pepino: PEer-to-Peer network INspectOr. http://p2ps.info.ucl.ac.be/pepino, 2008.

[46] E. Dolstra, M. de Jonge, and E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In 18th Conference on Systems Administration (LISA 2004). USENIX, 2004.

[47] John R. Douceur. The sybil attack. 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), 2002.

[48] A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal. Distributed slicing in dynamic systems. In 27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007). IEEE Computer Society, 2007.

[49] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In OTM Confederated International Conferences, Grid computing, high performAnce and Distributed Applications (GADA 2006), volume 4276 of Lecture Notes in Computer Science. Springer, 2006.

[50] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In ATEC '05: Proceedings of

the annual conference on USENIX Annual Technical Conference, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[51] UPnP Forum. Internet gateway device IGD protocol. Internet-Draft http://www.upnp.org/standardizeddcps/igd.asp, 2008.

[52] The Apache Software Foundation. Apache mina. http://mina.apache.org/, 2008.

[53] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, VLDB, pages 444–455. Morgan Kaufmann, 2004.

[54] D. Garlan, S.W. Cheng, A.C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer, 37(10), 2004.

[55] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems, pages 47–68. Cambridge University Press, 2000.

[56] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In 1st Workshop on Self-Healing Systems, WOSS 2002. ACM, 2002.

[57] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[58] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris M. Ouksel, editors, DBISP2P, volume 4125 of Lecture Notes in Computer Science, pages 74–85. Springer, 2005.

[59] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51–59, 2002.

[60] gnucitizens.org. Flash upnp attack. http://www.gnucitizen.org/blog/hacking-the-interwebs/, 2008.

[61] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In INFOCOM, 2004.

[62] Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In INFOCOM [5], pages 596–606.

[63] P. Goldsack. SmarFrog: Configuration, Ignition and Management of Distributed Applications. Technical Report http://www-uk.hpl.hp.com/smartfrog, HP Research Labs, 2003.

[64] Jim Gray and Leslie Lamport. Consensus on transaction commit. ACM Trans. Database Syst., 31(1):133–160, 2006.

[65] Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification, 2006.

[66] Rachid Guerraoui and Luís Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag, 2006.

[67] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. 5th International Workshop on Peer-to-Peer Systems (IPTPS06), 2006.

[68] Felix Halim, Rajiv Ramnath, Sufatrio, Yongzheng Wu, and Roland H. C. Yap. A lightweight binary authentication system for windows. Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM 2008), 2008.

[69] R. Hall, D. Heimbigner, and A. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In 21st International Conference on Software Engineering (ICSE'99), pages 174–183, Los Angeles, CA, May 1999.

[70] Ahmed Harbaoui, Bruno Dillenseger, and Jean-Marc Vincent. Performance characterization of black boxes with self-controlled load injection for simulation-based sizing. In Confrence Franaise sur les Systmes d'Exploitation (CFSE'6), February 2008.

[71] S. Haridi and N. Franzen. Tutorial of Oz, 2004.

[72] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The Autonomic Computing Paradigm. Cluster Computing, 9(1), 2006.

[73] A. Heydarnoori, F. Mavaddat, and F. Arbab. Towards an Automated Deployment Planner for Composition of Web Services as Software Components. Electr. Notes Theor. Comput. Sci., 160, 2006.

[74] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET08), 2008.

[75] D. Jackson. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol., 11(2), 2002.

[76] D. Jackson. Software Abstractions: Logic, Language, and Analysis. MIT Press, 2006.

[77] Daniel Jackson and K. J. Sullivan. COM revisited: tool-assisted modelling of an architectural framework. In ACM SIGSOFT Symp. on Foundations of Soft. Eng. (FSE). ACM, 2000.

[78] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In 5th IEEE/IFIP Conference on Software Architecture (WICSA'05). IEEE Computer Society, 2005.

