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 Four (M37-M40)

Due date of deliverable:	Nov. 15, 2009
Actual submission date:	Nov. 15, 2009
Start date of project:	June 1, 2006
Duration:	40 months
Dissemination level:	PU

Contents

List of Figures

Chapter 1

Introduction

This set of final deliverables for the SELFMAN project covers the four-month extension (M37-M40). The main goal for this extension is to continue the momentum of SELFMAN: to provide additional results, to complete existing results, and to make a bridge toward future projects. We have made the following additional deliverables in the extension:

- D2.4: Simulation and emulation environment for Kompics P2P framework (partner KTH). This deliverable is part of task T2.2 and extends D2.1c.
- D3.4: Optimizations for self-managing global storage services (partner ZIB). This deliverable is part of task T3.2 and extends D3.2b.
- D4.5: Third report on self-configuration support (partner INRIA). This deliverable is part of task T4.1 and extends D4.1b.
- D5.9: Distributed mobile application on gPhone (partner UCL). This deliverable is part of task T5.8 and subsumes D5.8.
- D5.10: Design and analysis of Beernet, the Mozart structured overlay network implementation (partner UCL). This deliverable is part of task T5.3 and consists of the Ph.D. dissertation of Boris Mejias.
- D5.11: Self-protection mechanisms which provide spam resistance (partner NUS). This deliverable is part of task T5.6 and extends D5.6.
- D6.1d: Second project workshop. The workshop was held on Sept. 15, 2009 in conjunction with SASO 2009.

In addition to these deliverables, we have written an article that distills many insights and results of SELFMAN: "Software design with interacting

feedback structures and its application to large-scale distributed systems" (see Appendix A.1).[1] This article can be seen as a continuation of Deliverable D5.7 "Guidelines for building self-managing applications". The major insight is that large-scale distributed systems can be designed as a set of *weakly interacting feedback structures*, where a feedback structure is a hierarchy of interacting feedback loops that together maintain one global system property. The overall system specification is the conjunction of these properties. This provides the foundation of a realistic methodology for building self-managing applications. For example, the Scalaris design consists of six weakly interacting feedback structures, which is an enlightening alternative to the traditional layered approach where it is presented as three layers.

[1] Submitted to CACM on Oct. 1, 2009, upon invitation by editor-in-chief Moshe Vardi.

Chapter 2

D2.4: Simulation and emulation environment for the Kompics P2P framework

2.1 Executive summary

In this deliverable we present the prototype of a simulation and emulation environment for the evaluation of peer-to-peer systems built using the Kompics component framework. The prototype is released together with the Kompics framework and can be downloaded from `http://kompics.sics.se`.

The evaluation environment comprises of (1) a Java-based domain-specific language for specifying peer-to-peer experiment scenarios, (2) generic support for executing Kompics system in a reproducible simulation mode, (3) a generic discrete-event simulator encapsulated in a Kompics component, paired by an orchestrator component for the emulation mode, (4) system-specific simulator components, and (5) component architectures and patterns that enable the execution of Kompics P2P systems in either simulation, in local real-execution/emulation mode, or in distributed deployment.

A few examples of experiment scenarios can be found and followed at `http://kompics.sics.se/trac/wiki/P2P`. These include some scenarios for experiments with Chord and Cyclon in both simulation and real-time execution mode, as well as some BitTorrent simulation experiments.

This evaluation environment was successfully used as a teaching tool in the Distributed Computing, Peer-to-Peer and Grids course (ID2210) at KTH. The framework was used as support for student assignments which required the implementation and evaluation of P2P systems including a structured overlay network, a gossip-based overlay, and a content distribution network.

2.2 Contractors contributing to the Deliverable

KTH(P2) has contributed to this deliverable.

KTH(P2) KTH has implemented and tested and is still improving a prototype of the simulation and emulation environment for P2P systems built using the Kompics component framework. KTH gave a demonstration of this evaluation environment at the P2P'09 conference in September 2009.

2.3 Overview of the evaluation environment

Kompics is a component model targeted at building distributed systems by composing protocols programmed as event-driven components. Kompics components are reactive state machines that are executed concurrently by a set of workers. Components communicate by passing data-carrying typed events through typed bidirectional ports connected by channels. Ports are event-based component interfaces. A port type represents a *service* or a *protocol* abstraction. It specifies the types of events sent through the port in each direction. Components may encapsulate subcomponents.

The Kompics runtime supports pluggable component schedulers. The default scheduler is multi-threaded and executes components in parallel on multi-core machines. We use a single-threaded scheduler for reproducible simulation.

In this deliverable we present the prototype of a simulation and emulation environment for the evaluation of peer-to-peer systems built using the Kompics component framework. Kompics systems can be uniformly evaluated in large-scale reproducible simulation and distributed deployment, using both the same system code and the same experiment scenarios.

The evaluation environment comprises of (1) a Java-based domain-specific language for specifying peer-to-peer experiment scenarios, (2) generic support for executing Kompics system in a reproducible simulation mode, (3) a generic discrete-event simulator encapsulated in a Kompics component, paired by an orchestrator component for the emulation mode, (4) system-specific simulator components, and (5) component architectures and patterns that enable the execution of Kompics P2P systems in either simulation, in local real-execution/emulation mode, or in distributed deployment.

2.3.1 Defining an experiment scenario

We designed a Java domain-specific language (DSL) for expressing experiment scenarios for P2P systems. We call a *stochastic process*, a finite random sequence of events, with a specified inter-arrival time distribution. Here is an example scenario composed of 3 stochastic processes:

```
StochasticProcess boot = new StochasticProcess() {{
   eventInterArrivalTime(exponential(2000)); // 2s
   raise(1000, chordJoin, uniform(16)); }};  // 1000 joins
StochasticProcess churn = new StochasticProcess() {{
   eventInterArrivalTime(exponential(500));// 500ms
   raise(500, chordJoin, uniform(16));     // 500 joins
```

```
   raise(500, chordFail, uniform(16)); }}; // 500 failures
StochasticProcess lookups = new StochasticProcess() {{
   eventInterArrivalTime(normal(50));      // 50ms
   raise(5000, chordLookup, uniform(16), uniform(14)); }};
boot.start();                              // start
churn.startAfterTerminationOf(2000, boot); // sequential
lookups.startAfterStartOf(3000, churn);    // in parallel
terminateAfterTerminationOf(1000, lookups);// terminate
```

1000 peers join in a space of $0..2^{16}$. The inter-arrival time between 2 consecutive joins is exponentially distributed with a mean of 2s. A churn process starts 2s after. Every 500ms on average (exp), a new peer joins or an existing peer fails. In parallel with the churn process, 5000 lookups are initiated uniformly around the ring ($0..2^{16}$) for keys in the first ring quadrant ($0..2^{14}$). The experiment terminates 1s after lookups are done.

2.3.2 Experiment profiles

We can reuse the same experiment scenario definition to drive simulation or local real-time execution experiments, as well as remote experiments where the system nodes are distributed over the machines of a cluster (possibly running ModelNet) or a testbed like PlanetLab or Emulab.

During simulation and local execution we model the network at the message-level. In simulation, we execute the same system code built for deployment. Calls for the current system time are trapped and the current simulation time is returned. Simulation enables deterministic replay, debugging, reproducible results, and large-scale experiments without loss of accuracy.

We developed an infrastructure for deploying and executing distributed experiments. Experiment scenarios are locally interpreted by a Master component which coordinates a set of remote Slaves. Each Slave resides on a machine available for the experiment and it manages a set of system peers.

2.3.3 Summary

The prototype is released together with the Kompics framework and can be downloaded from `http://kompics.sics.se`. Examples of experiment scenarios can be found at `http://kompics.sics.se/trac/wiki/P2P`. These include some scenarios for experiments with Chord and Cyclon in both simulation and real-time execution mode, as well as some BitTorrent simulation experiments.

2.4 Papers and publications

The simulation and emulation environment for the Kompics P2P framework was demonstrated at the P2P'09 conference in Seattle, on Septemper 2009.

Building and Evaluating P2P Systems using the Kompics Component Framework

Cosmin Arad, Jim Dowling, Seif Haridi.

A demonstration abstract [3] was published in the P2P'09 conference proceedings. This demonstration abstract is included in Appendix A.2. The abstract outlines the contents of the demonstration and briefly introduces the Kompics component model and its simulation and evaluation framework for peer-to-peer systems.

The demonstration was accompanied by a poster which is included on the next page.

THE ADVENTURES OF SELFMAN

Information Society Technologies

KOMPICS

Reactive Component Model for Distributed Computing

Kompics components

» are **reactive** / event-driven programming model

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

» are **decoupled** by **publish-subscribe ports** and **channels**

» can be **composed** out of **encapsulated** subcomponents

» form **dynamically reconfigurable** architectures

» can form flexible **fault supervision hierarchies**

Experiment profiles

» local / distributed **deployment**: 1 peer / OS process

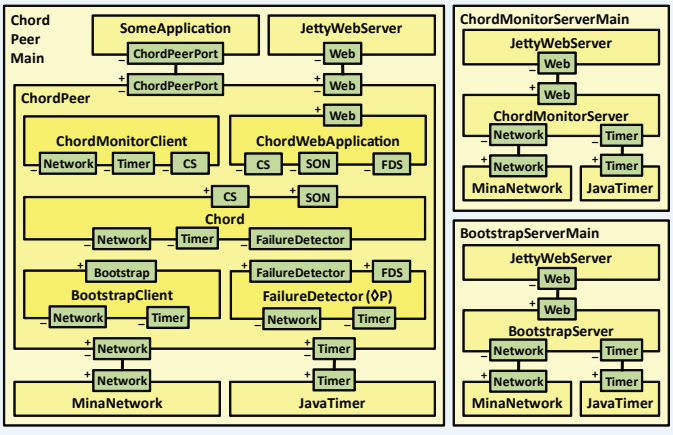
» local / distributed **execution**: multiple peers / OS process

» local **simulation**: multiple peers / OS process

» the deployment code is executable in **simulation** mode
- *using a deterministic single-threaded component scheduler*
- *replay debugging, reproducible results, large experiments*

» the same experiment scenario can be used for local simulation, local execution, or distributed execution

Chord deployment architecture



Peer-to-Peer framework

» reusable **components** and **patterns**
- *failure detection, bootstrap, monitoring*
- *communication, web-based interaction*

» implemented overlay **systems**
- *Chord, Kademlia, Cyclon, T-Man, BitTorrent*

» P2P **experiment scenario** definition **DSL**
- *specify & compose "stochastic processes"*
- *churn, system-specific actions, termination*

» generic **P2P simulator** / orchestrator
- *coupled with system-specific simulators*

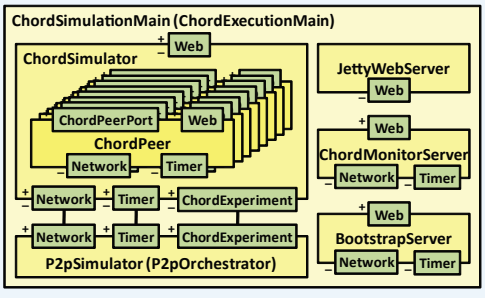
» reusable **latency** and **bandwidth** models

» Java implementation

Chord experiment scenario

```
StochasticProcess boot = new StochasticProcess() {{
  eventInterArrivalTime(exponential(2000));// ~2s
  raise(1000, chordJoin, uniform(16)); }}; // 1000 joins
StochasticProcess churn = new StochasticProcess() {{
  eventInterArrivalTime(exponential(500)); // ~500ms
  raise(500, chordJoin, uniform(16));      // 500 joins
  raise(500, chordFail, uniform(16)); }};  // 500 failures
StochasticProcess lookups = new StochasticProcess() {{
  eventInterArrivalTime(normal(50));       // ~50ms
  raise(5000, chordLookup, uniform(16), uniform(14)); }};
boot.start();                              // start
churn.startAfterTerminationOf(2000, boot); // sequential
lookups.startAfterStartOf(3000, churn);    // in parallel
terminateAfterTerminationOf(1000, lookups);// terminate
```

Chord simulation architecture



Documentation, examples, and source code at http://kompics.sics.se/

Distributed Computer Systems Group
Unit for Software and Computer Systems
Information and Communication Technology

Cosmin Arad (cosmin@sics.se)
Jim Dowling (jdowling@sics.se)
Seif Haridi (seif@sics.se)

Computer Systems Laboratory SICS

Chapter 3

D3.4: Optimizations for self-managing global storage services

3.1 Executive summary

The Global Storage Service developed within SELFMAN has been shown to scale as a storage backend for complex applications such as Wikipedia. Within this deliverable we present results from three different optimizations to the service:

- Storage services deployed over multiple geographically distributed data centers have recently been presented by companies such as Google, Yahoo! and Amazon. Data centers typically exhibit a hierarchy, while the SELFMAN storage service was originally developed for a flat topology. We have adapted the Global Storage Service to work better for a hierarchical structure by using a technique called prefix replication in combination with a gossip-based ring management algorithm.
- Applications such as Wikipedia have a non-uniform distribution of both storage and query workload. We have therefore investigated several active load balancing algorithms for reducing the moved load. In addition, a passive algorithm was developed to improve the load imbalance of the system by placing joining nodes according to the load.
- The distributed transaction algorithm used to guarantee strong consistency on storage operation normally need six steps to execute a transaction. With the goal of reducing the latency and traffic necessary to

perform a transaction, we have been able to improve the algorithm to four steps in the common case.

3.2 Contractors contributing to the Deliverable

ZIB(P5) Worked on load balancing, replication and improvements to the transaction layer on top of the structured storage.

3.3 Results

This section reports on the optimizations performed to the global storage service during the last phase of SELFMAN. We have mainly focused on three sub-problems: load balancing, transaction performance and data center detection and replication. The main results are summarized in this section and accompanied by a short introduction to the respective publications in Section 3.4.

3.3.1 Adapting the Storage Service to Data Centers

With the advent of cloud computing, applications and services are deployed over multiple, geographically separated data centers to ensure availability. Using the Global Storage Service in a multi-data center deployment is sub-optimal due to several basic assumptions in Structured Overlay Networks. 1) When a data center is unavailable the resulting node failures are correlated, 2) Bandwidth and latency between nodes inside a data center is often several magnitudes better than between nodes in different data centers and 3) the topology of a Structured Overlay Network is flat while a multi data center architecture is hierarchical.

We have addressed these problem using three different methods. First, we use item prefixes to be able to place replicas according to specific policies. For example, a majority of replicas within the same data center improves read/write performance. Second, we introduced a gossip-based ring maintenance algorithm which improves the repair of a ring for correlated failures. Finally, we developed a fully distributed data center detection algorithm to minimize the number of unnecessary inter-data center links [25, 24].

Fig. 3.1 illustrates how replicas can be placed in a global ring structure to ensure both, low latency and high availability for applications with different geographical user communities, like several instances of Wikipedia in different languages.

Fig. 3.2 shows the actual size and location (circles) and the estimated locations of the data centers (diamonds) that were identified by our algorithm after $1.5 \log_2 N$ communication rounds, with N being the number of nodes. We simulated 100 nodes based on the Grid 5000 node distribution and plotted all centroids identified by all nodes. After $1.5 \log_2 N$ rounds the error is already relatively small.

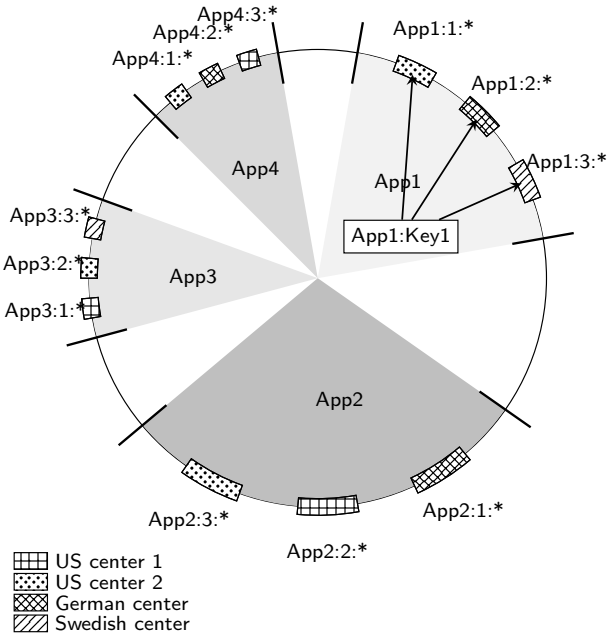


Figure 3.1: Geographic distribution of replicas using prefix replication.

3.3.2 Load Balancing

The goal of load balancing is to improve the fairness regarding storage as well as network and CPU-time usage between the nodes. Imbalance mainly occurs due to: 1) non-uniform key distribution, 2) skewed access frequency of keys and 3) node heterogeneity. First, by supporting range-queries as in the SELFMAN global storage service, an order-preserving hash function is used to map keys to the overlay's identifier space. With a nonuniform key distribution a node can become responsible for an unfair amount of items. Second, keys are typically accessed with different popularity which creates uneven workload on the nodes. The third issue, node capacity differences, also impacts the imbalance. For example, a low capacity node gets overloaded faster than a high capacity node.

A load balancing algorithm can have two modes: *active*, which triggers a node already part of the overlay to balance with other nodes and *passive*, which places a joining node at a position that reduces the overall system imbalance. Without passive balancing, system churn continually deteriorates the system balance since nodes are joining at random node IDs. We have devised an algorithm for both passive and active mode which tries to place the joining node at the best position depending on both storage and workload. Figure 3.3 shows the imbalance for the passive/active algorithm with basic

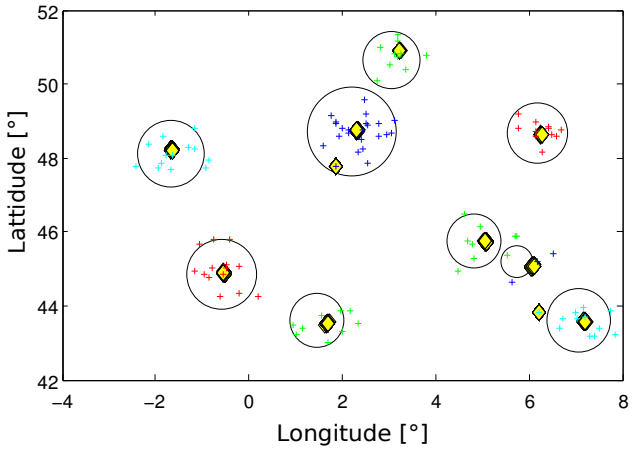


Figure 3.2: Geographic distribution of replicas using prefix replication.

hashing.

We continued to investigate the possible improvements to active load balancing algorithms by adding global estimates [8]. Global estimates are created through gossip and is including variations of the average load, the standard deviation and both combined. This information was used to extend Karger's [11] active randomized load balancing algorithm. Figure 3.4 shows the performance of the different algorithms using the load distribution from all English Wikipedia page titles. We conclude that by using global estimates we are able to move half the load or less as without the estimates. Further results from this work is presented in [13]. We are currently improving the simulation model by adding churn. In addition, we are also investigating a new design that, unlike DHTs, separate the storage and routing layer [10].

3.3.3 Transaction Performance

In the transaction algorithm presented in Deliverable 3.1b and c, there are six steps necessary to perform a transaction. We have improved this algorithm such that in the common case only four steps are needed. For a fast transaction validation, each node in the overlay permanently maintains a list of $r - 1$ other nodes, that can be used as Replicated Transaction Managers (RTMs). The location of these nodes could be according to the scheme of symmetric replication. Once these nodes are located, they are maintained through the use of failure detection. The short version of the protocol is shown in Figure 3.5.

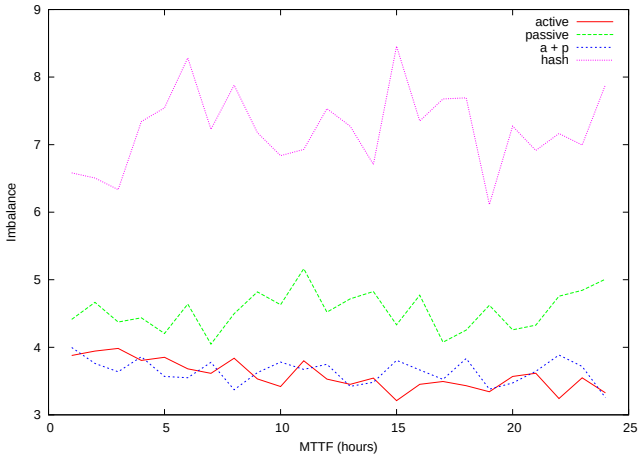


Figure 3.3: Imbalance with decreasing levels of churn.

3.4 Papers and Publications

Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers

Thorsten Schütt, Alexander Reinefeld, Florian Schintke, Marie Hoffmann. Presented at the IEEE Internationl P2P Conference 2009 (see A.3, [25]).

> We present a distributed algorithm for identifying the location of data centers and their relative sizes. This topology information can be used in P2P systems to improve the routing performance, replica placement, or job scheduling.
>
> The algorithm uses gossiping with local agglomerative clustering. It is robust to failures and it correctly identifies outliers that are caused, e.g., by temporarily overloaded nodes or network failures. We present empirical results on the Grid 5000 testbed.

Self-Adaptation in Large-Scale Systems: A Study on Structured Overlays Across Multiple Datacenters

Thorsten Schütt, Alexander Reinefeld, Florian Schintke, Christian Hennig. Presented at the IEEE SELFMAN Workshop at SASO 2009. (see A.4, [24]).

> With the recent focus on cloud computing a new type of system topology came up: clusters in geographically distributed data-

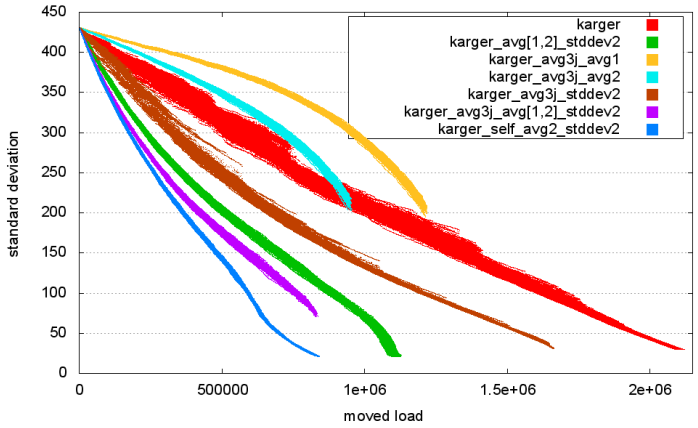


Figure 3.4: The systems load in standard deviation versus the amount of load moved for variations of Karger with global estimates.

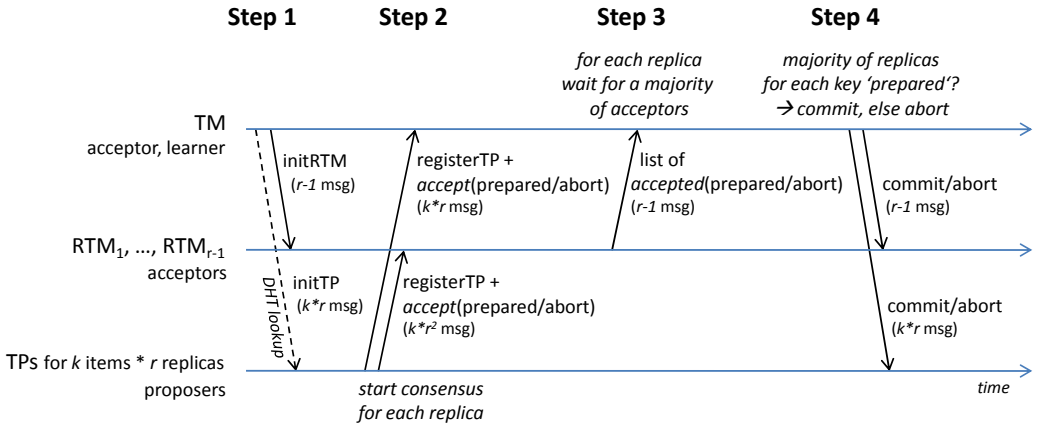


Figure 3.5: Timeline diagram of a fast transaction commit.

centers that are connected by high-latency networks. Current structured overlay networks (SONs) are not well prepared for such environments with heterogeneous network performance and correlated node failures.

We show how the beneficial features of SONs, namely selfmanagement, scalability, adaptability, and fault tolerance can be exploited for multi-datacenter environments. We present selfadaptive replica placement policies and latency-optimized routing for SONs on multiple datacenters. Empirical results of our gossip-based ring maintenance protocol demonstrate its ability to cope with correlated node failures and network partitioning.

Enhanced Paxos Commit for Transactions on DHTs

Florian Schintke, Alexander Reinefeld, Seif Haridi, Thorsten Schütt. ZIB Technical Report ZR-09-28 (see A.5, [23]).

> Key/value stores which are built on structured overlay networks often lack support for atomic transactions and strong data consistency among replicas. This is unfortunate, because consistency guarantees and transactions would allow a wide range of additional application domains to benefit from the inherent scalability and faulttolerance of DHTs.
>
> The Scalaris key/value store supports strong data consistency and atomic transactions. It uses an enhanced Paxos Commit protocol with only four communication steps rather than six. This improvement was possible by exploiting information from the replica distribution in the DHT. Scalaris enables implementation of more reliable and scalable infrastructure for collaborative Web services that require strong consistency and atomic changes across multiple items.

Towards Explicit Data Placement in Scalable Key/Value-stores

Mikael Högqvist, Stefan Plantikow. Presented at the IEEE SELFMAN Workshop at SASO 2009 (see A.6, [10])

> Distributed key/value-stores are a key component of many large-scale applications. Traditionally they have been designed using Distributed Hash Tables (DHTs). DHTs, however, setup a tight coupling between the naming of nodes and assignment of keys to nodes which limits application control over data placement.
>
> We propose using small amounts of shared state in a semi-centralized architecture for more flexible data placement by introducing explicit mapping between keys and nodes via an indirection layer (blockspace). Our design is based on a membership layer that provides O(1) routing thereby targeting interactive applications. We evaluate a centralized and decentralized approach showing that both have relatively low overhead and provide efficient load balancing.

Active/Passive Load Balancing with Informed Node Placement in DHTs

Mikael Högqvist, Nico Kruber. To be presented at the IFIP International Workshop on Self-Organizing Systems (IWSOS) 2009 (see A.7, [9])

> Distributed key/value stores are a basic building block for large-scale Internet services. Support for range queries introduces new challenges to load balancing since both the key and workload distribution can be non-uniform.
>
> We build on previous work based on the power of choice to present algorithms suitable for active and passive load balancing that adapt to both the key and workload distribution. The algorithms are evaluated in a simulated environment, focusing on the impact of load balancing on scalability under normal conditions and in an overloaded system.

Generic Self-Healing via Rejuvenation: Challenges, Status Quo, and Solutions

Artur Andrzejak. Presented at the IEEE SELFMAN Workshop at SASO 2009 (see A.8, [1])

> Software rejuvenation - in its simplest form a restart of a component or a program - is an efficient and universal approach for ad hoc healing of certain complex systems such as SOA components, telecommunication systems, and servers in data centers. Despite of its advantages this technique has not been widely deployed in other scenarios. The reasons are several shortcomings including loss of application availability and loss of working data due to a restart, and a lack of standardized support in operating systems, middleware, and component frameworks. In this position paper we argue that even partial remedies to these problems can turn rejuvenation into a powerful self-healing tool applicable to a larger variety of scenarios. We discuss rejuvenation-related problems, overview existing solutions, and propose a set of efficient architectural approaches which can pave the way to a universal adoption of this technique.

DHT Load Balancing with Estimated Global Information

Nico Kruber. Master Thesis at Humboldt University Berlin, 2009 (see A.9, [13])

One of the biggest impacts on the performance of a Distributed Hash Table (DHT), once established, is its ability to balance load among its nodes. DHTs supporting range queries for example suffer from a potentially huge skew in the distribution of their items since techniques such as consistent hashing can not be applied. Thus explicit load balancing schemes need to be deployed. Several such schemes have been developed and are part of recent research, most of them using only information locally available in order to scale to arbitrary systems.

Gossiping techniques however allow the retrieval of fairly good estimates of global information with low overhead. Such information can then be added to existing load balancing algorithms that can use the additional knowledge to improve their performance. Within this thesis several schemes are developed that use global information like the average load and the standard deviation of the load among the nodes to primarily reduce the number of items an algorithm moves to achieve a certain balance. Two novel load balancing algorithms have then been equipped with implementations of those schemes and have been simulated on several scenarios. Most of these variants show better balance results and move far less items than the algorithms they are based on.

The best of the developed algorithms achieves a 15-30% better balance and moves only about 50-70% of the number of items its underlying algorithm moves. This variation is also very robust to erroneous estimates and scales linearly with the system size and system load. Further experiments with self-tuning algorithms that set an algorithms parameter according to the systems state show that even more improvements can be gained if additionally applied. Such a variant based on the algorithm described by Karger and Ruhl shows the same balance improvements of 15-30% as the variant above but reduces the number of item movements further to 40-65%.

Chapter 4

D4.5: Third report on self-configuration support

4.1 Executive summary

This deliverable reports on a preliminary study towards a framework and infrastructure for component deployment in a WAN-based unstructured overlay network. It first describes a NAT-resilient gossip peer-sampling protocol which is key to build unstructured overlays in a realistic Internet environment, with multile forms of NAT coexisting. It then describes a peer-to-peer middleware, called Salute, which aims to support the concurrent deployment of distributed applications on heterogeneous computing resources distributed over the Internet. Salute combines new and well-known gossip protocols to build and maintain its application supporting infrastructure.

4.2 Contractors contributing to the Deliverable

INRIA(P3)has contributed to this deliverable.

4.3 Results

The work reported in this deliverable finds inspiration from the Managing Clouds manifesto [4], which describes the challenge of building a general-purpose framework for delivering and supporting in a self-organized fashion distributed applications running independently over a large scale dynamic pool of computing resources. The Managing Clouds paper also highlights key elements of such a framework: application suite descriptions, whose purpose is "to define what applications are running on the network and what resources should be assigned to these applications", and a middleware that is "responsible for keeping the system in a state that corresponds to the description", and that typically should comprise the following services: a bootstrapping service, responsible for starting up and application from scratch, a chrun handling service for assisting the application in handling chrun and failures, and a slicing service for assigning the right subset of nodes to aplications, according to application requirements specified in application descriptions.

In this report we present two pieces of works that contribute to the Managing Clouds challenge. The first one aims to build an (unstructured) overlay network in a realistic Internet environment with NAT (Network Address Translation) devices. Such an overlay can in turn be used as the network basis for delivering applications on clouds (i.e. dynamic pools of computing resources). The second one presents indeed a preliminary study towards a peer-to-peer middleware that supports the deployment and execution of independent applications on an unstructured overlay network. The central function of our peer-to-peer middleware, called Salute, is a dynamic slicing service that relies on a peer sampling service to maitain the overlay network of computing resources.

4.3.1 NAT-resilient gossip peer-sampling

Gossip protocols have received an increasing attention over the past decade because they are robust, simple and highly resilient to churn. Gossip peer-sampling protocols are extensively used to build and maintain unstructured overlay networks. They typically provide peers with a random sample of the network and maintain connectivity in highly dynamic settings. They rely on the assumption that, at any time, each peer is able to establish a communication with any of the peers of the sample provided by the protocol. Yet, this ignores the fact that there is a significant proportion of peers that now sit behind NAT devices (70% is a fair ratio in the current Internet), preventing direct communication without specific mechanisms. This has been largely ignored so far in the community. Our experiments (reported in Appendix

A.10) demonstrate that the presence of NATs, introducing some restrictions on the communication between peers, significantly hurts both the randomness of the provided samples and the connectivity of the p2p overlay network, in particular in the presence of high rate of peers arrivals, departures and failures (aka churn).

To deal with these issues we have developed a NAT-resilient peer sampling protocol, called *Nylon*, which is described in Appendix A.10. *Nylon* is a fully decentralized protocol, that spreads evenly among peers the extra load caused by the presence of NATs (to relay messages to nodes located behind NAT). *Nylon* also ensures that a peer can always estbalish a communication, and therefore intiate a gossip, with any peer in its sample. This is achieved through a variant of the hole punching method (which is traditionally used to traverse NATs) that establishes a path of relays (rendez-vous peers, i.e. public peers that can exchange messages with both a source and a destination peer) between peers.

Simulation results show that *Nylon* is highly resilient to churn (it tolerates the departure of 50% of the peers without partitioning), and that it fulfills its objective of evenly spreading the overhead induced by NATs between public and natted peers.

4.3.2 Adaptive deployment over unstructured overlays

The Salute framework manages the deployment of multiple independent applications over an unstructured overlay network (the reserve overlay), built and maintained by a gossip peer sampling protocol. Essentially, Salute provides each application with a dynamic subset of of nodes from the underlying overlay network, called a *slice*, where the application can be deployed and executed.

A request to deploy an application with Salute contains a slice specification, which describes the required number of nodes in the slice, together with their characteristics (capacities and availability profile). Salute comprises two key elements: a resource reservation service and a resource profiling service. The resource profiling service provides a categorization of nodes based on their capacities and their availability history. The resource reservation service itself relies on three services:

- The request propagation service, that broadcasts the reservation request among the nodes in the reserve overlay, so as to book enough resources to build the requested slice (a booked peer cannot belong to more than one slice). The request propagation service is typically implemented by an epidemic broadcast.

- The counting service is used during the lifecycle of a slice: it is used during request propagation to decide when the requested slice is complete, and during the lifetime of the slice in order to detect if the slice size falls under a certain threshold because of churn. The counting service is typically implemented using a gossip aggregation protocol.

- The peer sampling service in charge of maintaining the supporting overlay and the slices.

The protocol that implements the resource reservation service builds, monitors and maintains dynamic slices matching requested slice specifications. The reservation protocol also avoids deadlocks between concurrent slice requests by means of a priority mechanism.

The Salute framework, presented in more detail in Appendix A.11, constitutes only a preliminary design. We have not evaluated it yet, but Appendix A.11 discusses the evaluation criteria.

4.4 Papers and publications

The work on NAT-resilient gossip peer-sampling reported in this deliverable has been published as the following paper:

- Anne-Marie Kermarrec, Alessio Pace, Vivien Quéma and Valerio Schiavoni. NAT-resilient Gossip Peer Sampling. 29th International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society, June 2009.

Chapter 5

D5.9: Distributed mobile application on gPhone

5.1 Executive summary

This deliverable presents the development of DeTransDrawid, a collaborative drawing application showcasing the Beernet P2P network with transactional support [15]. DeTransDrawid allows several users to simultaneously edit the same drawing while guaranteeing both the coherence of the drawing and high performance of the application. We call the resulting application DeTransDrawid since it is based on the original DeTransDraw application presented in Appendix A.12. This deliverable describes the work done to implement DeTransDrawid on the gPhone, which in our case is an HTC Magic smartphone running Google's Android operating system. We explain each step, problem and choice taken during the development of the application.

A major problem we have had to face is the separation between Oz and Java. The application is running half in a Mozart emulator and half with the Android API. The communication between both parts is an important key factor for the performance of the application. The more communications there are, the worse is the performance.

5.2 Contractors contributing to the Deliverable

UCL(P1) has contributed to this deliverable.

UCL(P1) has ported the Mozart environment to Android and implemented a collaborative graphic drawing application in Mozart which runs on the Android operating system.

5.3 Introduction

We present the development of DeTransDrawid, a distributed, decentralized, interactive application for collaborative drawing, running on an Android operating system. We first present the specification that we want this application to fulfill, essentially a coherent state and good responsiveness and how to achieve them. We move on to describe the architecture of the solution retained and the structure of the Android target platform. We then highlight some of the most interesting parts of the implementation. Finally, we conclude by showing what remains to be done in term of polishing in order to provide a fully functional application.

5.4 Specification

DeTransDrawid is a collaborative drawing tool that can be used by several users simultaneously on the same drawing. The operation of the tool is explained in Appendix A.12. In this section we explain why DeTransDrawid is an interesting case study for the Beernet transactional storage.

As a distributed collaborative and interactive drawing application, DeTransDrawid has two goals: (1) keeping a coherent state and (2) being responsive. These two goals seem to be in conflict as the obvious solution for coherent state is to do a time-consuming global lock on the state to be modified while the obvious solution for responsiveness is to optimistically allow the user to do any modification she wants without regard for coherence.

In DeTransDrawid the drawing is made of a set of shapes. Each shape has its own set of parameters such as position, size, color, etc. An edit cycle consist of the user selecting a first shape, repeatedly modifying parameters and selecting more shapes and finally deselecting all shapes. For the drawing to remain coherent for all users, each edit cycle is made into a transaction on the global, distributed storage.

The tension between the two goals is now converted into a choice of scheduling policy. An optimistic scheduling will give adequate reponsiveness and coherence but at the price of aborted transactions, which can be very frustrating for the user when her very complex transaction involving lots of different shapes and fine-tuning of the parameter is aborted in favor of a trivial change to one shared shape by another user. A pessimistic scheduling will of course frustrate the user that will have to wait for locks before she can make any change, as insignificant as it could be (e.g., correcting a typo).

The transactions used in DeTransDrawid are both optimistic and pessimistic. They are optimistic in that the user is immediately allowed to select and modify shapes and her edits can be aborted. They are pessimistic in that as soon as the user select a shape, the correponding lock is requested and edits are aborted as soon as a lock is known to be unavailable. Figure 5.1 shows the precise state diagram. A user is initially holding no locks and therefore in the *No lock* state. When she selects some shapes, she sends requests for the corresponding locks and waits for them in the *Asking for locks* state. On abort, all locks are released ans she returns to the *No lock* state. If all locks are received, she moves on to the *Got locks* state from which she can commit her transaction (by unselecting all shapes) and return to the *No lock* state or ask for more shapes to be locked and return to the *Asking for locks* state, re-exposing herself to the risk of an aborted edit.

This optimistic-with-eager-locking approach is complemented by visual feedback to the user which can therefore know wheter the edit he is making

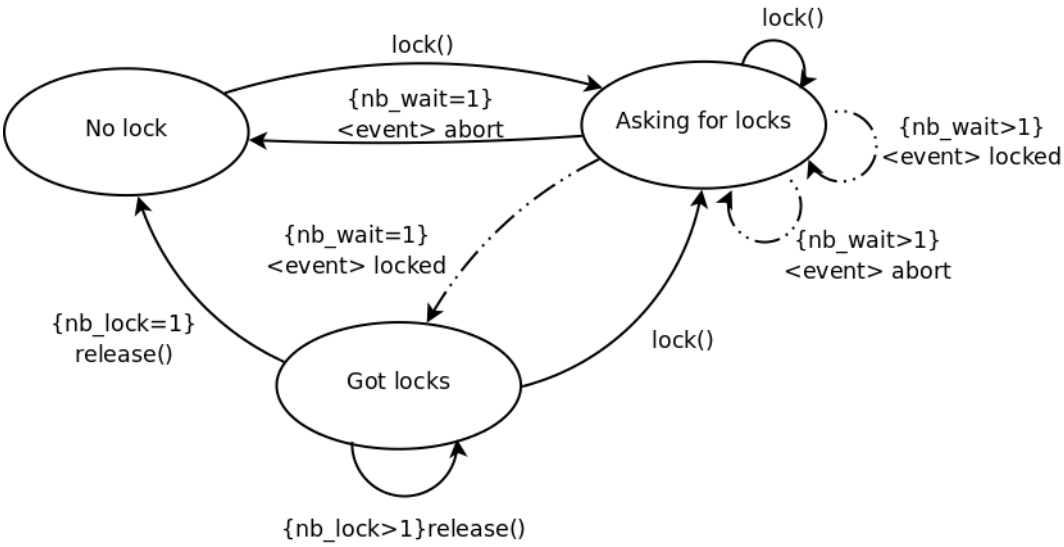


Figure 5.1: State diagram of a user

is still susceptible to rollback. This feature puts the user in control. He can decide to optimistically edit a shape to correct a simple mistake without delays or to pessimistically wait for the guarantee to be able to complete a major reorganisation of the drawing. Figure 5.2 shows the visual feedback. The user on the left as just selected the big rectangle and can optimistically modify it since the handles for it are red. However we know that this transaction will be aborted as the user on the right as the lock for this rectangle (as indicated by the black handles).

Finally, it was decided to structure the participants in two classes, since the cost and feeble bandwidth of smartphones makes them bad candidates for routing, only computers are participating as full peers. The smartphones attach themselves to one of these peers and do all their communication through it. However, this link is as reconfigurable as the links in the P2P network itself and the phone can reconnect to other peers should its connection be broken.

5.5 Architecture

Since the Beernet framework provides decentralized, replicated storage with a transactional layer supporting both eager-locking and notification of the locking status, it was choosen as the underlying platform for DeTransDrawid. The application is thus written in Oz and runs on the Mozart platform.

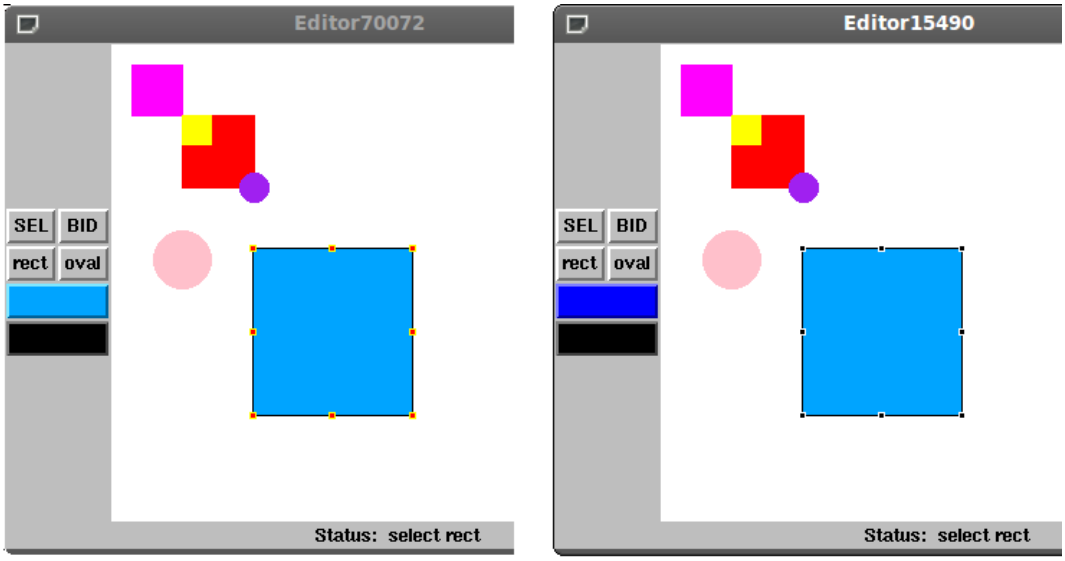


Figure 5.2: On the left, the user is in *Asking for locks* state. On the right, the user is in *Got locks* mode.

DeTransDrawid was ported to two different platforms. The first one is traditional desktop computers and similar platforms. The second is Android [20], the new cell-phone operating system developed by Google. Although based on a Linux kernel, this operating system is quite different from a standard Linux distribution.

An Android application is structured as a set of user visible screens (called tasks), background services, content providers, and event listeners. All of them are instantiated by the system based on user requests or environmental changes. All of them can be destroyed at certain points and recreated later without any visible change to the user. This is not transparent for the developer. As an example, Figure 5.3 illustrates the lifecycle of a task.

The current version of DeTransDrawid is made of a main task showing the drawing and a few accessory tasks to allow the user to connect to a drawing.

Upcoming versions should allow the application to be launched from other contexts such as the user clicking a link in a webpage or email, on receiving a special text message or by scanning special barcodes.

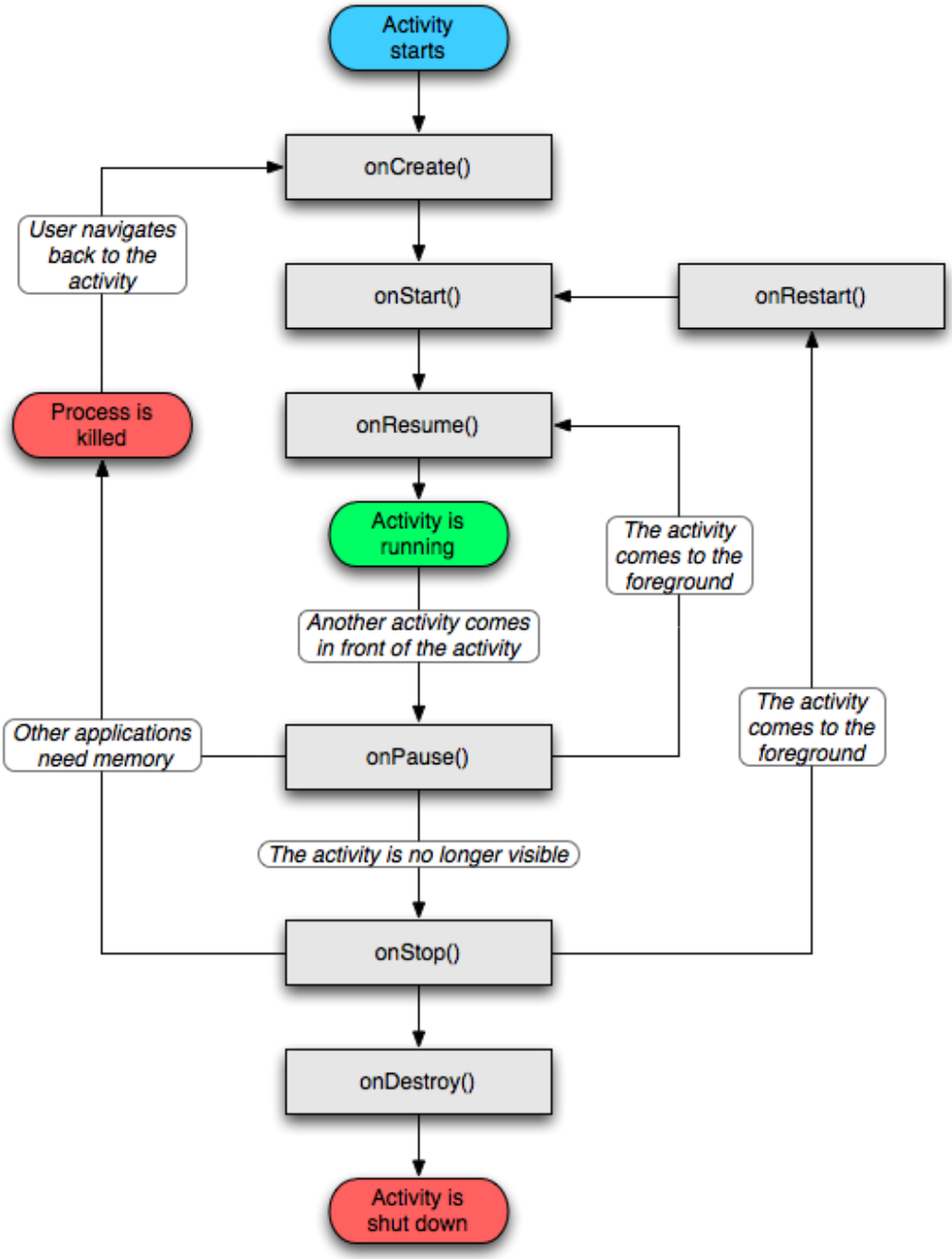


Figure 5.3: State diagram of an Android task

5.6 Implementation

In the very beginning of the development of the application, we first needed to port the Mozart environment on the platform. The application will be architecture in a Model-View-Controller pattern. The model and controller will be implemented in Mozart environment on top of the Beernet and the view will be displayed thanks to the Android API.

DeTransDrawid uses the Beernet ring. It first connects to a node of the ring and use it for network communication. DeTransDrawid may connect to an existing node of the ring or create a new node and join the ring. The first example appears in the Figure for node *P3* while the second is like node *P7*. As you can see on Figure 5.4, only the Oz part receives network messages. The GUI part is network independent. There is communication between the logic part and the ring, and between the logic part and the GUI part.