[79] Jose, Marina Petrova, and Petri Mahonen. Upnp service discovery for heterogeneous networks. In Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on, pages 1–5, 2006.

[80] JUnit.org. The junit testing framework. www.junit.org, 2008.

[81] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In Geoffrey M. Voelker and Scott Shenker, editors, IPTPS, volume 3279 of Lecture Notes in Computer Science, pages 131–140. Springer, 2004.

[82] A. Keller and R. Badonnel. Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM, number 3278 in Lecture Notes in Computer Science. Springer, 2004.

[83] A. Ketfi and N. Belkhatir. Model-driven framework for dynamic deployment and reconfiguration of component-based software systems. In

Metainformatics, International Symposium, MIS 2005, volume 214 of ACM International Conference Proceeding Series. ACM, 2005.

[84] T. Kichkaylo and V. Karamcheti. Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications. In 13th International Symposium on High-Performance Distributed Computing (HPDC 2004). IEEE Computer Society, 2004.

[85] G. N. C. Kirby, S. M. Walker, S. J. Norcross, and A. Dearle. A Methodology for Developing and Deploying Distributed Applications. In Component Deployment, Third International Working Conference, CD 2005, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[86] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. 32nd ACM Symposium on Theory of Computing, 2000.

[87] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas. Design, implementation, and performance of an automatic configuration service for distributed component systems. Software - Practice and Experience, 35(7), 2005.

[88] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, (1), 1985.

[89] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, New York, February 2005.

[90] L. Lan, G. Huang, L. Ma, M. Wang, H. Mei, L. Zhang, and Y. Chen. Architecture Based Deployment of Large-Scale Component Based Systems: The Tool and Principles. In Component-Based Software Engineering, 8th International Symposium (CBSE), volume 3489 of Lecture Notes in Computer Science. Springer, 2005.

[91] K.K. Lau and Z. Wang. Software Component Models. IEEE Trans. Software Eng., 33(10), 2007.

[92] L.Barbosa and J. Oliveira. State-based components made generic. Electronic Notes in Theoretical Computer Science, vol. 82, no.1, 2003.

[93] G. Leavens and M. Sitaraman, editors. Foundations of Component-Based Systems. Cambridge University Press, 2000.

[94] Jonathan Ledlie and Margo I. Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In INFOCOM [5], pages 1419–1430.

[95] BN Levine, C Shields, and NB Margolin. A survey of solutions to the sybil attack. Tech report 2006-052, University of Massachusetts Amherst, 2006.

[96] Y. Liu and S. Smith. A Formal Framework for Component Deployment. In 20th ACM Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2006.

[97] Zhiming Liu and He Jifeng, editors. Mathematical Frameworks for Component Software - Models for Analysis and Synthesis. World Scientific, 2006.

[98] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In Component Deployment, Third International Working Conference, CD 2005, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[99] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). IEEE Computer Society, 2006.

[100] Boris Mejías. CiNiSMO: Concurrent Network Simulator in Mozart-Oz. http://p2ps.info.ucl.ac.be/cinismo, 2008.

[101] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[102] S. Meng, B. K. Aichernig, L.S. Barbosa, and Z. Naixiao. A Coalgebraic Semantic Framework for Component-based Development in UML. Electr. Notes Theor. Comput. Sci., 122, 2005.

[103] S. Meng and L.S. Barbosa. Components as coalgebras: The refinement dimension. Theor. Comput. Sci., 351(2), 2006.

[104] M. Mikic-Rakic, S. Malek, and N. Medvidovic. Improving Availability in Large, Distributed Component-Based Systems Via Redeployment. In Component Deployment, Third International Working Conference, CD 2005, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[105] M. Mikic-Rakic and N. Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. In Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD'02), number 2370 in Lecture Notes in Computer Science. Springer, 2002.

[106] M. Miller and J. Shapiro. Paradigm regained: Abstraction mechanism for access control, 2003.