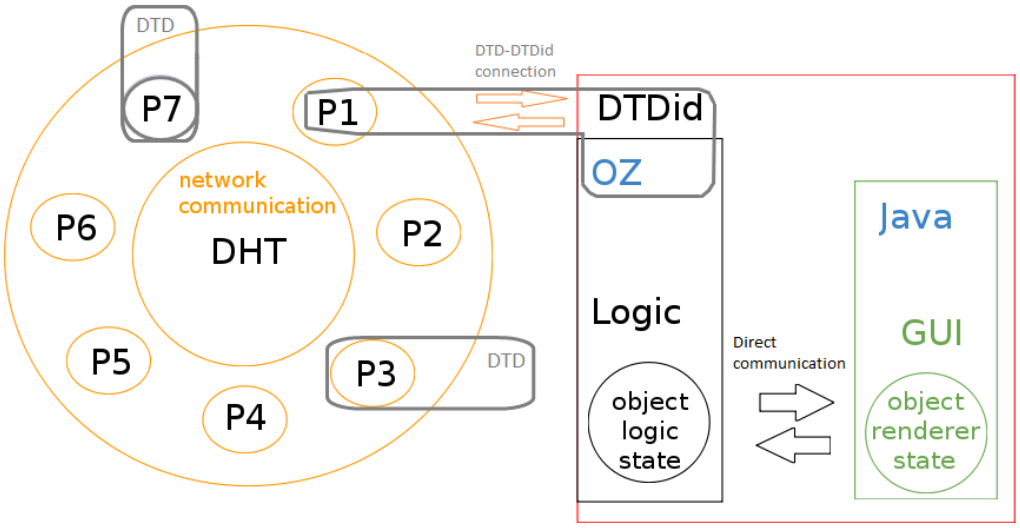


Figure 5.4: Structure of DeTransDrawid

The desktop version of DeTransDrawid is implemented in Oz using QTk for the graphical user interface and Beernet for the decentralized storage and transactional support.

The Android version of DeTransDrawid is implemented in Oz but since Tk is not available on Android, the GUI part has to be rewritten. The only supported language for user interfaces on Android is Java, running on the Dalvik virtual machine.

Porting DeTransDraw to Android thus required the following subtasks:

- Porting the Mozart environment to Android. This in turn was made more difficult by the following constraints:
  - Need to cross-compile the executable. The Google supported way to develop native application for the Android platform is only by using a specific cross-compiler. The build system for Mozart had only some support for cross-compilation for the Windows target platform.
  - Need to integrate in the Android build-system. Developement of native application for Android is only possible by using the Google provided build infrastructure which relies on non-recursive makefiles.

    While this is a more modern approach, it required adapting the whole build infrastructure of the Mozart platform.
  - Lack of a complete implementation of C++. The supported languages for native developement on Android are C and C++ but the C++ language is provided with extremely minimal library support, no runtime type information and no exception support.
  - Deficient loader. The loader of native application on the Android platform, part of the Bionic C library, is still in its infancy. This implies that certain constructs generated by the compiler are unusable and need to be worked around. It also means that dynamic library support is extremely limited. Mozart uses dynamic libraries to reduce it's startup time by offloading many modules to external libraries. Another consequence of this defficient linker is the requirement to use Android version 1.5 or better. The linker in previous versions is completely unusable.
- Creating a bridge to access the Java API for user interfaces from an Oz application.

  This bridge makes available the complete Java API accessible through reflection in the Java language by providing an Oz API at the same level of abstraction. Internally, all operations are serialized, the communication between the Oz virtual machine and the Java virtual machine is done over a TCP link and the two garbage collectors are made to cooperate.
- Developing a GUI for DeTransDraw using the primitives provided by the Java API. Since the level of abstraction in QTk is much higher than the one provided by the Java API of Android for the developement

of user interfaces, the GUI of DeTransDraw is less powerful on the Android version than on the desktop version. For now the Android version offers a read-only access to the drawing. Also, as a concession to speed and due to the parallel developement of all these parts, some elements of the GUI are developed directly in Java and not in Oz.

5.6.1 Porting Mozart on Android

The Mozart environment's porting was the first step of the development of DeTransDrawid. To allow Mozart interacting with the device, we need communication between Mozart and the Java API.

The application starts Mozart emulator and open a port to allow communication between both parts. We have created a project named *javaaccess* which implements the reflection mechanism for the communication.

We have developed the application in Eclipse environment, with the Android Development Toolkit (ADT) plugin. To run an application you then need two projects.

The first project is a Java project that we have called javaaccess. You can see on Figure 5.5 how to create this project.

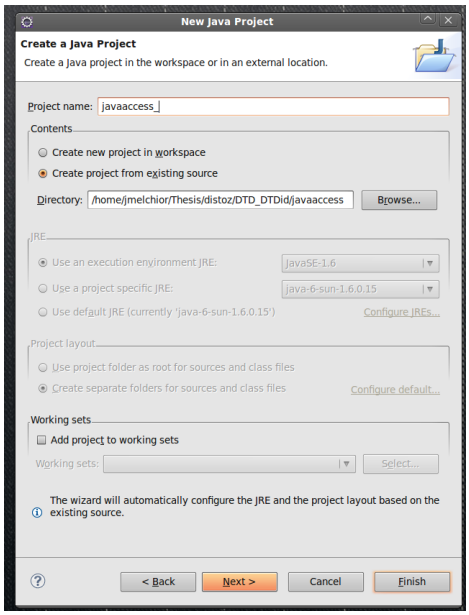


Figure 5.5: Creation of javaaccess project in Eclipse

You also need the Android project that we have called DeTransDrawId. You can see on Figure 5.6 how to create this project.

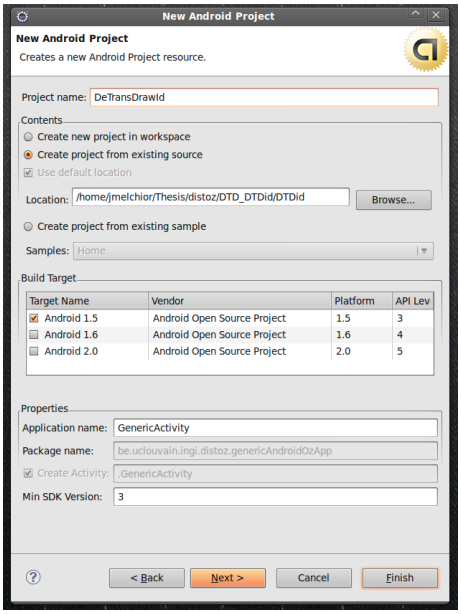


Figure 5.6: Creation of DeTransDrawId project in Eclipse

The result of the creation of this project is shown on figure 5.7.

To run Mozart on the device, we need to use the Java API to copy the cross-compiled environment on the device and then to run it before trying to use it. The first part is implemented in *ZipExpander.java* file. While the second part is implemented in *GenericApplication.java* file. The expander is started with the following command :

```
ZipExpander.expand(dir, getAssets().open("emu.zip"));
```

The file *emu.zip* contains the cross-compiled environment of Mozart. We put it in the assets of the project to ensure that the application will always find the environment to copy, unzip and run it on the device. After the environment has been unzipped from this file, Mozart emulator is started with the following commands :

```
ServerSocket s = new ServerSocket(4545);
File emu = new File(dir, "ozEmulator");
if (emu.exists()) {
   ozP = Runtime.getRuntime().exec(
   new String[] { emu.getCanonicalPath(), "-init",
   new File(dir, "Init.ozf").getCanonicalPath(), "-u",
                "x-oz://system/android/Main.ozf", "--" },
```

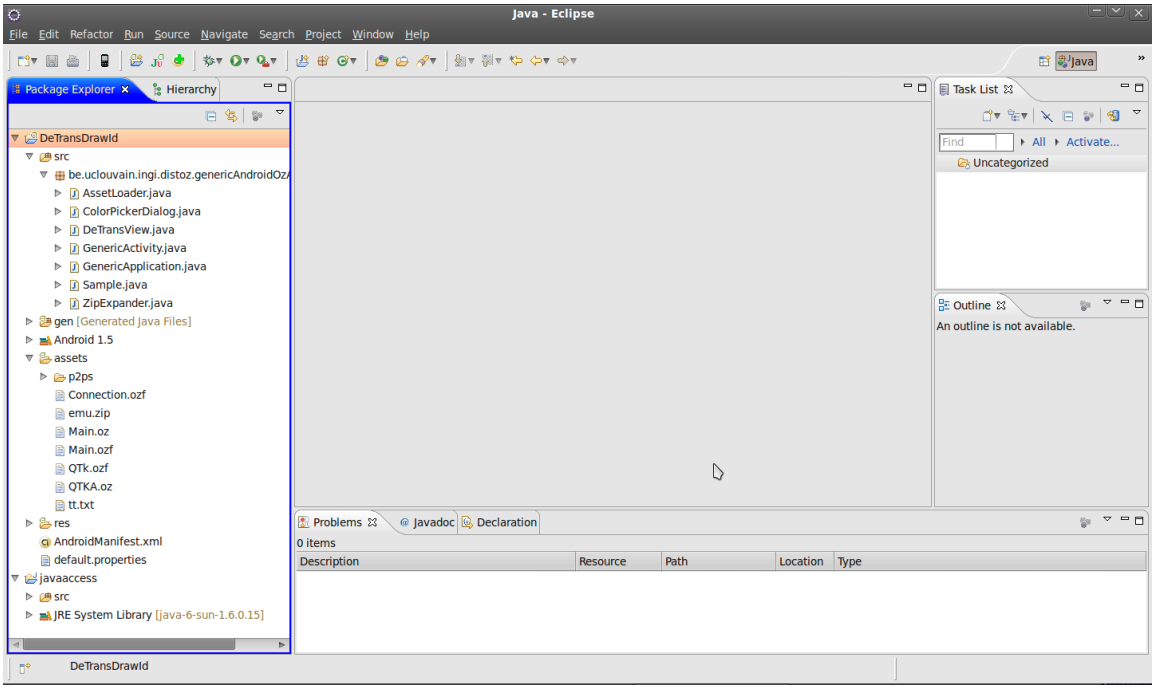


Figure 5.7: Eclipse environment after creating both projects

```
   new String[] {
        "OZ_ANDROID_PORT=4545" }, dir);
}
```

5.6.2 Communication Oz-Java

When there are interactions from Mozart environment, the classes and objects are brought to Mozart in a lazy way. For example the following command :

```
{{{J.c 'java.lang.System'}.get err}.p println(
string("ELSE *:* Hello World/Activity."#{Label Msg}))}
```

The first statement is {J.c 'java.lang.System'} which mean that we want the access the *System* class from java.lang package. For example, if we got the result in a variable named *R1*, the second statement is {R1.get err}. We want to get the object stored in the err variable from the class. This new result, R2, is now part of the third statement which is {R2.p println(Str)}. The method *println* from err is now called with the parameter Str. The variable Str is the Oz string

```
"ELSE *:* Hello World/Activity."#{Label Msg}
```

which needs to be converted to a Java string by the reflection process. In order to help this convertion, we need to explicitly note that Str is a string with *string(Str)* statement. This complex command in Oz correspond in Java to this statement (i.e : with onCreate method as label) :

```
System.err.println("ELSE *:* Hello World/Activity.onCreate");
```

To create a Mozart application on the device, you need a file named *Main.oz*. This file has to be a functor importing at least *'x-android:///J'* which is a Java object to access the Java part. It also need to export the *MainActivity* procedure that will be called when the application is created from Android.

```
functor
import
   J  at 'x-android:///J'
export
   MainActivity
```

The MainActivity procedure is called every time the Java part of the application notifies the Oz part with a handler. The procedure get the *Msg* as parameter. The code should look like that :

```
fun{MainActivity Msg}
   case Msg
   of onCreate(!BundleC#SavedInstanceState) then
      ...
   [] onOwnMethod1(_#Var1) then
      ...
   [] onOwnMethod2(_#Var2 _#Var3) then
      ...
   [] onDestroy() then
      ...
   [] onStop() then
      ...
   else
      {{{J.c 'java.lang.System'}.get err}.p
       println(string("ELSE *:* Hello World/Activity."#{Label Msg}))}
   end                % End of case
void(x)
end                   % End of MainActivity
```

Here is as example, the onDestroy method from Java part. It will called the MainActivity procedure thanks to the handler *h* and invoke method :

```
@Override
public void onDestroy() {
  super.onDestroy();
  try {
      h.invoke(this, GenericActivity.class.getMethod(
               "onDestroy",
               new Class[] {}),
               new Object[] {});
   } catch (Throwable e) {
      throw new RuntimeException(e);
   }
}
```

5.6.3 Graphical toolkit for Android

While developing we wanted to have the most generic Oz application. We first started by implementing logic and graphical parts in Oz. The graphical user

interface (GUI) part was created thanks to a QTk like toolkit. It simplifies the way to create application for Android because the GUI part is just the same as using QTk.

But when we have been able to use our toolkit for GUI in Android, we have noticed that we had a big performance issue. Creating the GUI from Oz part takes about between 10 and 20 seconds. As in this example :

```
W/System.err(  310): Hello World/Activity.onCreate
W/System.err(  310): 'begin : '#8080# 'miliseconds'
D/dalvikvm(  310): GC freed 8599 objects / 494272 bytes in 150ms
D/dalvikvm(  167): GC freed 5510 objects / 321824 bytes in 225ms
D/dalvikvm(  310): GC freed 6295 objects / 499288 bytes in 149ms
W/System.err(  310): 'Creation time for canvas+window : '#11.57#
' seconds'
```

As this part is only executed once, it was still possible to create application in Oz part but the main concern appeared when drawing objects on the canvas :

```
W/System.err(  310): going through onDraw
W/System.err(  310): 'begin : '#75870# 'miliseconds'
W/System.err(  310): 'Creation time for both rectangles : '#0.81
W/System.err(  310): 'Time for setARGB : '#0.81
W/System.err(  310): 'Time for drawing the object : '#0.17
W/System.err(  310): 'begin : '#79650# 'miliseconds'
W/System.err(  310): 'Creation time for both rectangles : '#0.81
W/System.err(  310): 'Time for setARGB : '#0.82
W/System.err(  310): 'Time for drawing the object : '#0.17
W/System.err(  310): 'Time for the complete draw : '#3.95
```

The time for creating an object is about two seconds and it growth linearly with the number of objects. In the example above, it took about four seconds to complete the drawing part. The code used for this benchmark is :

```
local Begin = {Property.get time}.total in
{System.show 'begin : '#Begin#'miliseconds'}
RectF1 = {RectFC.new i(float({IntToFloat (Object.xb+1)})
                       float({IntToFloat (Object.yb+1)})
                       float({IntToFloat (Object.xe-1)})
                       float({IntToFloat (Object.ye-1)}))}
RectF2 = {RectFC.new i(float({IntToFloat Object.xb})
                       float({IntToFloat Object.yb})
```

```
                        float({IntToFloat Object.xe})
                        float({IntToFloat Object.ye}))}
{System.show 'Creation time for both rectangles : '#
             {Float.'/' {IntToFloat {Property.get time}.total-Begin} 1000.0}}
end

local Begin = {Property.get time}.total in
   {P1.p setARGB(int(255) int(Red1) int(Green1) int(Blue1))}
   {P2.p setARGB(int(255) int(Red2) int(Green2) int(Blue2))}
   {System.show 'Time for setARGB : '#
                {Float.'/' {IntToFloat {Property.get time}.total-Begin} 1000.0}}
end

local Begin = {Property.get time}.total in
   case Object.form
      of rect then {Canvas.p drawRect(RectF2 P2)}{Canvas.p drawRect(RectF1 P1)}
      [] oval then {Canvas.p drawOval(RectF2 P2)}{Canvas.p drawOval(RectF1 P1)}
      [] text then {Canvas.p drawText(string(Object.text)
                              float({IntToFloat Object.xb})
                              float({IntToFloat Object.yb}) P1)}
      else {{{J.c 'java.lang.System'}.get err}.p
            println(string("error unkown object to draw"))}
   end
   {System.show 'Time for the complete draw : '#
                {Float.'/' {IntToFloat {Property.get time}.total-Begin} 1000.0}}
end
```

What we can understand from this code is that there is too much communication between Mozart and Java leading to multiple environment switches. In the first part, the creation of the rectangles is realized in six steps. The conversion of floats from Oz to Java in order to create the instance of Rectangle, the creation of the instance and the transport from Java to Oz of the instance. These three steps are repeated twice. It takes almost a second to compute this part. We experience the same duration for the *setARGB* method.

The performance issue would only be a minor problem if we were using most of the graphical libraries provided by the platforms. But with the Java API provided for Android, we have to face with another kind of library. The way the GUI works in Android is by refreshing the screen regularly. But communication between Oz and Java parts happens while the screen is refreshed. Thus, the drawing part has to be processed a lot of time. Which

means that the refresh time also grows linearly with the number of objects to draw.

5.6.4 Application structure

In the Oz part, the following network messages may come from the ring :

- (fingerChanged(... ... unit)#...)
- (join(≤ N ≥)#...)
- (leader#...)
- ('lock'(...)#...)
- (msg(locked(,,,))#...)
- (msg(update(,,,))#...)
- (onRing(...)#...)
- ('prepare'(id:... item:item(,,,) tid:...)#...)
- (prepared(key:... rkey:... tid:...)#...)
- (predSetChanged(,,,—,,, nil)#...)
- (rangeChanged(... ...)#...)
- (registerRTM(... tid:...)#...)
- (succChanged(... ...)#...)
- (succListChanged(,,,—,,, ,,,—,,,)#...)
- (update(id:... item:item(,,,) tid:...)#...)
- (voteAck(key:... rkey:... rtm:... tid:... vote:prepared)#...)

But we are only interested in message corresponding to the pattern msg(M). We only have these messages :

- (msg(locked(dt))#...)
- (msg(update(id:... item:item(,,,) tid:...))#...)

The message that is interesting for the state of the application is *update* which can be for an object or for the list of object. If the id is *dt*, it means that an object has been created or deleted. When the id is a number, an object has been modified. In both cases, we need to update the status of the application.

The logic part need to interact with the graphical part when update happens on the system. This communication works using the reflection mechanism. When Java gets the canvas to draw something on the screen, it propagates the event through Oz. The following methods are provided for this communication:

- draw(...) : object to draw (creation or modification)
- unDraw(...) : object that has to be removed

The first method is the reaction to the messages *update(id:dt ....)* and *update(id:... ...)*. If *dt* is updated, a new object has to be drawn. The logical part will find the new objects and then ask the GUI part to draw this new object.

If an object is modified, we got the second messages and the application draw the same object but with different values. The GUI part will erase the old information for the object with the new ones.

If an object is removed, we need to notify the GUI part with the *unDraw* method. The GUI stores the state of objects that are currently drawn. When the method *unDraw* is called, the GUI will remove this object from the Java store.

For the GUI part, we need to notify the logic part when new objects are drawn from the device, when objects are moved and when objects are removed. The two last operations need the selection of objects, the logic part have to be notified in order to lock the selected object. The following methods are provided for this communication:

- onConnect(String ticket) : when the user asks to connect to a ticket
- onSelect(Int id, ...) : reaction to the selection of an object
- onDraw(...) : refresh drawing on the screen
- newObject(...) : reaction to the creation of an object
- updateObject(...) : reaction to the modification of an object

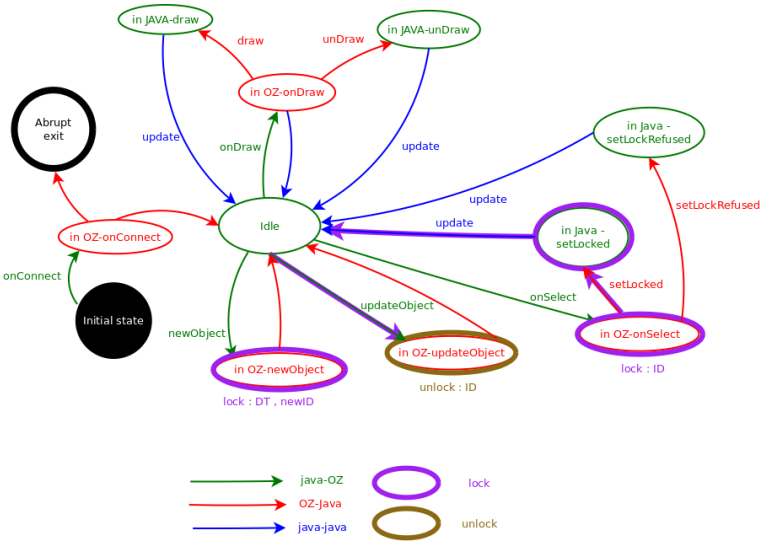


Figure 5.8: DeTransDrawid state diagram

These methods propagate from Java to Oz the information that the application state have been modified.

Figure 5.8 shows the state diagram of DeTransDrawId.

In the initial state, the application starts. The user needs to press menu button to access to functionalities of the application. The connect button will help to call onConnect method. It means that the application will try to connect to the ring thanks to a ticket that the user should enter in the textfield. If the connection is established, the application goes to idle state. In case where something wrong happened, the application exit abruptly.

The idle state is in Java and change state only if something happens there. Most of the time, the current state switches from *idle* to *in Oz-on draw*. This cycle allows the logic part to notify the Java part if objects have been created or removed. As we know that Android is constantly refreshing the screen, we know that the application will be sufficiently responsive to give visual feedback to application updates.

When interaction happens from Android user, there are other method calls. If the user draws an object, the *newObject* will propagate from Java to Oz in order to logically create this new item. Then, the application simply returns to Java. During this method, DT and a new id are locked to commit the creation of the object. The lock is released as soon as the change is commited.

The most interesting part of the application is in the last methods : *onSelect* and *updateObject*. When the user want to modify the position of an object, he needs to change from *drawing* mode to *select* mode. This is possible by going through the menu and push the select button. A click on an object will try to select it. While waiting for result about the lock, red dots will appear on the object. The application is now in *in Oz-onSelect* state waiting for the lock. If the lock is accepted, the application stores the lock and returns to Java part with *setLocked* method. Otherwise, the application cancels changes and returns to Java with *setLockRefused* method.

The lock begins in *in Oz-onSelect* state and is still active in *in Java-setLocked*. When Java parts has been notified about the lock, the application returns to *idle* state keeping the lock. Now the object appears with black dots as visual feedback that the lock has been accepted. The lock is released after the application is ready to commit the changes (when the user has clicked outside the object).

To release the lock, the first step is to propagate *updateObject* from Java to Oz. The logic part will then commit the changes, release the lock and then returns to Java part in *idle* state.

5.6.5 Screenshots of the applications

Figure 5.9 shows the application running on a desktop while Figure 5.10 shows it running on the Android platform. For evident reasons of readability, the picture is a screenshot of the application running in an emulator rather than a photograph of it running on the actual device.

5.6.6 Locking mechanism

In DeTransDrawid, the application also works over transactions. As the GUI part is separated from the logic part, transacations are implemented in Oz reusing the work done for DeTransDraw.

The first use is for loading the system when joining a Beernet ring. In procedure *LoadDT*, we only need to read data. We first get the list stored with key *DT*. Then, we get the object corresponding to each value in the list.

Here is the method :

```
%%% Post        : Load the work already done if it exists
LoadDT =
   proc{$}
      S
      ProcS = proc {$ Li}
```

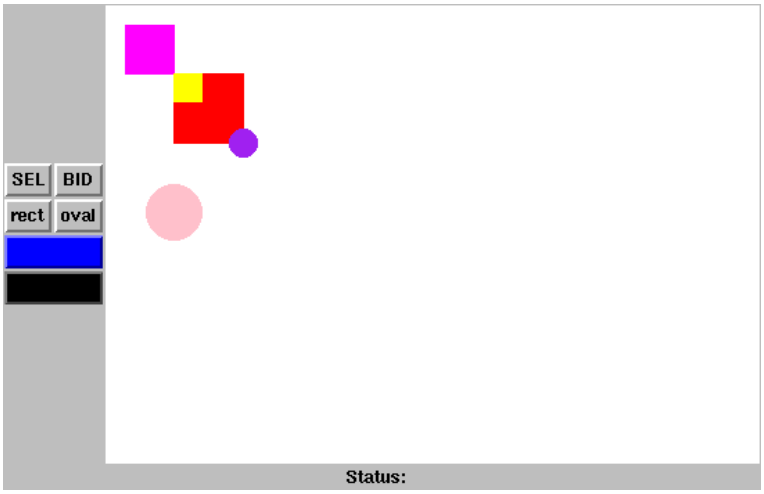


Figure 5.9: DeTransDraw running on the desktop

Figure 5.10: DeTransDraw running on Android

```
                  case Li of nil then skip
                  [] L|R then M
                  in
                     M = {GetVal L}
                     {DrawObject M}
                     {ProcS R}
                  else {System.show 'bug'}
                  end
               end
   in
      {Conn.exTrans Trans _}
      S = {GetVal DT}
      {ProcS S}
      LastDT:=S
   end
```

The DHT is accessed with the procedure *GetVal*, with the code :

```
%%% Pre                 : Key is the key of the value to get
%%% Post         : Val will store the value corresponding to Key

GetVal =
   proc{$ Key Val}
      Trans = proc {$ Obj}
                  {Obj read(Key Val)}
               end
      in
      try
         {Conn.exTrans Trans _}
         catch _ then Val = error
         end
   end
```

When we create an object, we need to access the DHT to add the object and the id the list. The method in DeTransDraw is :

```
%%% Pre          : Id is the key of the new object
%%%                Val is the value of the new object
%%% Post         : The new object is added in the DHT
AddItem =
   proc{$ Id Val}
```

```
      DTval
      KeyRing   % lock on DT
   in
      DTval = {GetVal DT}
      {Conn.locks [DT Id] KeyRing}
      if KeyRing == error then
         thread
            {Delay 500}
            {System.show 'retrying'}
            {AddItem Id Val}
         end
      else
         LastDT := {Append DTval [Id]}
         {Conn.commit KeyRing [DT#@LastDT Id#Val]} %%% Commit new value
      end
      {Conn.reader Id}
   end
```

The method asks the lock for keys *DT* and *Id* to get the list and create the new key. The *KeyRing* is the key to commit new values for DT and Id.

The method ends with {*Conn.reader Id*} statement which means that the application is now a reader of Id key in the DHT. Everytime this key is locked or updated, the application receives *msg(locked(...))* and *msg(update(..))* messages.

If the lock is refused, we can retry to ask for the lock until we are able to add the item. Indeed, this operation does not modify any existing object. We are only adding a new value to the list and a new key and value in the DHT.

When objects are selected and updated, we also need to lock them. They are first locked when there are selected and when they are unselected, the lock is released. Here is the method to release a selected object :

```
%%% Pre         : Id is the key of the object to release
%%% Post        : Release the object with the key Id and commit the changes
ReleaseObject =
   proc{$ Id}
      Val#_ = {Dictionary.get VDict Id}
      Key = {Dictionary.get KeyDict Id}
   in
      {Conn.commit Key [Id#Val]}
      Selected := {List.subtract @Selected Id}
   end
```

5.7 Conclusion and future work

In its current incarnation, the DeTransDraw application is little more than a proof-of-concept. However, most of the major technical problems are now solved. What remains to be done is a better integration of the application in its environment, particularly on Android. Some features such as removing object, multi-selection and changing size still needs to be implemented. The application also needs some end-user polishing, e.g., an application icon and availability on the Android market.

DeTransDraw and DeTransDrawid already show that an interactive application, running fully distributed on small embedded platforms can be made efficient, fast and reliable thanks to the principles outlined in the SELFMAN project.

5.8 Papers and publications

Decentralized transactional collaborative drawing

Jérémie Melchior, Boris Mejías, Yves Jaradin, Peter Van Roy, Jean Vanderdonckt. Submitted to COPS'09 (see Appendix A.12).

This paper proposes a decentralized architecture based on a peer-to-peer network providing decentralized transactional support with replicated storage. As a consequence, there is a gain in fault-tolerance and the transactional protocol eliminates the problem of network delay improving usability and network transparency. The same technique can be used for collaborative text editing and other collaborative tasks.

Decentralized transactional collaborative drawing - Demo

Boris Mejías, Jérémie Melchior and Yves Jaradin. Demonstrator at Collaboration Meeting for FP6 and FP7 projects. (see Appendix A.13).

This is a description of the demonstration we gave of DeTransDraw in the Internet of Services 2009 Collaboration Meeting for FP6 and FP7 projects.

Chapter 6

D5.10: Design and analysis of Beernet, the Mozart structured overlay network implementation

6.1 Executive summary

This deliverable presents the analysis of Beernet [15], the structured overlay network developed using the Mozart programming system. It is the successor of P2PS [21], presented in deliverable D1.5, in WP1, during year 2 of the SELFMAN project. Beernet implements the Relaxed-Ring [17] network topology presented as result of year 1, and it includes the high level layer for transactional DHT, which is the result of WP3, providing consistent symmetric replication.

The deliverable is presented as the draft of Boris Mejías's Ph.D. dissertation, and it is included in Appendix A.14. The dissertation makes an extensive review of existing structured overlay networks. It explains the contribution of the Relaxed-Ring, making not only an experimental evaluation of the algorithm, but also an analysis of it using feedback loops, which are part of the results of WP2. It also details the algorithms used for atomic transaction commits, contributing with an eager protocol for synchronous collaborative applications. It presents a set of applications built on top of Beernet to show the impact of it, and to emphasise its contribution.

6.2 Contractors contributing to the Deliverable

This deliverable is in the form of a dissertation written by a researcher of **UCL(P1)**. UCL is the main developer and author of the papers leading to this dissertation.

6.3 Results

Beernet stands for pbeer-to-pbeer network, where words peer and beer are mixed to emphasise the fact that this is a peer-to-peer network built on top of a *relaxed*-ring topology, considering that beers are usually a mean to achieve relaxation. This deliverable presents the draft of Boris Mejías's Ph.D. dissertation as the main result. The dissertation makes the analysis of Beernet by describing in detail the algorithm of the Relaxed-Ring, which is the network topology on which Beernet is implemented. The Relaxed-Ring is one of the results of WP1, beign part of the first two years of the project. The Relaxed-Ring is compared to other structured overlay networks by making a summary of the state-of-the-art in peer-to-peer networks. The main contribution of the Relaxed-Ring is that introduces non-transitive connectivity in the design of the protocols that provides self-organization of the ring. Beernet also includes the results on transactional DHT from WP3, by implementing the Paxos consensus algorithm, and also by developing the Eager Paxos protocol that is more suitable for synchronous collaborative applications. Four applications are also presented on this dissertation emphasizing the impact of the contribution of Beernet. The applications are Sindaca, a community-driven recommendation system described in Deliverable D5.3; DeTransDraw, a collaborative drawing tool, presented in detail in deliverables D5.8 and D5.9 5; and two applications designed and implemented by third parties, beign a decentralized wiki, and a decentralized version of Twiteer.

The contributions of this deliverable can be listed in detail as follows:

- The design of a protocol for self-organizing peer-to-peer networks creating a network topology called relaxed-ring. The network is able to deal with false suspicions in failure detection and with non-transitive networks such as the Internet, improving lookup consistency with respect to existing peer-to-peer networks. The relaxed-ring also provides self healing by triggering a failure recovery mechanism when the crash of a peer is detected.

- The relaxed-ring protocol is cost-efficient because it does not rely on periodic stabilization to repair the network when it is affected by churn. The relaxation introduces branches to the ring topology, but it keeps the routing algorithm competitive with $log(N)$ hops to reach any peer.

- We provide a self-adaptable routing topology that allows the relaxed-ring to take advantage of full connectivity in small networks, and logarithmic routing in large networks. The system can scale up and down

making it suitable for many different applications independent of the size of the network.

- We present the algorithms of the relaxed-ring using feedback loops to analyse and validate its self-management properties. The feedback loops help us to understand how the system monitors itself, analyses the information, and triggers the needed action to modify the system.

- We study and validate the Paxos consensus algorithm for atomic transactions on a replicated DHT, and we compare it with the well known solution for distributed transactions called Two-phase commit.

- We adapt Paxos consensus algorithm to provide eager locking of the transaction participants, and we extend it with a notification layer to make other peers aware of the modifications. This new protocol allows us to design application where users can collaborate synchronously.

- As proof-of-concepts, we have implemented Beernet, the pbeer-to-pbeer network, a relaxed way of doing peer-to-peer. It is an implementation of the relaxed-ring where peers are organized as a set of distributed-transparent actors. These actors represents components with encapsulate state and that communicates only via message passing, avoiding share state concurrency. Beernet also takes advantage of the fault-stream model for failure handling improving its modularity and network transparency. These characteristics provide a better programming framework for self configuration of components.

- We have implemented and presented to the research community three different demonstrators to introduce the concepts of the relaxed-ring, atomic transactional DHT, and synchronous collaboration with eager transactions.

- We develop two applications on top of Beernet to exploit optimistic and pessimistic transactions, and the notification layer. These application provide a community-driven recommendation system, and a collaborative drawing tool. Two other applications designed and developed by third parties are also presented so as to emphasize the impact of the contribution of the relaxed-ring and its transactional layer.

The dissertation is organized as follows. After the introduction, there is a review of all three generations of peer-to-peer systems, being structutured overlay networks the most important focus of the analysis. The systems we reviewed are not only studied from the point of view of their overlay graph,

but also from their self-managing properties. We also review distributed storage and the connection of peer-to-peer with Grid and Cloud Computing. The following chapter presents the protocols and algorithms of the Relaxed-Ring, being an important part of the contribution of this dissertation. The Relaxed-Ring is also studied using feedback-loops so as to understand its self-managing properties from a architectural and software design point of view. Note that feedback-loops are part of the results in WP2. Evaluation of the Relaxed-Ring, specially in comparison with other overlay graphs, is done experimentally using a concurrent multi-agent simulator.

Once we have presented the Relaxed-Ring, the dissertation continues with the study of distributed storage. We analyse Two-Phase commit, Paxos consensus algorithm, and we describe our contribution with Eager Paxos and the notification layer. Then, the dissertation describes the design decisions and implementation details of Beernet, which implements the Relaxed-Ring and its layer for transactional distributed hash tables using symmetric replication. Before the concluding the dissertation, we present a set of applications designed and developed using Beernet and the ideas of the Relaxed-Ring. Some of the applications are developed by the authors, and some of them are contributions of third parties, emphasizing the impact of this dissertation.

Apart from the contributions presented in the dissertation, there are other results included on this deliverable. We have presented the paper "Beernet: RMI-free peer-to-peer networks" [16] in the Workshop on Distributed Objects for the 21st Century (DO21) at ECOOP'09. This paper presents the architecture and programming concepts used in the implementation of Beernet. We have also published the paper "From mini-clouds to Cloud Computing" [18] at the SELFMAN SASO Workshop 2009. This paper is a proposal for future work of this deliverable.

The author has won the *"Best Presentation Award"* in the Doctoral Symposium of the "XtreemOS Summer School", held at the Wadham College of the University of Oxford, Oxford, UK, on September 10, 2009. The presentation was entitled "Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT" [14], and it summarized the contribution of the dissertation.

6.4 Dissertation, Publications and Award

This section is dedicated to give a brief introduction to the documents included as appendices on this deliverable. They correspond to a Ph.D. disseration, a workshop paper, and the abstract of a presentation that won an award in a Doctoral Symposium.

Beernet: A Relaxed-Ring for Self-Managing Decentralized Systems with Transactional Replicated Storage

The core of this deliverable corresponds to this Ph.D. disseration. The full version of the draft is to be found in Appendix A.14. Its content and contribution have been already introduced in Section 6.3 and in the Excutive Summary of this deliverable.

Beernet: RMI-free peer-to-peer networks

This paper is included in the proceedings of the Workshop on Distributed Objects for the 21st Century (DO21) at ECOOP'09. The paper describes the architecture of Beernet, and discusses language abstractions that are useful in distributed object, mainly for development of peer-to-peer system. The position is that RMI is considered harmful. The paper is included in Appendix A.15.

From mini-clouds to Cloud Computing

This paper has been accepted and presented in the Workshop on Architectures and Languages for Self-Managing Distributed Systems, SELFMAN at SASO09. The paper describes a proposal to apply the results of this dissertation in Cloud Computing. The paper is included in Appendix A.16.

Best Presentation Award: "Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT"

Boris Mejías, author of the Ph.D. dissertation, presented the main contribution of his work in the Doctoral Symposium of the "XtreemOS Summer School", held at the Wadham College of the University of Oxford, Oxford, UK, on September 10, 2009. He won the "Best Presentation Award". The abstract of the presentation, together with a copy of the certificate are included in Appendix A.17.

Chapter 7

D5.11: Evaluation of security mechanisms

7.1 Executive summary

Deliverable 5.11 reports on a self-protection mechanism for Wikipedia to identify spam and reduce the effect of spam (Section 7.3). The spam detection system builds on the general self-protection infrastructure for Wikipedia developed in Deliverable 5.6 where it was used to enhance the credibility of articles and edits in Wikipedia.

The particular spam problem we address is to identify the users who are spammers in Wikipedia. Often these are the anonymous contributors but nevertheless we can still identify them by requiring that they need to have an identity in a social network through the use of a trusted proxy in the infrastructure. The most difficult kind of attack to defend against is where the user creates multiple virtual identities in the social network and employs a large number of identities to make it easier to distribute the spam contribution among the virtual users (the sybil users). This stategy tries to make difficult to distinguish between an honest user from a spammer.[1] We map the sybil spam identification problem to one where edits correspond to votes and we want to be able to measure the votes given that there are sybil voters. The approach used is to measure the maximum flow in the graph which can be used to identify the bottlenecks which are typically the connections from the sybil region to the honest region of the graph. Initial experiments using data collected from Facebook show that the effect of spam can be effectively limited to the actual number of honest users the spammer

[1] We do not deal with approaches which try to understand the underlying semantics of the edit which would lead to difficult natural language understanding problems.

is connected to in the social network. The results show that an automatic and transparent mechanism for identifying spammers which is one of the challenges to the growth of Wikipedia is feasible.

We also report on an extension to Deliverable 5.6 in Section 7.4 which was not reported earlier as we had only stumbled upon the phenomena at the time of the report. Only certain small world networks are known to be navigable, meaning have routing algorithms which have short routes. Such small world network models however do not resemble actual social networks as their graph structure is much more sparse and less clustered. Conversely, models such as the Watts and Strogatz small world networks which can produce more clustered networks and have statistics closer to real world networks are believed to be not navigable using greedy routing algorithms. We discovered by enhancing local routing with local neighbourhood topology, Watts and Strogatz networks become much more navigable and the routing length approaches known navigable networks like the Kleinberg small world networks. We believe this may be applicable to real world networks which are more difficult to route than artificial small world networks.

7.2 Contractors contributing to the Deliverable

NUS(P7) contributed to this deliverable.

NUS(P7) enhanced the self-protection infrastructure for Wikipedia in Deliverable D5.6 to deal with the problem of spam.

7.3 Applications of the Wiki Credibility Infrastructure

7.3.1 Reducing Wiki spam with Social Networks

In Deliverable D5.6, we introduced a mechanism to enhance Wikipedia using existing and reputable third party services. The philosophy of Wikipedia is to allow anybody to contribute which includes anonymous users.[2] It turns out that high quality contributions come from the vast number of anonymous users who may only contribute once [2]. Such high quality (anonymous) authors may be experts or have established reputation elsewhere. Although their edits may be of high quality, it is desirable that such edits can be associated with a credibility measure which is indicative of the quality particularly when it is anonymous. This can be achieved using our Wikipedia enhancement [7] in Deliverable D5.6 to transfer the reputation from a third party to Wikipedia in a possibly anonymous fashion. In Deliverable D5.11, we will introduce another application of our Wikipedia enhancement framework which is targeted at the problem of how to reduce spam in Wikipedia.

A recent study found that Wikipedia growth is slowing down [26]. On the other hand the amount of spam or vandalism attempts continues unabated. An important trend identified in [26] is a new positive trend which shows that Wikipedia is losing editors. This is very significant since it is precisely the work of human editors which is the main spam and vandalism mechanism in Wikipedia. Although the use of human editors cannot be regarded as scalable, in practice, it was effective simply because the number of editors could keep pace with the spammers or vandals. Now, however, we may have come to a point when the existing Wikipedia development will gradually be threatened by the scale of the spam.

As such, this may be a good time to reevaluate how Wikipedia should work in the future — perhaps by accepting less edits and accepting only from reputable authors. The approach in Deliverable D5.6 is to make use of third party services if they exist to transfer credibility information into Wikipedia [7]. However, this does not deal with users who do not have any credibility or reputation, in particular, virtually created identities. In order to distinguish virtual identities, we leverage the power of social networks and the availability of extensive social networking sites such as Facebook, MySpace, etc. We focus on the problem of spammers which try to create fake identities of editors to take over an article. Fake identities are used to hide their activities and create an artificial community of editors.

[2] Anonymous users are users who do not register and are listed by their IP address.

In today's increasingly connected world, a user in Wikipedia is likely to be connected to other users in one of the social network. The social network can be viewed as a big graph which shows relationships between users in the network, i.e. the friends of a user. An honest user is expected to correspond to a single node in the graph. Spammers which try to exploit multiple identities can be analyzed to see how their relationships differ from honest users. Although spammer can create new virtual identities, the corresponding nodes in the graph are limited in what friends they can know.

One of the largest social networking sites is Facebook with more than 300 million active users [5]. Furthermore, much of the graph is publically visible. In Deliverable D4.4b, we developed a social network crawler for extracting the graph from sites like Facebook. We build upon those tools here for the Wikipedia spam prevention infrastructure.

If we use Facebook as the third party service for our Wikipedia enhancement, we will be able to use it to limit the number of probable spammers as well as detecting the presence of spammers from a certain period of time thus can be used as a tool to fight spams.

As mentioned above, given the slow growth of Wikipedia and ever high number of spams it may be a good time to moderate the edits. We can use the Wikipedia enhancement to link every edits in Wikipedia to a user in social networking site (e.g. Facebook) via trusted proxy. This requires any (anonymous) users to use their Facebook account to make an edit in Wikipedia. However, to maintain the anonymity, the trusted proxy can be setup to anonymize the account but still be able to map back the edit to the particular user whenever requested. Therefore, users can remain anonymous and their edits can still be redarded as qualtiy edit if they linked the edits with reputable third parties.

The linkage of Wikipedia edits to a user in a social network gives more information to Wikipedia admins to fight for spammers. A typical spammers can no longer do edit freely due to the linkage to the social networking account. If the spammers do the spam using the same social networking account, it will be easy for the Wikipedia admin to revert back all his edits. On the other hands, and attempt to use different social networking account will be easily captured by analyzing the social network graph.

Originally, we had intended to explore various Sybil defence approaches in the literature in the Wikipedia context. However, recently, the SumUp approach [27] has been shown to be very promising and more effective than existing approaches Thus, we decided to base our Wikipedia spam mechanism on vote collection using SumUp.

Analyzing the social networking graph to determine the spammers requires substantial compute effort which may not be feasible for real-time

editing response given the load of Wikipedia. We propose to collect all edits during a specific period of time. Then Wikipedia contacts the trusted proxy/proxies using our credibility infrastructure to map the edits to users in the social network graph. The problem of analyzing the social network structure is reduced to a problem of Sybil Voting. We treat the users who do the edit as voters and select some trusted users (in the social network) as the vote collectors. The number of accepted Sybil votes can then be limited using SumUp algorithm [27]. The maximum-flow from each voters to the vote collectors is calculated. Once the flow is saturated, the Sybil region becomes disconnected from the honest region as there is no more flow from the Sybil to honest region. At this time, those voters that cannot be collected are likely to be the Sybil voters who may reside in (several) "disconnected from honest" regions. By using this assumption, the remainder of the sybil nodes can be found by graph connectivity tests on the Sybil voters. One limitation of our approach is that it only deals with edits from virtual nodes, the sybil users.

In summary, our Wikipedia enhancement [7] has several potential applications that can improve the trust of content and edits in Wikipedia by associating the edits with existing third party services to transfer the reputation as well as a potential tool for the Wikipedia administrators to fight spams by associating the edits with social networking sites such as Facebook. The association between edits and a user in a social network can be used to analyze potential spam attacks as well as to rate-limit the number of edits made by the Sybil attacker to the number of attack edges (friends) the attacker has in the social network graph. One advantage of this approach is that it can be integrated easily into Wikipedia without much effort and provides transparent and automatic self protection mechanisms for Wikipedia.

7.3.2 Experiments

We evaluated the effectiveness of our proposed Wikipedia self-protection mechanism by testing it with real social network data from Facebook. First, we crawled a subset of Facebook graph, using an approach similar to [22]. The subgraph extracted from Facebook consists of 73719 nodes and 5992544 edges. The average node degree is 81.289 and average clustering coefficient is 0.364. It is not possible to visualize the whole graph here, so we show the first 1000 collected nodes in Figure 7.1.

SumUp [27] is a resilient vote aggregation system that leverages the trust network among users to defend against Sybil attacks. Sybil attacks is an attack where adversaries creates many identities in the trust network to outvote the honest users. Defending against non Sybil attack is easy if there

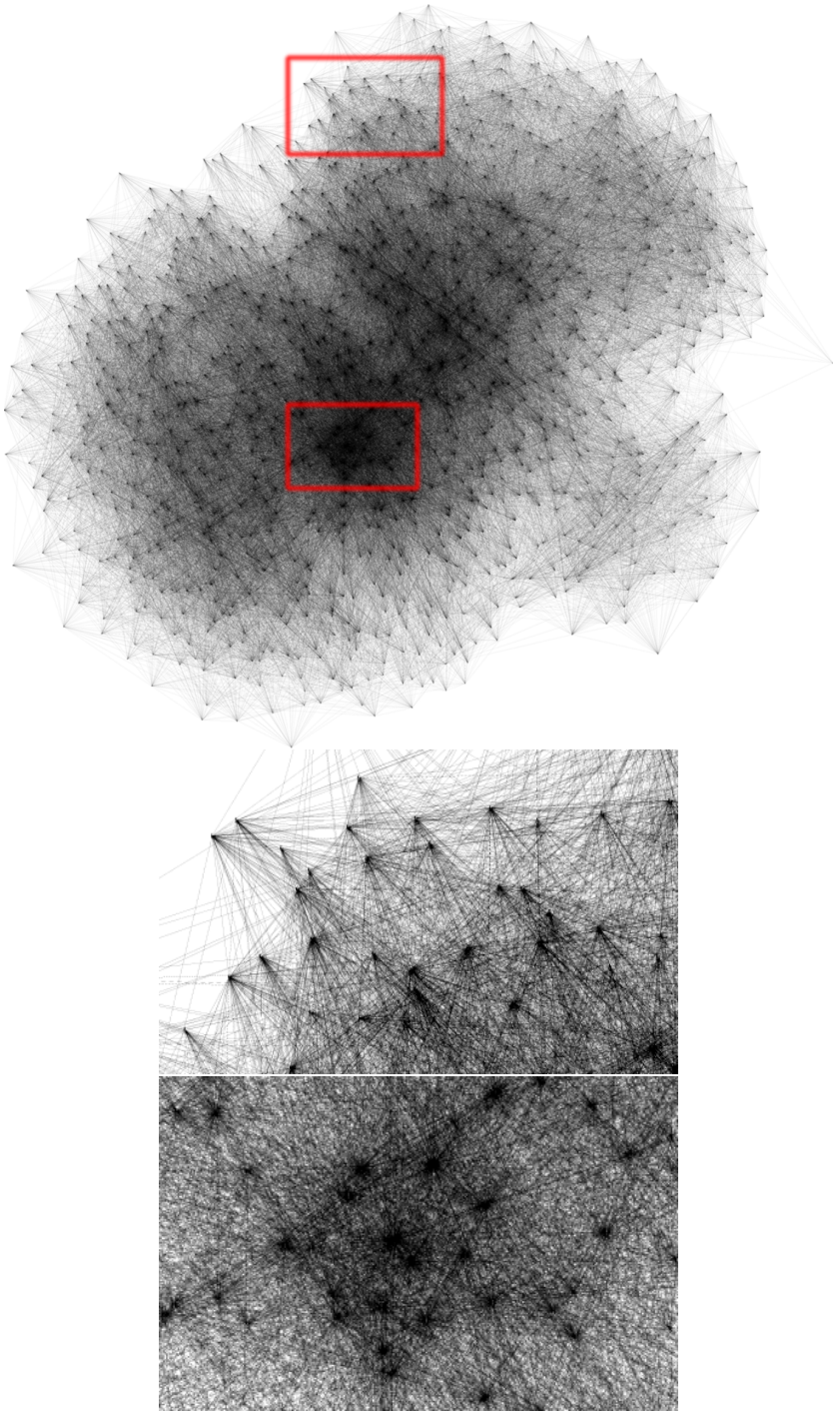


Figure 7.1: Facebook subgraph with 1000 nodes visualized. The two pictures at the bottom are the zoomed-in version of the red regions of the graph.

is only one attacker using the same identity. The hardest defense is against many different (independent) real attackers.