[107] Monika Moser, Seif Haridi, Tallat M. Shafaat, Thorsten Schütt, Mikael Högqvist, and Alexander Reinefeld. Transactional dht algorithms. Technical report, Zuse Institute Berlin, 2008.

[108] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2008.

[109] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. Cluster Computing, 9(2), 2006.

[110] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In Alexander Clemm, Lisandro Zambenedetti Granville, and Rolf Stadler, editors, DSOM, volume 4785 of Lecture Notes in Computer Science, pages 256–267. Springer, 2007.

[111] F. Puhlmann and M. Weske. Using the Pi-Calculus for Formalizing Workflow Patterns. In Business Process Management, 3rd Int. Conference, BPM 2005, number 3649 in Lecture Notes in Computer Science. Springer, 2005.

[112] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. Asynchronous, Hierarchical, and Scalable Deployment of Component-Based Applications. In Component Deployment, 2nd Int. Working Conference, CD 2004, number 3083 in Lecture Notes in Computer Science. Springer, 2004.

[113] R. Ramnath, R.H.C. Yap, and W. Yongzheng. WinResMon: A Tool for Discovering Software Dependencies, Configuration and Requirements in Microsoft Windows. Proceedings of the 20th conference on Large Installation System Administration Conference-Volume 20 table of contents, pages 14–14, 2006.

[114] A Rao, K Lakshminarayanan, S Surana, R Karp, and I Stoica. Load balancing in structured p2p systems. In In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems. Springer, 2003.

[115] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In SIGCOMM, pages 161–172, 2001.

[116] David D. Redell. Naming and Protection in Extendible Operating Systems. PhD thesis, University of California, Berkeley, September 1974.

[117] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses, 2005.

[118] M. D. Rodriguez-Moreno and P. Kearney. Integrating ai planning techniques with workflow management system. Knowledge-Based Systems, 15(5-6), 2002.

[119] J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols, 2007.

[120] J. Rosenberg, R. Mahy, and C. Huitema. Traversal using relay NAT (TURN). Internet-Draft http://www.jdrosen.net/midcom_turn.html, September 2005.

[121] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats), 2003.

[122] J. C. Rowanhill, G. S. Wasson, Z. Hill, J. Basney, Y. Kiryakov, J. C. Knight, A. Nguyen-Tuong, A. S. Grimshaw, and M. Humphrey. Dynamic System-Wide Reconfiguration of Grid Deployments in Response to Intrusion Detections. In High Performance Computing and Communications, 3rd International Conference, HPCC 2007, number 4782 in Lecture Notes in Computer Science. Springer, 2007.

[123] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, Middleware, volume 2218 of Lecture Notes in Computer Science, pages 329–350. Springer, 2001.

[124] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM Center Report BPM-06-22, BPM Center, 2006.

[125] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, number 4001 in Lecture Notes in Computer Science. Springer, 2006.

[126] Oskar Sandberg. Distributed routing in small-world networks. The Eighth Workshop on Algorithm Engineering and Experiments (ALENEX06), 2006.

[127] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Structured overlay without consistent hashing: Empirical results. In CCGRID, page 8. IEEE Computer Society, 2006.

[128] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Range queries on structured overlay networks. Computer Communications, 31(2):280–291, 2008.

[129] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. Journal of Peer-to-Peer Networking and Applications, 2008.

[130] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Handling network partitions and mergers in structured overlay networks. In Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing (P2P'07), September 2008.

[131] Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. On consistency of data in structured overlay networks. In Proceedings of the 3rd CoreGRID Integration Workshop, April 2008.

[132] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In Proceedings of the

3rd International ICST Conference on Scalable Information Systems (Infoscale'08). ACM, June 2008.

[133] Haiying Shen and Cheng-Zhong Xu. Hash-based proximity clustering for load balancing in heterogeneous dht networks. In IPDPS. IEEE, 2006.