SumUp works by assuming that a single user in the trust network can only have limited number of friends, typically hundreds. To limit the number of votes from Sybil users, SumUp select a trusted user as the vote collector in the trust network and run a maximum-flow algorithm from the voters to the vote collector. This way, the number of Sybil votes can be limited to the number of friends (called the *number of attack edges*) that the Sybil node has. SumUp further reduces the number of attack edges by employing link pruning and reducing the capacity along the path from the voter to vote collector via negative feedback.

We reduce the Wiki spam prevention problem into a voting problem employing SumUp. All edits in Wikipedia from a certain period of time are mapped to user/nodes in the social network. The users in the social network are the voters. We then pick a trusted node in the social network as the vote collector for the SumUp algorithm. We can also manually inspect the edits and use negative feedback as well as link pruning. The voting result will tell us which edits in Wikipedia are likely to be coming from the same Sybil attacker.

This experiment uses the collected graph from Facebook with $N = 74$K nodes. We assume that the edits are already mapped to users in the Facebook graphs. The edit model is that the users are randomly selected from the graph. Thus, some are honest users while others are Sybil users. We then measure how effective is SumUp in identifying the Sybil users that did the edits. We will then also identify the rest of the Sybil users linked from the rejected Sybil user since they are in the same region but didn't vote. This further captures all other Sybil nodes in the graph that haven't been used by the Sybil user. We purposedly created several Sybil regions in the Facebook graph so that we are able to measure how the effectiveness against Sybil attacks.

Figure 7.2 shows the number of accepted Wikipedia edits (Sybil votes) as the number of Sybil nodes is varied in the social network. We assume that we have a single attacker that has 200 friends (this is substantially higher than the average node degree). There is a single vote collector that can receive up to 700 edits/votes by distributing the 700 tickets as in [27]. The experiment shows that initially the number of edits accepted from sybil nodes is proportional to the number of sybil nodes in the trust network as the number of nodes increases. However, after 200 sybil nodes, the number of Sybil votes is limited to 200 which is the number of attack edges. What this means is that it is feasible to limit the number of Wikipedia edits to the maximum number of attack edges of the attacker.

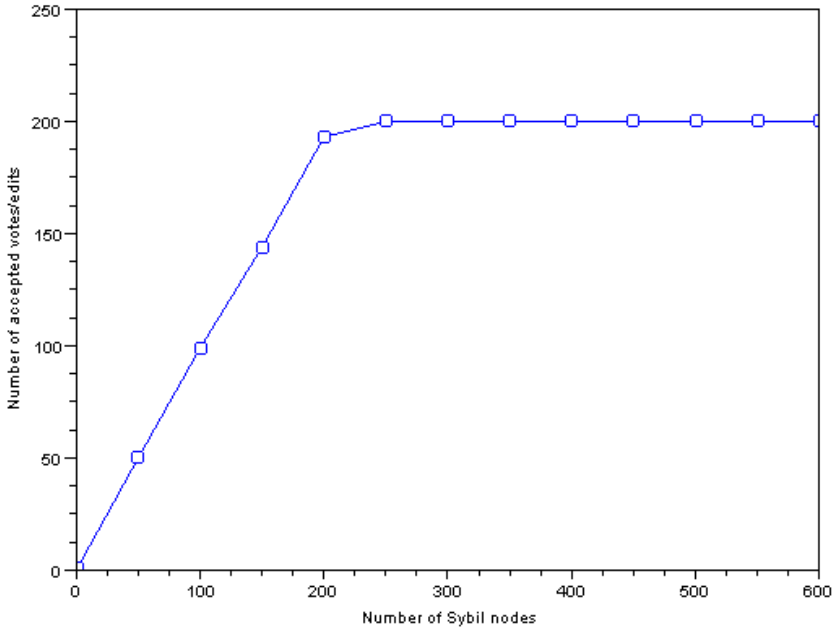


Figure 7.2: Varying the number of Sybil nodes

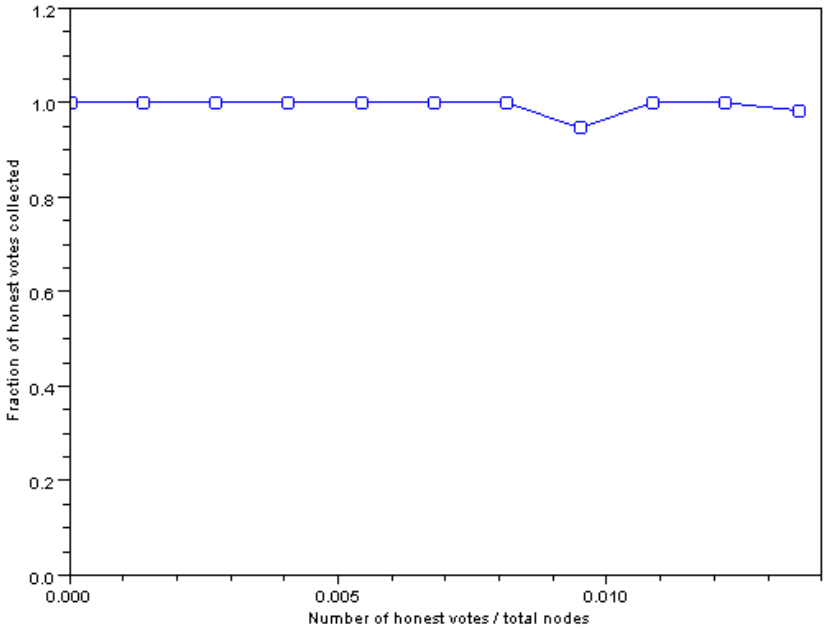


Figure 7.3: Acceptance rate when there is no attack

The next experiment in Figure 7.3 shows the fraction of the honest edits accepted when there is no attack. What we would like is that all honest edits are acceepted when there are no attacks, i.e. the edits only come from honest nodes. We can see that the acceptance rate is around 100%, thus, the protection mechanism doesn't affect the honest users.

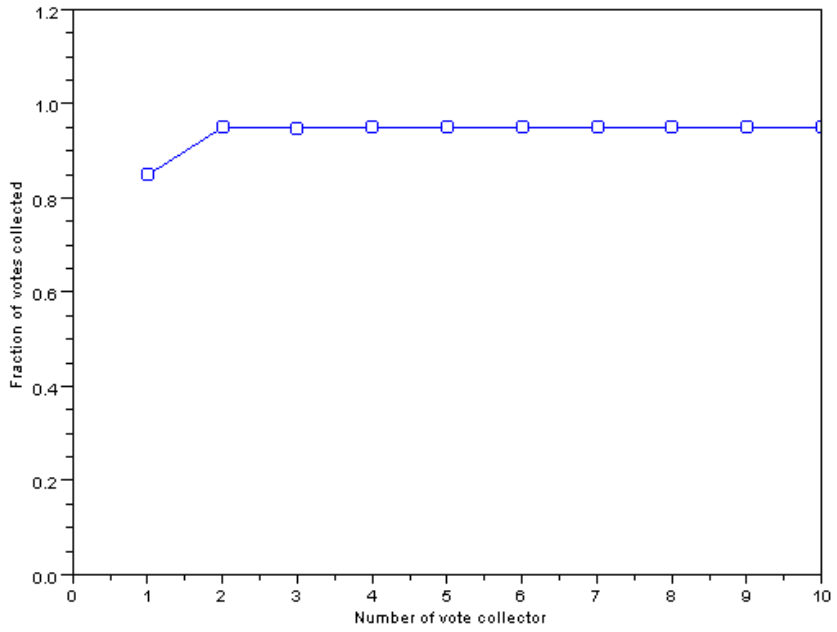


Figure 7.4: Varying the number of vote collectors

Vote collectors plays a crucial role in this framework. Having more vote collectors can give rise to more robustness. Figure 7.4 shows an experiment to explore the importance of having robust vote collectors. In this experiment, we create 250 sybil nodes with 200 attack edges from the sybil region to the honest region. A single vote collector may suffer from the problem that it may lose some edits/votes because of overcapacity on some edge. The experiment shows that having more vote collectors reduces this problem as the fraction of votes collected becomes stable after two vote collectors. Only about 95% of the edits are allowed because the sybil region is limited to about 200 edits.

To summarize, the experiments show that it is feasible to limit the effect of spam edits to the number of attack edges of the sybil nodes. Once the sybil region is identified, an adminstrator could remove all the spam including the spam which was not rejected since SumUp allows some spam to get through.

7.4 Navigability in the Watts and Strogatz Small World Model

We report on an extension of the work in Deliverable 5.6. This work is reported in Deliverable 5.11 as it is some new results we discovered around the end of Deliverable 5.6 and thus was not yet ready to go into the Deliverable 5.6 report at the time. The results are also reported in the SELFMAN@SASO09 workshop [6].

The study of small-world networks (SWNs) has become popular with the growth of social networking sites such as Facebook. SWNs were first studied by Stanley Milgram [28] who showed experiments forwarding letters that the length of the chain was between five and six. This is also popularly known as "six degrees of separation". This suggests that SWNs should have a small diameter. Furthermore, SWNs are not just random graphs but they have other properties, most notably, small diameter. A well studied model of a SWN is the one proposed by Watts and Strogatz [29]. Their model (WS SWN) has the virtue of simplicity, while capturing the two properties.

We investigate the problem of finding effective routes between nodes (also known as *navigability*), in WS SWN [6]. The property of small diameter ensures the existence of a short route, but it does not mean that finding a short route is easy, especially in a distributed setting. Ideally, the routing length should be polylogarithmicly bounded and the routing algorithm should not require global information about the whole SWN graph. We revisit the issue of routability for two reasons. Firstly, WS SWN makes it easy to construct SWNs with different amounts of clustering which makes it useful as a model for social networks. Secondly, a number of papers [12] have promoted the idea that WS SWN is not navigable.

Rather than strict greedy routing, we will look at local routing algorithms which are greedy-routing like so as to get better navigability in WS SWN. We employ the NoN-Greedy [19] routing strategy to help reducing the average routing length. In NoN-Greedy routing, each node knows the link information of its neighbors (1-lookahead). With more (but still local) information, the routing can be better guided towards the target and reduces the unnecessary routes to the wrong paths early in the routing thus significantly cut down the routing length.

Preliminary results shows that WS SWN is more navigable than it was suggested. With 1-lookahead, the routing length approaches that of the Kleinberg SWN which is known to be navigable. Increasing the lookahead beyond 1-lookahead appears to give only small gains but substantially increases the storage needed.

7.5 Papers and publications

Wiki credibility enhancement[3]

Felix Halim, Wu Yongzheng and Roland H.C. Yap, Fifth International Symposium on Wikis and Open Collaboration (WikiSym), 2009 (see Appendix A.18).

> Wikipedia has been very successful as an open encyclopedia which can be edited by anybody. However, the anonymous nature of Wikipedia means that readers may have less trust since there is no way of verifying the credibility of the authors or contributors. We propose to transfer external information from outside Wikipedia to Wikipedia pages. These additional information is meant to enhance the credibility of the content. For example, it could be the education level, professional expertise or affiliation of the author. We do this while maintaining anonymity. In this paper, we present the design and architecture of such system together with a prototype.

Routing in the Watts and Strogatz Small World Networks Revisited

Felix Halim, Yongzheng Wu, Roland H.C. Yap, Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2009 (see Appendix A.19).

> Routing in small world networks (SWNs) have mainly been studied in the context of Kleinberg-like SWNs because of their navigability. If one wants to employ real world SWNs such as social networks for self-managing overlay networks, we believe that models like the Watts and Strogatz's SWN (WS-SWN) may be more suitable because the resulting graphs from the WS-SWN construction have properties closer to real world SWNs. Furthermore, WS-SWN provides a parameter p to adjust the clustering coefficient and the diameter of the graph.
>
> The drawback is that WS-SWN is not navigable using greedy routing. We demonstrate some preliminary experiments which suggest that WS-SWN may be more navigable than previously

[3] An earlier draft of this paper appears in the Deliverable D5.6. This paper is the one which is published in WikiSym'09.

thought. Using additional routing information (such as 1-lookahead), routing performance in WS-SWN seems to approach greedy routing performance in the Kleinberg SWN model. This is interesting since it suggests that various graphs such as the WS-SWN may be more usable for routing than previously thought.

Appendix A

Publications

A.1 Software design with interacting feedback structures and its application to large-scale distributed systems

Software design with interacting feedback structures and its application to large-scale distributed systems

Peter Van Roy
Univ. catholique de Louvain
Place Sainte Barbe, 2
B-1348, Louvain-la-Neuve
peter.vanroy@uclouvain.be

Seif Haridi
Royal Institute of Technology
Box 1263
S-164 28 Kista
seif@it.kth.se

Alexander Reinefeld
Zuse Institute Berlin
Takustr. 7
D-14195 Berlin-Dahlem
ar@zib.de

ABSTRACT

As Internet programs become larger and more complex, designing them and predicting their behavior become daunting. In addition to users coming and going and acting concurrently, "abnormal" events such as software errors, partial failures, attacks, and hotspots become normal. To address this problem, we propose that these programs should be designed from the start as a set of interacting feedback structures. Each feedback structure consists of one or more feedback loops and continuously maintains one system property. In a well-designed system, no part should exist outside of a feedback structure. We motivate this approach with examples of robust systems from biology and computing.

To show the power of the approach, we have built the open-source Scalaris transactional store, which combines a structured peer-to-peer network, a replicated storage layer, and a transaction layer. Scalaris consists of six feedback structures working together in a harmonious way. Scalaris scales smoothly and efficiently to hundreds of nodes, handles node and network failures, and performs load balancing. Scalaris uses a modified Paxos uniform consensus algorithm to implement atomic commit. A distributed Wiki built with Scalaris won first prize in the IEEE International Scalable Computing Challenge (SCALE 2008).

In this approach, a system's specification consists of a conjunction of properties, each of which is implemented by one feedback structure. This achieves separation of concerns by defining the concerns in terms of the feedback structures that naturally implement them. We are currently studying how to design with this approach and we are extending the approach to design for a desired global behavior using reversible phase transitions. Such systems will be much easier to design, predict, and manage, and will be less subject to global problems such as multicast storms, chaotic behavior, and cascading failures. They will provide well-defined behavior for a wide range of environmental conditions, even extremely hostile ones.

Keywords

software design, complex systems, distributed systems, feedback loops, self management, transactions, replication, peer-to-peer

1. INTRODUCTION

It is now possible to build Internet applications that are more complex than ever before. The Internet has reached a higher level of availability and scale than ever before using computing nodes that are more powerful than ever before. Experience shows that it is difficult to build applications that take advantage of this complexity: they are hard to design, predict, and manage. They are subject to hostile environmental conditions with frequent node failures and communication problems. They are subject to global problems such as hotspots, attacks, multicast storms, chaotic behavior, and cascading failures [4].

To address these problems, we propose to design applications from the start as a set of feedback structures. Each feedback structure is designed to manage one (global) system property. It consists of a collection of feedback loops, often organized as a hierarchy where each feedback loop may control an inner loop and be controlled by an outer loop. Interaction between feedback structures is limited and well-defined. In a well-designed system, no part exists outside of a feedback structure. Each feedback loop continuously tries to achieve one specific (local) goal by means of an algorithm at its core that is integrated into the system with detection and actuation components. It is important to distinguish between the system level (feedback structures) and the building block level (feedback loops).

We claim that by using design rules and patterns for feedback structures, it is practical to build large-scale systems that are robust, adaptable, easy to understand, and easy to maintain. We motivate the use of feedback structures with examples of real systems taken from both biology and computing. To substantiate the claim, we have built Scalaris as part of the SELFMAN project [28, 34]. Scalaris is a self-managing transactional store built on top of a structured peer-to-peer network. We use a structured peer-to-peer network as the foundation because it already provides scalability and robustness with a design based on feedback and self organization. Scalaris uses an improved variant of the Paxos uniform consensus algorithm at the heart of its transaction manager. Scalaris contains a large number of interacting feedback loops, organized as six feedback structures, that perform self healing to maintain connectivity, do merging of split rings, manage replicas, and implement transactions,

and self tuning to achieve efficient routing and to spread load.

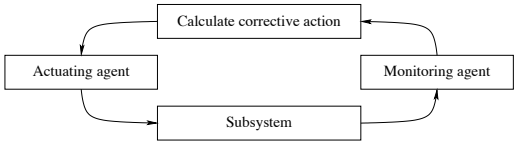


Figure 1: A feedback loop

1.1 Feedback loops and feedback structures

A *feedback loop* in its general form consists of four parts: a monitor, a corrector, and an actuator, attached to a subsystem. We assume without loss of generality that the parts are concurrent components (agents) communicating by asynchronous message passing, as depicted in Figure 1. A part can perform either a global or local action. For example, a global monitor can use gossip-based aggregation to continuously calculate global information and a global actuator can use a broadcast or publish/subscribe mechanism. The corrector contains an abstract model of the subsystem and a goal. The feedback loop runs continuously, monitoring 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 [9]) but there is no need for it to be complete.

A simple example of a feedback loop is a transaction manager. It manages system resources according to a goal, which can be optimistic or pessimistic concurrency control. The monitor accepts lock requests and the actuator gives the response according to the concurrency control algorithm. 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.

A *feedback structure* is a collection of interacting feedback loops that together manage one system property. The feedback loops are typically organized to use both hierarchy and stigmergy, the two basic mechanisms of loop interaction. Through *stigmergy*, loops act on a shared subsystem, and through *hierarchy*, one loop directly controls another.

Very little systematic work exists on how to design with interacting feedback loops. In real systems, however, interacting feedback loops are the norm. But these feedback loops do not interact haphazardly. As far as we can tell, they are always organized as *weakly interacting feedback structures*. We can therefore study feedback structures separately from their interactions. This is why the study of feedback structures is invaluable for designing and understanding real systems. The system specification then consists of a conjunction of system properties, each of which is implemented by one feedback structure. We find that dividing system functionality into feedback structures is a natural way to define and to separate concerns in real systems.

1.2 Interdisciplinary nature

Using feedback loops for system design is an old idea that dates back at least to Norbert Wiener's work on cybernetics [37]. It is being used successfully in many areas both inside and outside of computing:

- *Artificial intelligence.* For example, Brooks' subsumption architecture implements intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [5].

- *Management of computer systems.* This is done at many levels. A simple example is automatic memory management (garbage collection), in which a programmer manages not individual memory blocks but rather the garbage collection policy. Another example is IBM's Autonomic Computing initiative, which reduces management costs by removing humans from low-level management loops [16]. It is used primarily for clusters and databases.

- *Telecommunications.* Armstrong *et al* show how to build reliable telecommunications software in Erlang using the principle of supervisor trees [3]. Each internal node in a supervisor tree corresponds to a feedback loop that monitors part of the system.

- *Control theory.* Hellerstein *et al* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [14]. Hellerstein gives 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.* These algorithms can be formulated as feedback structures. For example, fault-tolerance algorithms use a feedback loop based on a failure detector [12]. The implementation of the failure detector itself requires a feedback loop.

- *Structured overlay networks*, also called structured peer-to-peer networks. They are inspired by previous generations of peer-to-peer networks with random neighbors but provide guaranteed lookup and performance [32]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. Our work in the SELFMAN project is in this area.

- *Social systems* and *biological systems.* Senge *et al* show how to debug problems in human organizations by modeling them as feedback structures [29]. Many natural and biological systems use feedback structures and do self organization [11, 7, 22].

We have taken ideas from many of these disciplines to forge our approach. Some disciplines are needed to design a feedback loop's core algorithm. Others are needed to understand design rules and patterns for interacting feedback loops.

1.3 Structure of the article

This article presents our methodology in a condensed form, supported with many examples.

- Section 2 gives two nontrivial examples of feedback structures, from biology and computing, and derives several design rules from them.

- Section 3 presents the open-source Scalaris transactional storage library, its design, and the Distributed Wikipedia application we have built with it. We also present the Beernet library, which differs in important

ways from Scalaris. We contrast two ways of presenting the Scalaris architecture: a conventional presentation as a layered system and a novel presentation as a set of six interacting feedback structures.

- Section 4 explains how to design Scalaris and similar systems by giving a set of guidelines for the design of one feedback structure and for the decomposition and orchestration of multiple feedback structures.
- Finally, Section 5 recapitulates the approach and points to two important future directions: designing robust systems with reversible phase transitions and justifying and completing the methodology through formal techniques.

2. FEEDBACK STRUCTURES

We study working systems to gain insight in how to construct feedback structures. It is important to understand the basic design rules and patterns before attempting a formal analysis. We give two examples out of many nontrivial systems that consist of multiple interacting feedback loops. Our first example comes from biology: the human respiratory system, which was designed by evolutionary processes. Our second example comes from software design: the TCP protocol family, which was designed by human designers over several decades in response to the exponentially growing Internet. Other interesting examples are given in [33] (subsumption architecture, fault tolerance in Erlang) and [34] (human endocrine system, Hill equations, collective intelligence).

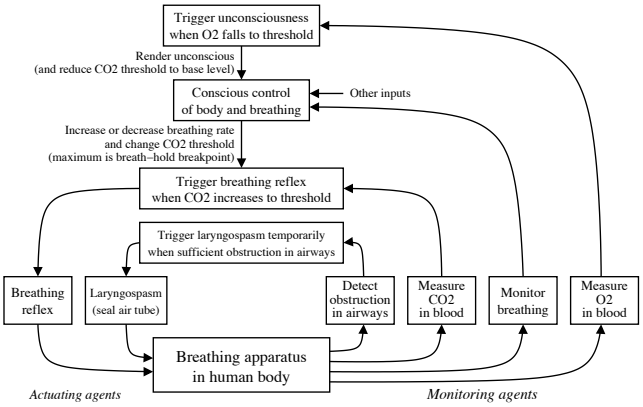


Figure 2: The human respiratory system as a feedback structure

2.1 The human respiratory system

Successful biological systems survive in natural environments, which can be particularly harsh. We study them to gain insight in how to design robust software. Figure 2 shows the parts of the human respiratory system and how they interact. We derived this figure from a precise medical description of the system's behavior [38]. 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). Three loops make a hierarchical tower which interacts using stigmergy with the fourth loop. From this figure we can deduce what happens in many realistic cases. For example, holding one's breath increases the CO_2 threshold so that the breathing reflex is delayed. Eventually the breath-hold threshold is reached and the breathing reflex happens anyway. For a trained person 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. For example, if a person falls into a lake, conscious control of breathing is part of a swimming movement to get to the shore. We can justify why conscious control 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 structures. This very power means that it needs a check.

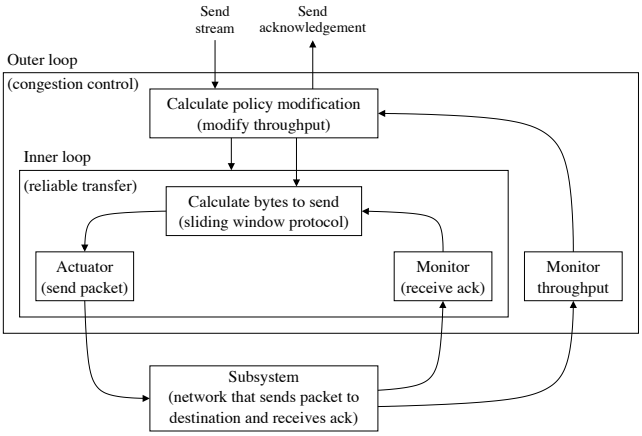


Figure 3: TCP as a feedback structure

2.2 Transmission Control Protocol (TCP)

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 feedback loops that interact hierarchically to implement a reliable byte stream transfer protocol with congestion control [15]. The protocol sends a byte stream from a source to a destination node. Figure 3 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 implements a sliding window: the actuator sends packets so that the sliding window can advance. The slid-

ing 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.

These two loops are part of a much larger feedback structure, in which the individual TCP connections all share a common network. Congestion is felt by all congestion control loops, which will all reduce their window sizes. This is an example of collaboration using stigmergy. It causes the overall throughput to increase, since the network no longer wastes its resources transmitting packets that will be dropped before reaching their destination.

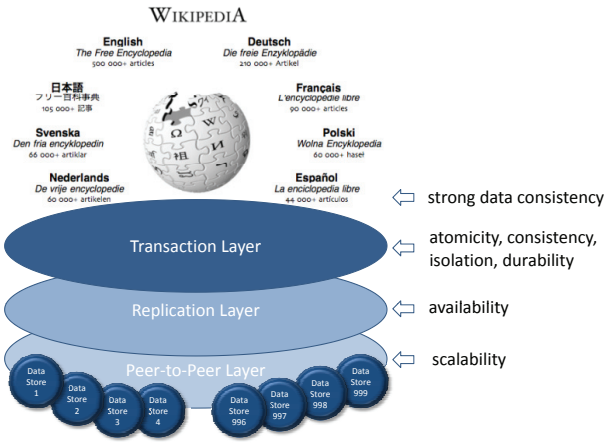


Figure 4: Distributed Wikipedia built on top of Scalaris

3. SCALARIS

Scalaris is an open-source library providing a self-managing data management service for Web 2.0 applications [27, 25, 24]. Web 2.0 initiated a business revolution: service providers offer Internet services for many activities, shopping, online banking, information, social networking, and recreation. In today's society Web 2.0 is no longer a convenience, but customers rely on its continuous availability, regardless of time and space. Even the shortest interruption, caused by system downtime or network partitioning, may cause huge losses in reputation and revenue. In addition to 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Continuous availability, high performance, and scalability were key requirements in the design of Scalaris. To satisfy these requirements, we designed Scalaris to be self managing.

As a challenging benchmark for Scalaris, Figure 4 shows how we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". Wikipedia is among the ten most frequently accessed websites. It handles about 50,000 requests per second, of which 48,000 are cache hits in the proxy server layer and 2,000 are processed by ten servers in the master/slave MySQL database layer [39]. The proxy and web server layers are embarrassingly parallel and therefore trivial to scale. From a scalability point of view, only the database layer is challenging. Because our implementation uses Scalaris to replace the database layer, it inherits all the favorable properties of Scalaris such as scalability and self management. Instead of using a relational database, we map the Wikipedia content to the Scalaris key/value store. On a page update, a transaction across all affected keys (content, backlinks, categories, etc.) and their replicas is done. With a synthetic benchmark, Scalaris achieves 14,000 read+write transactions per second on 15 servers, increasing almost linearly with the number of servers [26]. This number cannot be directly compared to the Wikipedia number since the work and the processors are not the same, but it does show that Scalaris is a credible implementation.

We have built a second library, Beernet, that differs from Scalaris in some important points. Whereas Scalaris is based on a Chord# overlay network, Beernet uses a relaxed ring structure [21, 20]. We relax the connectivity condition, requiring only that a node be in the same ring as its successor (instead of both its successor and predecessor). Ring maintenance then does not need periodic stabilization and does not rely on transitive connectivity. The relaxed ring has a "bushy" structure that converges with local operations to a perfectly connected ring. We also modify the transaction manager to request locks quickly and to notify all nodes of modified state. We need these modifications for our collaborative drawing application, DeTransDraw, which uses transactions to overcome network delays while maintaining a coherent global drawing.

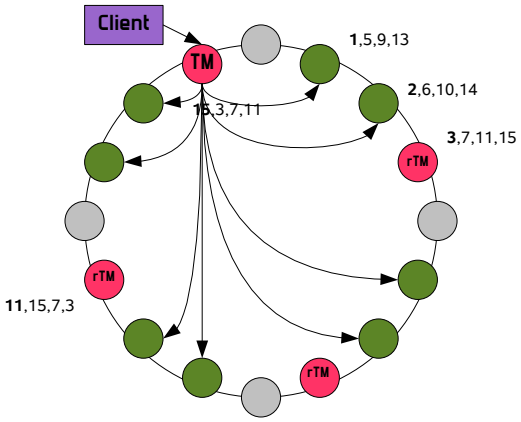


Figure 5: The Scalaris transaction protocol

3.1 Transactions on an overlay network

Scalaris is a structured overlay network extended with a transaction layer using a replicated key/value storage. Its architecture provides the traditional ACID properties of transactions in a scalable decentralized setting. It does not attempt to replace current database management systems with their general, full-fledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multiplayer online games. Figure 4 shows the three layers of the system:

1. At the bottom, an enhanced structured peer-to-peer network, with logarithmic routing performance, pro-

vides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographical ordering instead of random hashing enables control of data placement which is necessary for low latency access in multi data center environments.

2. The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures. We use symmetric replication, in which the data is replicated symmetrically around the ring.

3. The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses a fast consensus protocol with low communication overhead that has been optimally embedded into the peer-to-peer network.

Figure 5 shows how the transaction protocol works on a structured peer-to-peer network with 16 nodes. A client initiates a transaction by asking its nearest node, which becomes a transaction manager. Other nodes that store data are transaction participants. Given symmetric replication with degree f (4 in the figure), we have f transaction managers (TM and rTM in the figure) and f replicas for the other participating nodes. A modified version of Lamport's Paxos uniform consensus algorithm is used for node agreement [19, 12]: each replicated transaction manager (rTM) collects votes from a majority of participants and locally decides on abort or commit. The transaction manager (TM) then collects a majority from the replicated transaction managers and sends its decision to all participants. This algorithm achieves commitment if more than $f/2$ nodes of each replica group are alive. The algorithm's operation seems simple; things are actually more subtle because it is correct even if nodes can at any time be falsely suspected of having failed. All we know is that after some unknown finite time, the failure suspicions are correct (eventually perfect failure detection). In our experience, this failure detector is adequate for an Internet setting, where nodes may crash and communication may be interrupted.

3.2 Feedback structures in Scalaris

Instead of the conventional layered presentation of the previous section, we can present the architecture of Scalaris in a more enlightening way as a set of six feedback structures and their interactions:

1. *Connectivity management.* This feedback structure maintains the connectivity of the ring using periodic successor list stabilization.

2. *Merge management.* This feedback structure monitors when it is possible to merge the ring after it has split into several rings due to network partitioning or other network problems. It uses the merge algorithm to converge continuously to a single ring [30].

3. *Routing management.* This feedback structure maintains efficient routing tables using periodic finger stabilization.

4. *Load balancing.* This feedback structure balances load by monitoring each node and moving nodes when necessary to distribute load evenly.

5. *Replica management.* This feedback structure maintains the invariant that there will always eventually be f replicas of each data item. Whenever there is a potential new replica, it uses consensus to propose a new replica set.

6. *Transaction management.* This feedback structure uses consensus among replicated transaction managers and storage nodes to perform atomic commit. If the transaction manager TM fails, then one of its replicas rTM takes over. Multiple takeovers are tolerated by consensus.

The Scalaris specification then consists of the conjunction of the six properties implemented by these feedback structures. Interactions between the feedback structures are possible when the perceived set of correct nodes changes, due to nodes joining, leaving, failing, or suspected of failing. We handle the interactions as follows:

- Connectivity management, replica management, and routing management interact when the set of nodes changes. This does not affect correctness because each manager always converges towards its ideal solution. Oscillations do not occur because there are no cyclic dependencies (connectivity management is not affected by the other two). We choose the time delays of the different managers to improve efficiency.

- Replica management and transaction management interact because the number of replicas can change. This may affect consistency if there are temporarily more than f replicas. This is an extremely rare situation, but it can be handled by changing the majority criterium of the consensus algorithm.

- Covert stigmergy between feedback structures may occur because the network is a shared resource. Connectivity and merge management messages must be given priority over other messages, since otherwise the overlay network may become disconnected at high loads. To minimize other bad effects due to stigmergy, the management load on the network should be kept as constant as possible. If connectivity management does less work, then routing management takes up the slack.

Because these six feedback structures act at all layers of the system, we can say that the Scalaris implementation is self managing *in depth*. For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. In traditional database systems, changing system size and tuning require human interference which is error prone and costly. In both these situations, the same number of administrators in Scalaris can operate much larger installations.

4. DESIGN GUIDELINES

A self-managing application consists of a set of interacting feedback structures, each of which manages one system property [36]. For this reason, we sometimes call a feedback structure a "manager". We first explain how to design one feedback structure and then we explain how to combine feedback structures to make the complete system. This section gives a partial methodology; the complete methodology is still a subject of future research.

4.1 Designing one feedback structure

A feedback structure consists of a set of feedback loops that collaborate together. An important design rule is that each feedback loop should target a separate part of the management. In the TCP example, the inner loop implements the sliding window and the outer loop does congestion control by changing a parameter of the inner loop. Each feedback loop can then be designed and optimized separately using control theory [14] or discrete systems theory [8]. This works well for feedback loops that are mostly independent. If the feedback loops interact in a stronger way, then the design must take these interactions into account. In a well-designed feedback structure, the interactions will be small and can be handled by small changes to each of the participating feedback loops. It can happen that parts of the feedback structure do not fit into the "mostly separable single feedback loops" pattern. We have encountered several examples of this in the SELFMAN project. In that case we recommend the following approach:

- In the case of a large number of agents that collaborate, the best approach is to design a distributed algorithm [12] or a multi-agent system [31] to perform the task. For example, in SELFMAN we needed an algorithm to perform atomic commit for distributed transactions, in the face of possible node failures and communication interruptions (imperfect failure detection). We found that a modified version of the Paxos uniform consensus protocol was an essential part of the solution. This is a complex algorithm whose correctness is not trivial to prove [12, 23]. Instead of trying to reinvent it in terms of interacting feedback loops, we used the existing knowledge about this algorithm.

- In the case when the feedback structure consists of more than one loop intimately tied together, the global behavior must be determined by analyzing the structure as a whole and not by trying to analyze each loop separately. To our knowledge, no general methodology for doing this exists. We have made progress on two fronts: design rules and patterns for common feedback structures. We have made a comprehensive survey of feedback loop patterns [6]. Some commonly occurring patterns, such as "Tragedy of the Commons", have been extensively studied in the literature. Unfortunately, the literature is extremely fragmented. Studies of feedback loop patterns exist in widely different disciplines, such as business management [29], biology [7, 22], and computer science [14]. Complete classification of these patterns is still future research.

4.2 Combining feedback structures

We now explain how to build a system as a set of feedback structures. This is done in two steps, decomposition and orchestration [2]. Decomposition divides the overall management into separate feedback structures. Orchestration handles the interactions between these feedback structures.

In decomposition, each task focuses on a single property of the system and is performed by a single feedback structure. For example, in the distributed store we distinguish connectivity, efficient routing, load balance, replicated storage, and transactions. Each of these is done by a different feedback structure. Connectivity is done through ring maintenance and the merge algorithm. Efficient routing is done through finger table maintenance. Load balancing is done through a load distribution algorithm. Replicated storage is done through the symmetric replication algorithm. Transactions are done through the replicated transaction managers.

For a successful orchestration, it is crucial to perform the right decomposition. The managers should be independent or interact only in a simple way. Because interactions can be subtle (see Section 3.2), it is important to simplify them as much as possible at design time. For the different kinds of interactions, we give design rules to achieve this. We then enumerate all possible interactions and modify the system so they do not result in undesirable behavior.

4.2.1 Handling interactions

We identify three ways in which managers can interact and we explain how to handle them [1]:

- *Stigmergy.* This occurs when managers make changes to a shared subsystem. Each change made by a manager may be sensed by another manager. This is the most common and is often hard to control. It is a powerful way to communicate for managers that otherwise have no direct communication channel, such as the TCP congestion control loops. Since stigmergic communication tends to be noisy, the managers must be designed to tolerate this.

- *Hierarchy.* This occurs when one manager directly controls another. This situation often occurs inside a single feedback structure, when an outer loop controls an inner loop. For example, it occurs inside the TCP structure and in the human respiratory system. To handle this, we choose the control parameter to be a natural parameter of the system being controlled and we model the control in terms of this parameter.

- *Direct interaction.* This occurs when two managers interact as peers. It does not mean that one manager controls the other, but one manager may interact with another. Direct interaction is sometimes needed since two independent managers affecting the same resource may cause undesired behavior, such as races or oscillation. It must be handled carefully to avoid replacing one kind of undesired behavior by another. We can avoid many problems by designing each manager around a monotonic function with a limiting value that corresponds to perfect behavior. Each manager then increases its own function in discrete steps.

4.2.2 Build the system in the right order

An important technique to reduce the interaction of feedback structures is to add the different properties in the right order. In this way, each new property can be added in (almost) orthogonal fashion to the system. For the transactional store, we propose the following order:

- The first property is self healing: the structured peer-to-peer network is based on a ring structure and uses feedback loops to repair the ring if a node joins, leaves, or fails, or to repair network partitioning.

- We add self tuning in two steps. The first step is to add extra routing links to the nodes (called "fingers" in the literature) to make the routing efficient. This

is done through a feedback loop that continuously corrects the fingers depending on the changing structure of the ring. The second step is to update the ring dynamically to remove hotspots. This is done through a feedback loop that periodically collects node load information and performs balancing operations in which an unloaded node leaves the ring and rejoins near a loaded node to take over some of the load.

- We add self configuration. Components use the efficient routing to communicate, in particular to inform nodes when to add or remove new components.

- Finally, we add self protection. We continuously modify the ring's topology to approach a small-world network, which is resistant to certain kinds of collusion. We also add an observer of node behavior that can eject bad nodes from the ring. This form of self protection protects against malicious users; it does not protect against attacks to the infrastructure itself.

5. CONCLUSIONS AND PROSPECTS

To tame the complexity of Internet applications, we propose to build them using feedback structures. Each feedback structure consists of a hierarchy of feedback loops that together monitor and correct one system property. Feedback structures interact minimally and in a well-defined way. No part of the system should exist outside of a feedback structure. We motivate this approach by giving examples of real systems taken from biology and software (the human respiratory system and the Internet TCP protocol family). In our own work in the SELFMAN project [27, 20, 28, 34], we have built structured peer-to-peer networks that survive in realistically harsh environments (with imperfect failure detection and network partitioning). We have developed software, including the Scalaris and Beernet libraries and the Distributed Wikipedia and the DeTransDraw collaborative drawing tool, to show that our solutions are credible. The Scalaris architecture consists of six feedback structures whose interactions are carefully controlled.

5.1 Reversible phase transitions

We have shown that a structured peer-to-peer network can react to a hostile environment by doing a reversible phase transition [35]. To be precise, if the network is partitioned, then the overlay network continues to work as several smaller overlays. If the partition goes away, then a merge algorithm is run that merges the smaller overlays back into a single large overlay [30]. This is an exact analogy to a physical phase transition as explained by thermodynamics [10, 13]. This behavior is predictable and can be exposed to the application as an API so that it can be written to survive the transition. Important research questions are how to design a system together with determining its complete (reversible) behavior in phase space, how phase transitions should be exposed to an application as an API, and how they should affect application design. We are preparing a followup project to SELFMAN to answer these questions.

5.2 A complete and justified methodology

We have motivated why it is useful to design systems using feedback structures and we have presented our own techniques in this area. For practical system design, it is important to have a complete methodology that is formally justified and that allows to design systems with desired global properties. To our knowledge, no such methodology exists yet. Most of the knowledge in this area is fragmented and deriving formal properties is difficult. Formal analysis of systems with multiple interacting feedback loops is difficult [17]. Techniques from theoretical physics are necessary to show the existence of phase transitions [18]. Clearly, it is not practical for a system developer to do formal analysis at this level.

We propose a research agenda to create a complete methodology. First we study existing feedback loop systems to build a library of patterns and rules. Second we translate the patterns and rules into a process calculus. The translation should be correct in a formal sense, e.g., according to the definition of abstract interpretation [9]. Third we prove the relevant properties of the patterns and rules. Important properties include global correctness, stability, compositionality, and phase behavior. Finally, we step back from the formal treatment and use the original patterns as design elements following the rules. The developer can rely on the proofs without having to do any formal analysis. We consider the creation of this methodology as one of the most important tasks for software development as the Internet continues to grow in complexity.

6. ACKNOWLEDGMENTS

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265).

7. REFERENCES

[1] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. *Distributed Control Loop Patterns for Managing Distributed Applications.* Workshop on Decentralized Self Management for Grids, P2P, and User Communities (part of SASO 2008), Oct. 21, 2008.

[2] Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi. *A Design Methodology for Self-Management in Distributed Environments.* GRID4ALL project.

[3] Joe Armstrong. "Making reliable distributed systems in the presence of software errors". Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, Nov. 2003.

[4] Ken Birman, Gregory Chockler, and Robbert van Renesse. *Toward a Cloud Computing Research Agenda.* ACM SIGACT News, 40(2), June 2009, pp. 68-80.

[5] Rodney A. Brooks. *A Robust Layered Control System for a Mobile Robot.* IEEE Journal of Robotics and Automation, RA-2, April 1986, pp. 14–23.

[6] Alexandre Bultot. "A Survey of Systems With Multiple Interacting Feedback Loops and Their Application to Programming". Master's report, Université catholique de Louvain, August 2009.

[7] Scott Camazine, Jean-Louis Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. "Self-Organization in Biological Systems". Princeton University Press, 2001.

[8] Christos G. Cassandras and Stéphane Lafortune. "Introduction to Discrete Event Systems". Second

Edition. Springer-Verlag, 2008.

[9] Patrick Cousot 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.

[10] Ernest G. Ehlers. "The Interpretation of Geological Phase Diagrams". Dover Publications, 1987. Originally published by W.H. Freeman and Company, 1972.

[11] Gary William Flake. "The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation". MIT Press, 2001.

[12] Rachid Guerraoui and Luís Rodrigues. "Introduction to Reliable Distributed Programming". Springer-Verlag, 2006.

[13] Hermann Haken. "Synergetics: An Introduction. Nonequilibrium Phase Transitions and Self-Organization in Physics, Chemistry, and Biology". Third Edition. Springer-Verlag, 1983.

[14] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. "Feedback Control of Computing Systems". Wiley-IEEE Press, August 2004.

[15] Information Sciences Institute. "RFC 793: Transmission Control Protocol Darpa Internet Program Protocol Specification". Sept. 1981.

[16] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing*. IEEE Computer 36(1), pp. 41-50, Jan. 2003.

[17] Supriya Krishnamurthy, 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.

[18] Supriya Krishnamurthy 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).

[19] Leslie Lamport. *The part-time parliament*. ACM Trans. Comput. Syst. 16(2), pp. 133-169, 1998.

[20] Boris Mejías. "Algorithms for Self-Managing Large-Scale Decentralized Networks". Ph.D. dissertation, Université catholique de Louvain, 2009 (in preparation). See `beernet.info.ucl.ac.be`.

[21] Boris Mejías and Peter Van Roy. *The Relaxed-Ring: A Fault-Tolerant Topology for Structured Overlay Networks*. Parallel Processing Letters 18(3), pp. 411-432, Sept. 2008.

[22] Gerhard Michal (ed.). "Biochemical Pathways: An Atlas of Biochemistry and Molecular Biology". John Wiley & Sons and Spektrum Akad. Verlag, 1999.

[23] Monika Moser and Seif Haridi. *Atomic commitment in transactional DHTs*. In CoreGRID Symposium, August 2007.

[24] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. *Transactions for Distributed Wikis on Structured Overlays*. In A. Clemm, L. Z. Granville, and R. Stadler, editors, DSOM, Springer LNCS volume 4785, 2007, pp. 256-267.

[25] Alexander Reinefeld, Florian Schintke, Thorsten Schütt, and Seif Haridi. *A Scalable, Transactional Data Store for Future Internet Services*. In Towards the Future Internet, G. Tselentis *et al* (eds.), IOS Press, 2009.

[26] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. "Enhanced Paxos Commit for Transactions on DHTs". ZIB-Report 09-28, Zuse Institute Berlin, Sept. 2009.

[27] Thorsten Schütt. "Scalaris: A Scalable Transactional Data Store for Web 2.0 Services". Technical report, Zuse Institute Berlin, 2008. See `code.google.com/p/scalaris`.

[28] SELFMAN: Self Management for Large-Scale Distributed Systems Based on Structured Overlay Networks and Components. European Commission 6th Framework Programme three-year project, June 1, 2006–Sept. 30, 2009. See `www.ist-selfman.org`.

[29] Peter M. Senge, Art Kleiner, Charlotte Roberts, Richard B. Ross, Bryan J. Smith. "The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization". Nicholas Brealey Publishing, 1994.

[30] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*. Journal of Peer-to-Peer Networking and Applications (PPNA), 2009 (to appear).

[31] Yoav Shoham and Kevin Leyton-Brown. "Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations". Cambridge University Press, 2009.

[32] Ion Stoica, 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.

[33] Peter Van Roy. *Self Management and the Future of Software Design*. Springer ENTCS 182, June 2007, pp. 201-217. Proceedings of Third International Workshop on Formal Aspects of Component Software (FACS '06), Sep. 2006.

[34] 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*. Springer LNCS volume 5382, 2008, pp. 153-178. Revised postproceedings of FMCO 2007, Amsterdam, The Netherlands, Oct. 2007.

[35] Peter Van Roy. *Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions*. First International Conference on Visions of Computer Science (BCS08), London, UK, Sep. 22-24, 2008.

[36] Peter Van Roy. *Guidelines for Building Self-Managing Applications*. SELFMAN project deliverable D5.7, 2009. See `www.ist-selfman.org`.

[37] Norbert Wiener. "Cybernetics, or Control and Communication in the Animal and the Machine". MIT Press, Cambridge, MA, 1948.

[38] Wikipedia, the free encyclopedia. Entry "drowning", August 2006. See `en.wikipedia.org/wiki/Drowning`.

[39] Wikipedia, the free encyclopedia. Entry "wikipedia", 2009. See `en.wikipedia.org/wiki/Wikipedia` (section *Software and Hardware*).

A.2 Building and Evaluating P2P Systems using the Kompics Component Framework

Building and Evaluating P2P Systems using the Kompics Component Framework

Cosmin Arad
Royal Institute of Technology (KTH)
icarad@kth.se

Jim Dowling, Seif Haridi
Swedish Institute of Computer Science (SICS)
jdowling,seif @sics.se

***Abstract*—We present a framework for building and evaluating P2P systems in simulation, local execution, and distributed deployment. Such uniform system evaluations increase confidence in the obtained results. We briefly introduce the Kompics component model and its P2P framework. We describe the component architecture of a Kompics P2P system and show how to define experiment scenarios for large dynamic systems. The same experiments are conducted in reproducible simulation, in real-time execution on a single machine, and distributed over a local cluster or a wide area network.**

This demonstration shows the component oriented design and the evaluation of two P2P systems implemented in Kompics: Chord and Cyclon. We simulate the systems and then we execute them in real time. During real-time execution we monitor the dynamic behavior of the systems and interact with them through their web-based interfaces. We demonstrate how component-oriented design enables seamless switching between alternative protocols.

***Keywords*-peer-to-peer; evaluation; component framework; design; simulation; experimentation; deployment.**

I. Introduction

Comprehensive evaluation of P2P systems comprises analysis, simulation, and live performance measurements. We present Kompics [1], a model for building reconfigurable distributed systems from event-driven components. Kompics systems can be uniformly evaluated in large-scale reproducible simulation and distributed deployment, using both the same system code and the same experiment scenarios.

Very similar in spirit, but without a hierarchical component model, is the ProtoPeer [2] toolkit for prototyping and evaluating P2P systems. Mace [3] generates distributed systems code from a high-level specification while Splay [4] allows system specification in a high-level language.

II. Kompics and the P2P Component Framework

Kompics is a component model targeted at building distributed systems by composing protocols programmed as event-driven components. Kompics components are reactive state machines that are executed concurrently by a set of workers. Components communicate by passing data-carrying typed events through typed bidirectional ports connected by channels. Ports are event-based component interfaces. A port type represents a *service* or a protocol *abstraction*. It specifies the types of events sent through the port in each direction. A component either provides (+) or requires (–) a port. Components may encapsulate subcomponents.

*This work was funded by the SELFMAN EU project, contract 34084.