[134] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In 30th International Conference on Software Engineering (ICSE 2008). ACM, 2008.

[135] P. Srisuresh. State of peer-to-peer (p2p) communication across network address translators (nats), 2008.

[136] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In SIGCOMM, pages 149–160, 2001.

[137] K. J. Sullivan, M. Marchukov, and J. Socha. Analysis of a Conflict between Aggregation and Interface Negotiation in Microsoft's Component Object Model. IEEE Trans. Software Eng., 25(4), 1999.

[138] C. Jennings Cisco Systems. Internet draft: Nat classification test results, 2005.

[139] C. Tibermacine, D. Hoareau, and R. Kadri. Enforcing Architecture and Deployment Constraints of Distributed Component-Based Software. In 10th International Conference Fundamental Approaches to Software Engineering (FASE), volume 4422 of Lecture Notes in Computer Science. Springer, 2007.

[140] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In OSDI, 2002.

[141] G. Valetto and G. E. Kaiser. Using Process Technology to Control and Coordinate Software Adaptation. In 25th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2003.

[142] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(1), 2003.

[143] A. van der Hoek. Configurable software architecture in support of configuration management and software deployment. In 21st International Conference on Software Engineering (ICSE). IEEE Computer Society, 1999.

[144] P. Van Roy and S. Haridi. Concepts, Techniques and Models of Computer Programming. MIT Press, 2004.

[145] Peter Van Roy. Self management and the future of software design. In Third International Workshop on Formal Aspects of Component Software (FACS '06. Springer ENTCS 182, September 2006.

[146] Peter Van Roy. Overcoming software fragility with interacting feedback loops and reversible phase transitions. In International Academic Research Conference on Visions of Computer Science, organized by the British Computer Society, September 2008.

[147] Peter Van Roy, Ali Ghodsi, Jean-Bernard Stefani, Thierry Coupaye, Alexander Reinefeld, Ehrhard Winter, and Roland Yap. Self management of large-scale distributed systems by combining peer-to-peer networks and components. Technical report, CoreGRID Network of Excellence, December 2005.

[148] Peter Van Roy, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. Self management for large-scale distributed systems: An overview of the selfman project. In Revised postproceedings of FMCO 2007, 2008.

[149] voip info.org. Nat survey, 2007.

[150] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. J. Network Syst. Manage., 13(2), 2005.

[151] Spyros Voulgaris, Maarten van Steen, and Konrad Iwanicki. Proactive gossip-based management of semantic overlay networks. Concurrency and Computation: Practice and Experience, 19(17):2299–2311, 2007.

[152] S. Y. Wang, C. L. Chou, and C. C. Lin. The design and implementation of the nctuns network simulation engine. Simulation Modelling Practice and Theory, 15(1):57–81, 2007.

[153] X. Wang, W. Li, H. Liu, and Z. Xu. A Language-based Approach to Service Deployment. In IEEE Int. Conf. on Services Computing (SCC). IEEE Computer Society, 2006.

[154] Yong Wang, Zhao Lu, and Junzhong Gu. Research on symmetric nat traversal in p2p applications. In ICCGI '06: Proceedings of the International Multi-Conference on Computing in the Global Information Technology, page 59, Washington, DC, USA, 2006. IEEE Computer Society.

[155] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. Nature: Macmillan Publishers Ltd, 1998.

[156] Weishuai Yang and Nael B. Abu-Ghazaleh. Gps: A general peer-to-peer simulator and its use for modeling bittorrent. In MASCOTS, pages 425–434, 2005.

[157] Y.Takeda. Symmetric nat traversal using stun, 2007.

[158] Yao Yue, Chuang Lin, and Zhangxi Tan. Analyzing the performance and fairness of bittorrent-like networks using a general fluid model. In GLOBECOM, 2006.

[159] Yingwu Zhu and Yiming Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. IEEE Trans. Parallel Distrib. Syst., 16(4):349–361, 2005.