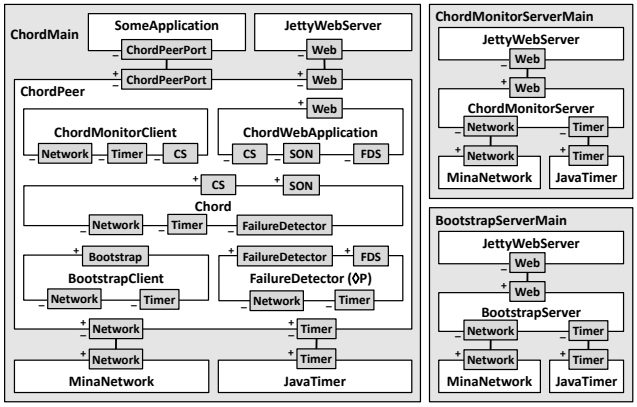


Figure 1. The left figure shows the architecture of a Chord process. The Chord protocol is implemented by the Chord component using Network, Timer, and FailureDetector abstractions. The Network and Timer abstractions are provided by the MinaNetwork [5] (which handles connection management and message serialization) and JavaTimer components. The ChordMonitorClient periodically inspects the Chord status (CS port) and sends it through the network to the ChordMonitorServer (top right). The ChordWebApplication renders this status on a web page upon request from the JettyWebServer [6] (which provides web browser access). On the right we have the component architectures of the monitoring and bootstrap server.

The Kompics runtime supports pluggable component schedulers. The default scheduler is multi-threaded and executes components in parallel on multi-core machines. We use a single-threaded scheduler for reproducible simulation.

We developed a set of utility components and methodology for building and evaluating P2P systems. Service abstractions for network and timers can be provided by different component implementations. The framework contains reusable components that provide bootstrap and failure detection services. System-specific components are developed for global system monitoring and web-based interaction. We highlight the elements of the P2P framework in the architecture of our Chord implementation illustrated in Figure 1.

III. Defining an experiment scenario

We designed a Java domain-specific language (DSL) for expressing experiment scenarios for P2P systems. We call a *stochastic process*, a finite random sequence of events, with a specified inter-arrival time distribution. Here is an example scenario composed of 3 stochastic processes:

```
StochasticProcess boot = new StochasticProcess() {{
   eventInterArrivalTime(exponential(2000)); // ~2s
   raise(1000, chordJoin, uniform(16));  }}; // 1000 joins
StochasticProcess churn = new StochasticProcess() {{
   eventInterArrivalTime(exponential(500));// ~500ms
   raise(500, chordJoin, uniform(16));     // 500 joins
   raise(500, chordFail, uniform(16)); }}; // 500 failures
StochasticProcess lookups = new StochasticProcess() {{
   eventInterArrivalTime(normal(50));      // ~50ms
   raise(5000, chordLookup, uniform(16), uniform(14)); }};
boot.start();                              // start
churn.startAfterTerminationOf(2000, boot); // sequential
lookups.startAfterStartOf(3000, churn);    // in parallel
terminateAfterTerminationOf(1000, lookups);// terminate
```

1000 peers join in a space of . The inter-arrival time between 2 consecutive joins is exponentially distributed with a mean of 2s. A churn process starts 2s after. Every 500ms on average (exp), a new peer joins or an existing peer fails. In parallel with the churn process, 5000 lookups are initiated uniformly around the ring ( ) for keys in the first ring quadrant ( ). The experiment terminates 1s after lookups are done.

IV. Experiment profiles

We can reuse the same experiment scenario to drive simulation or local real-time execution experiments, as well as remote experiments where the system nodes are distributed over the machines of a cluster (possibly running ModelNet [7]) or a testbed like PlanetLab [8] or Emulab [9].

During simulation and local execution (see Figure 2) we model the network at the message-level. In simulation, we execute the same system code built for deployment. Calls for the current system time are trapped and the current simulation time is returned. Simulation enables deterministic replay, debugging, reproducible results, and large-scale experiments without loss of accuracy.

We developed an infrastructure for deploying and executing distributed experiments. Experiment scenarios are locally interpreted by a Master component which coordinates a set of remote Slaves. Each Slave resides on a machine available for the experiment and it manages a set of system peers.

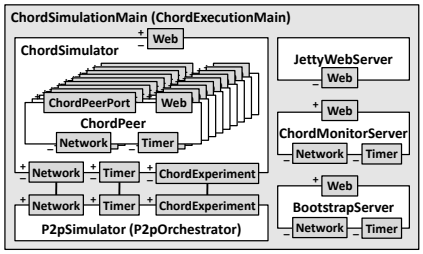


Figure 2. The simulation architecture with all peers and the bootstrap and monitor servers within one process. ChordSimulationMain is executed using a single-thread simulation scheduler for deterministic replay and simulated time advancement. P2pSimulator is generic. It interprets experiment scenarios and sends system-specific scenario events (e.g. chordJoin, chordLookup) to the ChordSimulator which manages the ChordPeers (same from Figure 1). The P2pSimulator provides a Network abstraction and can be parameterized with a custom network latency and bandwidth model. For real-time local execution we replace the P2pSimulator with a P2pOrchestrator, which interprets the same experiment scenario but in real time. In addition, components are executed by the default multi-threaded scheduler.

V. Demonstration overview

This demonstration consists of evaluations of two P2P systems developed in Kompics: Chord [10] and Cyclon [11]. Each system is first evaluated in a reproducible simulation experiment. We reuse the same experiment scenario to execute the systems in real time. We observe the dynamic behavior of the systems though the web interface of the monitoring server, which aggregates the global system state periodically. We also inspect the local state of a few system nodes though their web interfaces and we interact with Chord by manually issuing lookups from different nodes.

We reuse the same scenario definition to drive a distributed experiment where nodes are deployed remotely on some cluster machines or on PlanetLab [8]. We repeat some of the previous system interactions. This illustrates the uniform experience of evaluating real systems across simulation, local execution, and distributed deployment.

We use a BitTorrent [12] system developed in Kompics, in a simulation experiment, to demonstrate a realistic bandwidth emulation model. Finally, we return to local execution to experiment live with different scenario definitions.

VI. Summary

We briefly introduced the Kompics component model and we described the component architecture of the Chord overlay developed using the Kompics P2P framework. We showed how to define experiment scenarios for large and dynamic systems and how the same experiments are conducted in reproducible simulation, in real-time execution on a single machine, and distributed over a local cluster or a wide area network.

The source code used for this demonstration, including the Kompics runtime, the P2P framework, experiment scenarios, and implementations of Chord, Cyclon, and BitTorrent, is available online at `http://kompics.sics.se`.

References

[1] C. Arad, J. Dowling, and S. Haridi, "Developing, simulating, and deploying peer-to-peer systems using the Kompics component model," in *COMSWARE'09*.

[2] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer, "Protopeer: a p2p toolkit bridging the gap between simulation and live deployement," in *SimuTools*, 2009.

[3] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," in *PLDI '07*.

[4] L. Leonini, E. Rivière, and P. Felber, "Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *NSDI'09*.

[5] (2004-2009) Apache MINA. [Online]. Available: http://mina.apache.org/

[6] (1995-2009) Mortbay Jetty. [Online]. Available: http://www.mortbay.org/jetty/

[7] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 2002.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, 2003.

[9] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI'02*.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *IEEE Transactions on Networking*, vol. 11, February 2003.

[11] S. Voulgaris, D. Gavidia, and M. Steen, "Cyclon: Inexpensive membership management for unstructured P2P overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, June 2005.

[12] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. 1st Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2003.

A.3 Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers

Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers*

Thorsten Schütt, Alexander Reinefeld, Florian Schintke, and Marie Hoffmann
Zuse Institute Berlin

Abstract

We present a distributed algorithm for identifying the location of data centers and their relative sizes. This topology information can be used in P2P systems to improve the routing performance, replica placement, or job scheduling.

The algorithm uses gossiping with local agglomerative clustering. It is robust to failures and it correctly identifies outliers that are caused, e.g., by temporarily overloaded nodes or network failures. We present empirical results on the Grid 5000 testbed.

1 Introduction

When deploying P2P systems in the Internet, it is important to minimize the stretch between the under- and the overlay network. Knowing the current topology of the underlay allows to speed up the routing process, to improve replica placement, and to optimize gang scheduling of parallel processes—to name just a few of the many advantages.

With the recent focus on *cloud computing* and *data centers* topology-aware process placement became an important research topic. Widely varying latencies of inter- and intra-center links and correlated resource failures make it difficult to determine an optimal process mapping. If the topology and size of data centers were known, existing P2P protocols [9] could be optimally mapped onto data centers to provide better access latency with less maintenance traffic [11].

We present an algorithm for gossip-based topology inference which uses only local knowledge. It derives the network topology by continuously checking the network latencies between nodes with a *network coordinate system* [1, 12] and grouping them with *agglomerative clustering* [2]. This allows to identify clusters of nearby located nodes and to detect outliers (e.g., caused by temporary overload) with high confidence. The algorithm is robust with respect to failures and it finds node clusters in a logarithmic number of distributed communication steps.

*Part of this work was carried out under the SELFMAN and XtreemOS projects funded by the European Commission

While topology inference is our main focus in this paper, the clustering algorithm can also be used for other tasks. As an example, we determined CPU speeds and memory sizes of the nodes in the Grid 5000 testbed (Sec. 4). The resulting resource classes can be used to aid Grid schedulers or to determine optimal replica placement in P2P systems [9].

In Section 2 we briefly recall gossiping, network coordinate systems and clustering techniques. In Section 3 we present our gossip-based algorithm for agglomerative node clustering and in Section 4 we present results obtained on the Grid 5000 environment. Section 5 gives a brief conclusion and outlook.

2 Background

Gossiping [7] is a well-established method in distributed systems. It is used for information dissemination, information aggregation, peer sampling [10], and overlay construction [5]. Gossip algorithms are easy to implement, they are tolerant to node failures and they converge fast—usually in a logarithmic number of communication rounds. In each gossip operation, an active node selects a peer at random, exchanges information with it, and updates its local data with the received information.

Network Coordinate Systems are used to build a model that predicts the round-trip time between any two nodes. The algorithm starts in each node by assigning itself a random coordinate which is subsequently refined by exchanging coordinate information with other nodes to improve the prediction quality. We use the gossip-based Vivaldi [1] algorithm with a 2-dimensional Euclidean coordinate space. Several models for network coordinate systems have been evaluated [1], including higher-dimensional Euclidian spaces, Euclidian spaces with height vectors representing the individual delays of the access line to the Internet core network (e.g. queueing and DSL link delays, oversubscribed links) and spherical coordinates which were initially deemed to model the earth surface best. For our purpose, simple 2D coordinates are sufficient.

Clustering is a common technique in data mining [4, 3] to group data so that the members of a group have similar properties. Clusters are usually represented by centroids,

```
1  // active thread
2  Peer p := selectRandomPeer()
3  sendTo(p, centroids)
4  receiveFrom(p, remoteCentroids)
5  // aggregate data
6  centroids := centroids ∪ remoteCentroids
7  centroids := agglClustering(centroids, r)
8  centroids := normalize(centroids)
```

```
1  // passive thread
2
3  (p, remoteCentroids) := receiveFromAny()
4  sendTo(p, centroids)
5  // aggregate data
6  centroids := centroids ∪ remoteCentroids
7  centroids := agglClustering(centroids, r)
8  centroids := normalize(centroids)
```

Figure 1. Framework for gossip-based clustering.

```
1  agglClustering(centroids, radius):
2    // get indices of closest centroids
3    (p, q) := closestPoints(centroids)
4    while size(centroids) > 1
5      and dist(p, q) < radius:
6      // aggregate sizes
7      size := p.size + q.size
8      // merge closest centroids
9      new_centroid := (p.centroid*p.size
10         + q.centroid*q.size)/size
11     // update centroids
12     centroids.remove(p)
13     centroids.remove(q)
14     centroids.add(new_centroid, size)
15     (p, q) := closestPoints(centroids)
16   return centroids
```

Figure 2. Agglomerative clustering.

i.e. the centers of the clusters. Two widely used algorithms are *k-means clustering* [8] and *agglomerative clustering* [2]. The former clusters the data points into k groups and tries to minimize the distance of data points from their respective centroids. The latter clusters data points in the same group iff their similarity is below a given threshold. In k-means the number of clusters is given as a parameter, while in agglomerative clustering the number of clusters is determined by the similarity threshold and the data distribution.

We focus on agglomerative clustering because the number of data centers is generally not known in advance. We define a similarity metric based on network latency and give a latency threshold for when nodes are located in the same site.

3 Gossip-based Clustering Algorithm

Each node executes the algorithm shown in Fig. 1. The algorithm consists of an active thread which initiates the communication and a passive thread that waits for incoming messages. Each node maintains a list of already detected clusters with their centroids and relative sizes. The sizes of all clusters sum up to 1.

On startup, each node initializes its centroids list with its own coordinate and the relative cluster size 1. It then selects a random communication partner with a peer sampling algorithm [10, 6]. The partners exchange their centroids lists and run the agglomerative clustering algorithm (called in line 7 of Fig. 1, function shown in Fig. 2) on the merged lists. The centroids' relative sizes are then normalized so that they sum up again to 1 (line 8 of Fig. 1).

The agglomerative clustering algorithm shown in Fig. 2 iteratively merges the two closest centroids p and q and computes the weighted average (line 9, 10 of Fig. 2) resulting in a new list of centroids.

4 Results

We used our clustering algorithm to determine the network topology and various node attributes of the French Grid 5000 testbed[1]. It comprises 1604 compute nodes spread over nine sites (data centers) which are initially unknown to the algorithm.

4.1 Topology Inference

To determine the topology of Grid 5000, we assign each node a network coordinate. This coordinate is used for initializing the nodes' local view. Since the number of data centers is not known, we use agglomerative clustering (Fig. 2) with a threshold denoting the maximum expected latency inside a data center. The centroids resulting from agglomerative clustering are network coordinates which represent the centers' center of gravity and their relative sizes.

Fig. 4 shows the actual size and location (circles) and the estimated locations ($\diamond$) of the data centers that were identified by our algorithm after $1.5 \log_2 N$ communication rounds, with N being the number of nodes. We simulated 100 nodes based on the Grid 5000 node distribution and plotted all centroids ($\diamond$) identified by all nodes. After $1.5 \log_2 N$ rounds the error is already relatively small. Convergency is shown in Fig. 3 and discussed later.

Detecting Outliers Nodes that are overloaded or weakly linked to the network due to wrong configuration will be slow in answering requests. Consequently, the clustering

[1] https://www.grid5000.fr/

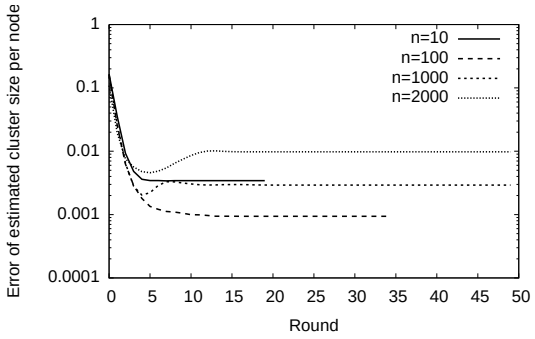


Figure 3. Error reduction per communication round for different grid sizes.

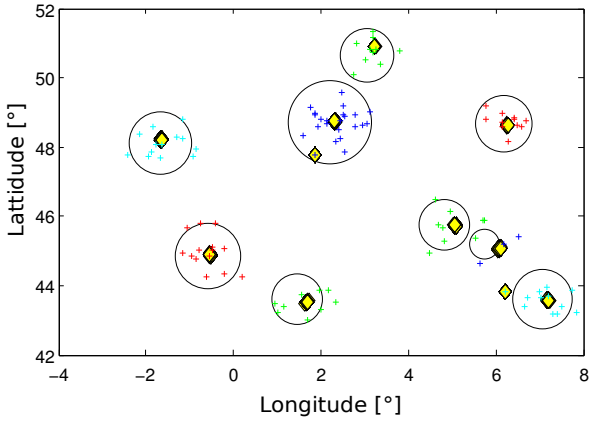


Figure 4. Centroids and relative cluster sizes in Grid 5000. $+$ denotes node locations and $\diamond$ denotes the estimated centroids of each node. The circles' sizes show the actual relative cluster sizes determined by central clustering.

algorithm puts their coordinates far away from all other nodes, resulting in singletons which clearly identifies these points as outliers. Similarly, network failures within a data center can cause a subset of nodes to form a cluster, but again, they are easy to identify, because they will lie far outside of other data centers' clusters.

Convergence To analyze the convergence of the cluster algorithm, let $|c|$ be the total number of centroids in a system with N nodes and let c_i be the centroids in the local view of node i. Then c_{ij} is the jth centroid of c_i and w_{ij} is its relative size. Furthermore, let w_k be the relative size of the closest real centroid c_k obtained by a globally informed clustering algorithm that minimizes the expression $||c_{ij} - c_k||$. Then the average error of the cluster sizes is

$$\text{error} = \frac{1}{|c|} \sum_{i=1}^{N} \sum_{j=1}^{|c_i|} (w_{ij} - w_k)^2$$

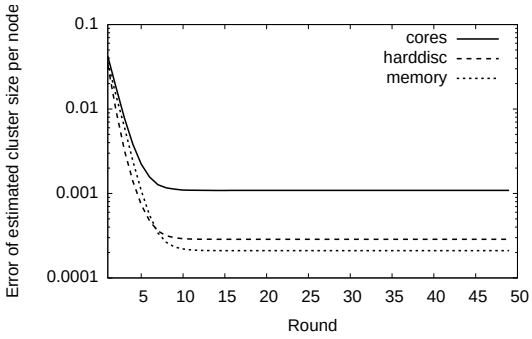


Figure 5. Error reduction per commmunication round.

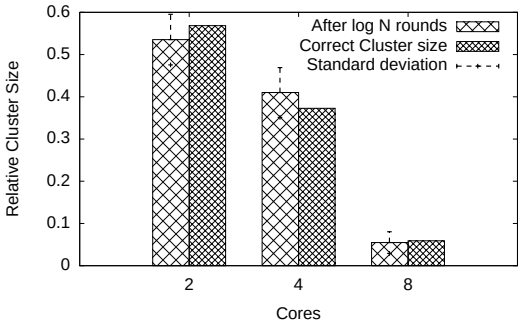


Figure 6. Grid5000 nodes clustered by #cores.

For Fig. 3, we simulated networks with different numbers of nodes. The graph shows that the algorithm converges after $\approx \log_2 N$ rounds. Fig. 5 presents the same metric for clustering node properties (discussed below). As can be seen, the error decreases exponentially, eventually converging after ten rounds.

4.2 Aggregating Resource Data

The nodes in Grid 5000 are heterogeneous and were procured from different vendors. We used our algorithm to additionally aggregate the processor speeds, the number of cores, the main memory sizes, and the hard disk sizes.

Cores For the number of cores, we put nodes in the same cluster if their number of cores differed by less than one. For such integer-valued attributes, the clustering can be precisely steered: If the similarity threshold is set to 1, nodes will only end up in the same cluster when their attribute has exactly the same value. Fig. 6 shows that three centroids with 2, 4, and 8 cores were found in Grid 5000.

The left bars show the average cluster sizes after running $\log_2 N$ communication rounds. For comparison, the

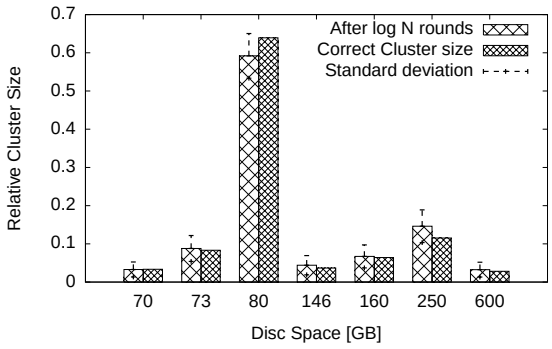


Figure 7. Grid5000 nodes by disc size.

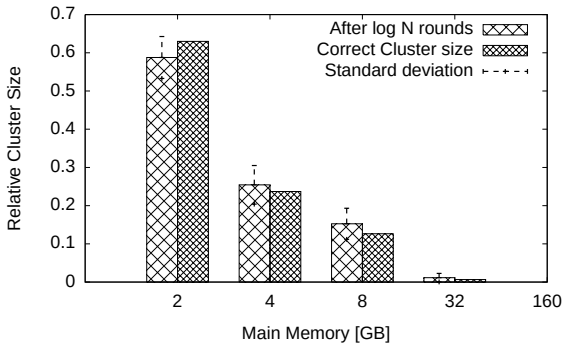


Figure 8. Grid5000 nodes by memory size.

right bars show a (hypothetical) central algorithm with complete knowledge. All data lies in the confidence interval and the average cluster sizes are close to the exact values after $\log_2 N$ rounds (see also Fig. 5).

Disk Size For determining the hard disk sizes, we set the similarity threshold to 1 GB. Hence only disks of exactly the same size are clustered together. Fig. 7 shows that all seven different hard disk sizes were correctly identified and that the relative sizes are also close to the correct values.

Memory Size For computing the main memory sizes we used 1 GB as the similarity threshold. After $\log_2 N$ rounds all four main memory classes were correctly identified (Fig. 8). Although the relative sizes span several orders of magnitude, the approximations are reasonably good.

5 Conclusion and Future Work

We presented a simple, yet powerful gossip-based clustering algorithm for data aggregation in distributed systems. The algorithm is robust with respect to failures and it correctly identifies outliers. Empirical results on Grid 5000 are in good agreement with the actual data.

The algorithm can be used for a wide variety of data aggregation tasks like topology inference, replica placement, or process placement. When the total number of nodes in the overlay is known (or can be approximated), the number of nodes in each cluster can be determined by multiplying its relative size by the total number. This information can be used for mapping data replicas or for job scheduling. In the latter case, it could be beneficial to group nodes e.g. into 'fat' nodes with multiple cores and a large main memory and into 'normal' nodes.

Our work was motivated by the need for obtaining topology information in global P2P networks. We intend to use the clustering algorithm for deploying DHTs onto data centers. The clustering information will be used to improve the routing by adding extra pointers to the routing table so that each routing table has a given number of pointers to each data center. Proximity routing along the extra pointers will then be used to minimize lookup latencies and the relative cluster sizes could be used to decide how many pointers should be placed to each data centers.

References

[1] F. Dabek, R. Cox, M. F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. ACM SIGCOMM 2004.

[2] W. H. E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. Journal of Classification, 1:7-24, 1984.

[3] I. Eyal, I. Keidar, R. Rom. Distributed Clustering for Robust Aggregation in Large Networks. HotDep, Jun. 2009.

[4] A.K. Jain, M.N. Murty, P.J. Flynn. Data Clustering: A Review. ACM Computing Surveys, 31(3), Sept. 1999.

[5] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. ESOA 2005.

[6] M. Jelasity, R. Guerraoui, A. M. Kermarrec, and M. van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. Middleware'04.

[7] A. M. Kermarrec and M. Steen. Gossiping in Distributed Systems. ACM Operating System Review 41(5). Oct. 2007.

[8] H. Steinhaus. Sur la division des corp materiels en parties. Bulletin L'Acadmie Polonaise des Science C1. III, IV, 1956.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. ACM SIGCOMM 2001.

[10] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. J. Network Syst. Manage. 13(2), 2005.

[11] M. Waldvogel and R. Rinaldi. Efficient Topology-Aware Overlay Network. ACM SIGCOMM 2003.

[12] G. Wang and T. S. E. Ng. Distributed algorithms for stable and secure network coordinates. IMC 2008.

A.4 Self-Adaptation in Large-Scale Systems: A Study on Structured Overlays Across Multiple Datacenters

Self-Adaptation in Large-Scale Systems: A Study on Structured Overlays Across Multiple Datacenters

Thorsten Schütt, Alexander Reinefeld, Florian Schintke, Christian Hennig
Zuse Institute Berlin, Germany

***Abstract*—With the recent focus on cloud computing a new type of system topology came up: clusters in geographically distributed datacenters that are connected by high-latency networks. Current structured overlay networks (SONs) are not well prepared for such environments with heterogeneous network performance and correlated node failures.**

We show how the beneficial features of SONs, namely self-management, scalability, adaptability, and fault tolerance can be exploited for multi-datacenter environments. We present self-adaptive replica placement policies and latency-optimized routing for SONs on multiple datacenters. Empirical results of our gossip-based ring maintenance protocol demonstrate its ability to cope with correlated node failures and network partitioning.

I. Introduction

Structured Overlay Networks (SONs) provide self-management and fault-tolerance properties, that are also of interest to providers of datacenters and cloud computing. However, SONs like Chord [19] are ill prepared for deployment over multiple datacenters, as some of their basic assumptions are violated:

- Node failures are no longer independent from each other, but become correlated when a whole data center experiences a power outage or network link failure. To improve the data availability, replicas must be explicitly assigned to different datacenters, rather than placing them randomly by consistent hashing. In Section II we present a flexible data placement scheme for Chord$^{\#}$ [14], a SON without hashing.
- In multi-datacenter scenarios the message delays are either negligible (for *intra*-datacenter), or very long (for *inter*-datacenter). A similar pattern holds true for the bandwidth. To provide faster data access, we present approaches for latency-optimized routing in multi-datacenter SONs in Section III.
- The increased probability of correlated node failures calls for adapted overlay maintenance schemes. We present and evaluate an improved gossip-based ring maintenance algorithm based on T-MAN [8] in Sections IV and V.

Deploying SONs in multi-datacenter environments has the advantage that the nodes are more reliable (because there is less churn than in usual P2P networks) and that the environment can be trusted (no Byzantine faults).

To minimize the administration cost, all data and services of multiple applications should be hosted on a single overlay and the overlay should be able to support individual placement policies per application.

Part of this work was carried out under the SELFMAN and XtreemOS projects funded by the European Commission.

II. Assigning Data Ranges to Datacenters

In the following, we present a framework for a key-value store spanning over multiple datacenters. We start from a DHT which does not hash keys but stores them in lexicographical order on the ring. Examples are skip graphs [1], Mercury [2], and Chord$^{\#}$ [14]. We analyze how this property can be exploited for the deployment in datacenter environments.

A. Replication

DHTs typically replicate data items by successor list replication [19] or by key-based replication. The former stores copies of the items in the successors of the responsible node and the latter stores them under multiple keys. All replicas of an item can be found with a parallel lookup. Symmetric replication [7] is a special case of key-based replication where the keys are evenly distributed over the key space.

We introduce *prefix replication* for DHTs that store their items in lexicographical order [14]. It adds a prefix to the key to derive the replicas' keys. As an example, the three replicas of 'gnat' would be stored at keys '1:gnat', '2:gnat', and '3:gnat'.

By adding the application name as a second prefix to the replicas several applications with different replication policies can be hosted on the same overlay. A replicated key name has the form <appname>:<replica_number>:<key>. Thereby, keys of different applications will populate disjoint parts of the ring. Fig. 1 shows an example with four applications, each of them with the replication degree 3.

B. Replica Placement Policies

Replica placement algorithms have to cope with two contradicting goals. On the one hand, the items should be stored near the user to improve the access speed. On the other hand, the items should be spread over several datacenters to guarantee availability in the face of datacenter outages. For read-only data and for weakly consistent data as in Dynamo [6] both aspects can be trivially accomplished by spreading a sufficient number of replicas over all sites.

We aim at providing strong data consistency with concurrent updates. Hence, all read and write operations must be performed on a *majority* of the replicas [16] and it is no longer sufficient to put just one replica near the user, but a majority. The remaining replicas should be spread over the other sites to improve the availability. The differently shaded parts in Fig. 1 illustrates how replicas can be placed in a global ring structure to ensure both, low latency and high availability for applications with different geographical user communities, like several instances of Wikipedia in different languages.

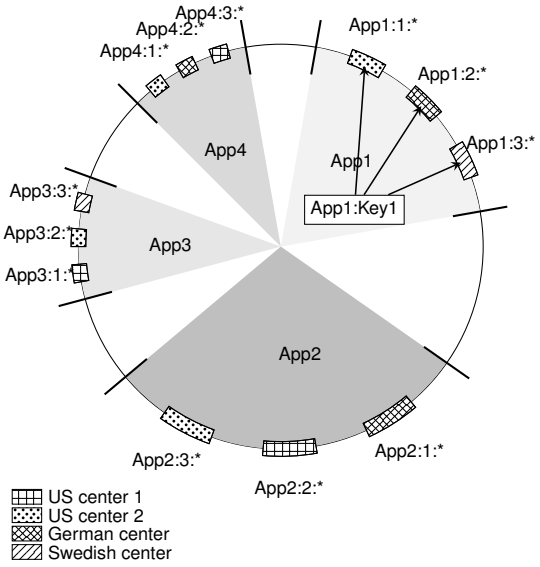


Fig. 1. Prefix Replication

C. Implementation of Placement Policies

The described replica placement policies can be enforced by a standard load-balancing algorithm like that of Karger *et al.* [9]. It autonomously distributes items when nodes join or leave the system. To achieve a suitable geographical placement within datacenter bounds, we modify Karger *et al.*'s algorithm to exchange the workload according to our policy restrictions. More specifically, replicas violating the geographic restrictions are migrated to nodes in the targeted region, while all other load is exchanged within the same geographical region.

Different policies can be enforced for different applications. Policies can be changed at any time and are automatically implemented by the balancing scheme.

III. Latency-aware Routing over Datacenters

When deploying SONs over multiple datacenters, the inter-datacenter latency dominates the overall access latency. Consequently, the use of inter-datacenter links should be minimized, whereas intra-datacenter links can be used more liberally. Proximity routing [3] and k-ary routing [13] are two classical schemes that are used to reduce the access latency in DHTs. Unfortunately, they do not consider clustered nodes in datacenters.

In the following, we first present an agglomerative clustering algorithm for finding nodes in datacenters and then present approaches to reduce the traffic between datacenters.

A. Determining the Location of Datacenters

To be able to reduce the number of long-distance hops, the memberships of nodes in datacenters must be known. For this purpose, we devised an agglomerative clustering algorithm [15] (Alg. 1) that determines the relative node locations by measuring pairwise message latencies. Because latencies cannot be clustered directly, we assign to each node a point in a two-dimensional space so that the Euclidean distance

Algorithm 1 Datacenter detection

```
1: initialize
2:     centroids := {(vivaldi(self()), 1.0)}
3: end

4: every interval time units
5:     peer := selectRandomPeer();
6:     sendto peer : Shuffle(centroids)
7: end event

8: upon event Shuffle(set remoteCentroids) from p
9:     sendto p : ShuffleResponse(centroids)
10:    centroids := update(centroids ∪ remoteCentroids)
11: end event

12: upon event ShuffleResponse(set remoteCentroids)
13:    centroids := update(centroids ∪ remoteCentroids)
14: end event

15: function update(set centroids)
16:    centroids := agglClustering(centroids, radius)
17:    centroids := normalize(centroids)
18:    return centroids
19: end function

20: function normalize(set centroids)
21:    result := ∅
22:    foreach (centroid, size) in centroids do
23:        result := result ∪ {(centroid, size/2)}
24:    end foreach
25:    return result
26: end function
```

to any other nodes reflects the network latency between them. This is done distributedly using a network coordinate system like Vivaldi [5].

Alg. 2 shows the gossip based clustering algorithm that uses Vivaldi. In the beginning, each node assumes that there is only one cluster of size 1 (itself). After several gossiping rounds each node has an estimate of the centroid (average network coordinate of a set of nodes) of each datacenter and their relative sizes. In each gossip step two nodes exchange their current view on the system: both nodes concatenate the two views and recluster them locally.

Once the membership of nodes in clusters (resp. datacenters) is known, the number of long-distance routing hops between datacenters can be minimized. For this purpose, each peer maintains (at least) one finger to a node in each replica range of Fig. 1. Any lookups can then be directly forwarded to the target datacenter without any intermediate hop. Local routing will forward the request to the target node without leaving the destination datacenter – assuming bi-directional routing as described in the next Section.

B. Bi-directional Routing

DHTs typically maintain routing pointers only in one direction. Bi-directional routing, that is maintaining pointers in both directions, does not pay of, because it reduces the hops by only 1 (e.g. from $0.5 \log N$ to $0.5 \log N - 1$) while doubling the storage overhead.

In our datacenter scenario, however, bi-directional routing is beneficial – despite the additional storage overhead. A lookup that is started at the 'end' of the key range hosted in a datacenter for a key that is stored at the 'beginning' of the

Algorithm 2 Datacenter Range detection

```
1: initialize
2:     if getDataCenter(self()) ≠ getDataCenter(succ) then
3:         borders := {(self(), getDataCenter(self()),
4:           getDataCenter(succ))}
5:     else
6:         borders := ∅
7:     end if
8: end

9: every interval time units
10:    peer := selectRandomPeer();
11:    sendto peer : SHUFFLE(borders)
12: end event

13: upon event SHUFFLE(remoteBorders) from p
14:    sendto p : SHUFFLERESPONSE(borders)
15:    borders := borders ∪ remoteBorders
16: end event

17: upon event SHUFFLERESPONSE(remoteBorders)
18:    borders := borders ∪ remoteBorders
19: end event
```

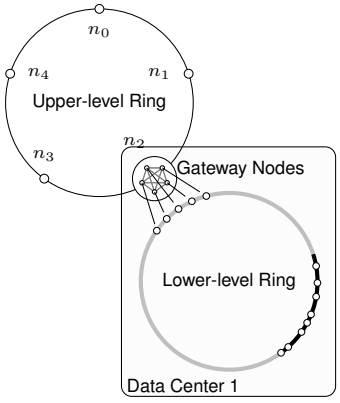


Fig. 2. Hierarchically Structured Overlay for Multi-Datacenter Deployment.

datacenter's range can use a 'backwards' pointer to reach the target key without the need to leave the datacenter. By this means, bi-directional routing avoid inter-datacenter latency by routing (if possible) inside datacenters.

C. Ethernet Broadcasting

Inside a datacenter, multicasts should be performed with a Ethernet broadcast operations. For its subnetwork, each node needs to maintain a finger to all other nodes and their responsibilities. Every node periodically broadcasts its ring identifier and IP-address using Ethernet broadcast messages. With this scheme we achieve a one-hop data access with a slightly larger routing table.

D. Hierarchically Structured Overlay

The traffic between datacenters can also be reduced with a *hierarchically* structured overlay. The architecture presented so far (Fig. 1) provides higher flexibility and better performance than a standard DHT. However, the geographical location of the datacenters is not reflected in the overlay topology and neighboring nodes may be thousands of miles apart.

Moreover, a large amount of TCP connections are kept open between datacenters for successor and finger pointers. This can be avoided by introducing gateway nodes that are connected to the gateway nodes of other datacenters and forward inter-datacenter requests from/to local nodes (Fig. 2).

On the inter-datacenter level, a standard overlay is built where each datacenter appears as *one* peer for each of its replica ranges. This role is performed by the gateway nodes of each datacenter. To avoid overloading of single gateway nodes and to improve fault-tolerance, they can be implemented with replicated state machines which are distributed over multiple machines. Standard load-balancing techniques can be used to hide this fact from other peers.

Even though, the gateway nodes appear to be one replicated peer, only a small subset of their state has to be consistently replicated among them. It is sufficient to consistently replicate [10] the node's position on the ring and the pointer to the successor resp. predecessor. There is no need to keep the routing tables synchronous because they do not affect data consistency. As the gateway nodes do not hold any data, the synchronization overhead is low.

The lower level ring stores the data for which the datacenter is responsible at the upper level. It is divided into an 'active' part (marked bold in Fig. 2) and a 'passive' part. The active part is populated with the data items. It corresponds to the segment of the upper ring for which n_2 is responsible.

The upper level ring is oblivious of the hierarchical structure. This transparency allows to build systems with multi-level hierarchies to even better control the flow of network traffic. In the extreme, one could deploy a global ring on the top, one ring per continent on the next level comprising several datacenters, a ring per container, and a ring per rack on the lowest level.

IV. Coping with Datacenter Failures

In Chord, nodes and keys are randomly hashed on the identifier space, and hence the failure of physically neighbored nodes does not too much affect the nodes in the logical ring. A successor list of length $\log_2 N$ is usually sufficient to repair gaps in the ring – even with a high churn.

With our prefix replication, this assumption is violated. Even worse, neighboring nodes will *likely* be hosted in the same datacenter and their failures will correlate. A network outage in a datacenter can cause thousands of adjacent nodes to disappear from the overlay at the same time. Chord's ring maintenance algorithm cannot fix this, because the ring gap is wider than the successor list length.

A. Gossip-based Ring Maintenance

Our ring maintenance algorithm copes with correlated node failures. It is based on T-MAN [8], a gossip protocol for the construction of arbitrary overlay structures. We adapted T-MAN for continuous ring maintenance as follows.

To accelerate the detection of crashed nodes, each node in the local view is monitored by a failure detector. In case of a failure, the node is removed from the view.

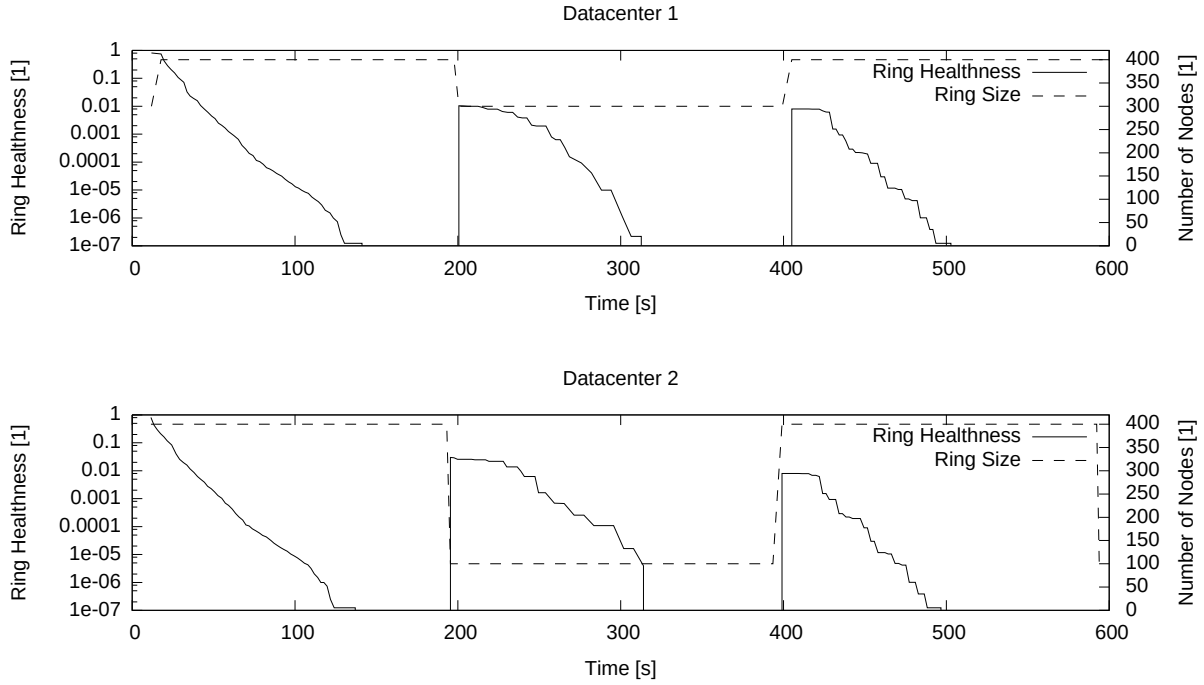


Fig. 3. Network partitioning and repair with modified T-MAN (0s: startup, 200s: network partitioning, 400s: network repair)

Dead nodes are stored in a *dead-node-cache (DNC)*. This is a FIFO queue with a fixed size of 10 elements in our case. DNC nodes are periodically contacted to detect re-appearing nodes, e.g. after repair of a network partitioning.

T-MAN [12] initializes the local view with a set of random nodes. The shuffling continues until the view does not change anymore. When the view becomes stable, the view is re-initialized with random nodes and the procedure starts from the beginning. It takes up to $O(\log N)$ shuffle rounds until defects in the overlay are repaired. In contrast to T-MAN, we never reset the local view and we include in each shuffle operation some random nodes.

T-MAN uses a ranking function based on the distance in the key space, $d(a,b) = \min(N-|a-b|, |a-b|)$, for building rings. Our view, in contrast, is built on the distance in the node space and thereby builds separate predecessor and successor lists, as Fig. 4 shows. This improves the reliability under churn and with correlated failures. On a datacenter outage, the preceeding node of that datacenter will detect that all its successors are gone. In the next T-MAN round, it will accept any random node to fill the missing successor list entries.

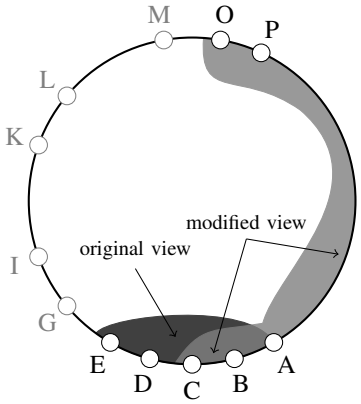


Fig. 4. Succ/pred list with original resp. modified T-MAN.

V. Evaluation

We implemented the described algorithm in our transactional key-value store *Scalaris* [16]. We simulated 400 nodes on two datacenters: 300 in *datacenter 1* and 100 in *datacenter 2*. Each datacenter was simulated on a single server and the network partitioning was simulated by removing the network connection between the two servers. We used a Cyclon Interval of 4.9 seconds, a T-MAN interval of 10 seconds, and a failure detector timeout of 3 seconds.

We simulated the scenario in Fig. 1 with one application. Each datacenter is responsible for a disjoint segment of the ring. When partitioning the network, a contiguous segment of $1/4$ resp. $3/4$ nodes disappears.

Fig. 3 shows the sizes of the datacenters and the healthiness of the ring over the observation time (600s). 'Healthiness' is the aggregated deviation of the local views (predecessors and successors) from the correct view based on global information. Additionally, we weight the errors according to their relative position in the list. It is more important for the direct successor

to be correct than for the last node in the list. Finally, we normalize the error to the interval $[0, 1)$.

The system was started at $t = 0s$ by joining 400 nodes to the system during the first 10 seconds. After ≈ 140 seconds, T-MAN has fixed the ring structure. In this period, T-MAN performed ≈ 14 shuffle rounds.

At $t = 200s$, we disconnected the two servers. For datacenter 1, the ring size drops to 300 nodes, because datacenter 2 is unavailable. Analogously, the ring size for datacenter 2 drops to 100 nodes. At the same time, the error increases, as the predecessor and successor lists of some nodes in the datacenter are invalid. After $\approx 110s$ (at $t = 310s$, after 11 shuffle rounds) the local views became correct again, representing two separate rings, one per datacenter.

At $t = 400s$, we re-connected the links between the two servers and the nodes in the DNC are detected to have become alive again. T-MAN starts to repair the ring. As can be seen, the ring size goes up to 400 nodes and the ring becomes fixed after $\approx 100s$ (at $t = 500s$, after 10 shuffle rounds).

VI. Conclusion

We presented and analyzed techniques for an improved autonomous mapping of structured overlays onto global P2P networks over multiple datacenters. Prefix replication allows to implement fine-grained replica placement policies for improved data availability and reduced lookup latency.

Our system autonomously infers the network topology, detects datacenters and optimizes its routing tables. Our simulation results with 400 nodes showed that with gossip-based ring maintenance SONs can repair themselves also in presence of correlated node failures and network partitioning without global knowledge.

Due to the self-* properties of Scalaris, globally distributed services can be run with low administrative overhead. Adding, removing and updating nodes can be done at any time without preparing or reconfiguring the system. The system will adapt itself accordingly. This eases the job of datacenter operators, reduces the possibility of human errors and allows maintenance without scheduled downtimes.

References

[1] J. Aspnes and G. Shah. Skip graphs. *SODA*, 2003.

[2] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. ACM SIGCOMM 2004.

[3] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. International Workshop on Future Directions in Distributed Computing, 2002.

[4] P. Costa, G. Pierre, A. Reinefeld, T. Schütt, and M. van Steen. Sloppy Management of Structured P2P Services. HotAC, Chicago, June 2, 2008.

[5] F. Dabek, R. Cox, M. F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. ACM SIGCOMM 2004.

[6] 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, 2007.

[7] A. Ghodsi, L. Onana Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. International Workshop on Databases, Information Systems and Peer-to-Peer Computing, 2005.

[8] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. ESOA, 2005.

[9] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. IPTPS, 2004.

[10] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. CACM 21(7), 1978.

[11] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. NSDI, 2005.

[12] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on Demand. P2P 2005.

[13] L. Onana Alima, S. El-Ansary, P. Brand and S. Haridi. DKS(N, k, f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications. CCGRID 2003, 2003.

[14] T. Schütt, F. Schintke, and A. Reinefeld. Structured overlay without consistent hashing: Empirical results. GP2PC'06, 2006.

[15] T. Schütt, A. Reinefeld, F. Schintke, M. Hoffmann. Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers. 9th Int. Conf. on Peer-to-Peer Computing, 2009.

[16] T. Schütt, F. Schintke, A. Reinefeld Scalaris: Reliable Transactional P2P Key/Value Store - Web 2.0 Hosting with Erlang and Java. ACM SIGPLAN Erlang Workshop, 2008.

[17] T. M. Shafaat, A. Ghodsi, S. Haridi. Handling Network Partitions and Mergers in Structured Overlay Networks. P2P 2007.

[18] T. M. Shafaat, M. Moser, T. Schütt, A. Reinefeld, A. Ghodsi, S. Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. Infoscale, 2008.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. ACM SIGCOMM 2001.

[20] S. Voulgaris, D. Gavidia, M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. J. Network Syst. Manage. 13(2), 2005.

A.5 Enhanced Paxos Commit for Transactions on DHTs

Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FLORIAN SCHINTKE, ALEXANDER REINEFELD, SEIF HARIDI, THORSTEN SCHÜTT

Enhanced Paxos Commit for Transactions on DHTs

ZIB-Report 09-28 (September 2009)

Enhanced Paxos Commit for Transactions on DHTs

Florian Schintke*,Alexander Reinefeld*, Seif Haridi† and Thorsten Schütt*
* Zuse Institute Berlin
† Royal Institute of Technology, Sweden

***Abstract*—Key/value stores which are built on structured overlay networks often lack support for atomic transactions and strong data consistency among replicas. This is unfortunate, because consistency guarantees and transactions would allow a wide range of additional application domains to benefit from the inherent scalability and fault-tolerance of DHTs.**

The Scalaris key/value store supports strong data consistency and atomic transactions. It uses an enhanced Paxos Commit protocol with only four communication steps rather than six. This improvement was possible by exploiting information from the replica distribution in the DHT. Scalaris enables implementation of more reliable and scalable infrastructure for collaborative Web services that require strong consistency and atomic changes across multiple items.

I. Introduction

Distributed hash tables (DHTs) and other structured overlay networks (SONs) were developed to provide an efficient key based location of nodes and associated data in the presence of node joins, leaves and crashes (churn). Due to churn, two challenges arise in such systems: (1) When a node crashes, all data stored on this node is lost. (2) When a node is suspected to be crashed, lookup inconsistencies and responsibility inconsistencies may occur, which may lead to wrong query results or loss of update requests. Responsibility inconsistency occurs when multiple nodes believe they are responsible for an overlapping range of items.

The first issue can be addressed by data replication. The second issue can only be relieved but not overcome: It was shown that in an asynchronous network atomic overlay maintenance is impossible [7] and thus responsibility inconsistency is unavoidable. Clearly, data consistency cannot be achieved if responsibility consistency is violated. But as shown in [19], the probability of inconsistent data accesses can be reduced by increasing the replication degree, and performing reads on a majority of replicas. In typical Internet scenarios, for example, only three replicas give a consistency probability of five nines. It can be further improved by adding more replicas or by increasing the share of nodes required for a quorum, but it can never be made 100%[1].

Scalaris [16] is a transactional key/value store which uses symmetric key replication [8] to ensure data availability in the face of churn. Data consistency is enforced by performing all data operations on a majority of replicas.

In this paper, we present improved algorithms for concurrency control and transaction processing, that are based on approaches presented in [14], [16]:

- We show how Paxos Commit can be efficiently embedded into a DHT to perform a low latency non-blocking atomic commit on replicated items. Our commit protocol including the commit phase and the validation phase requires just four message delays in the failure-free case (Sect. V-B).
- We discuss failure scenarios and explain how they are dealt with (Sect. V-B).
- We illustrate how transactions are executed and validated in Scalaris and how concurrency control is performed using readers-writer locks (Sect. V-C and Sect. V-D).
- We evaluate the latency-critical path of our commit protocol by checking each step for its earliest start time (Sect. VI).

Before going into the details in Sect. V and VI, we discuss related work in the following, describe our general overlay structure and replication scheme in Sect. III and provide the fundamentals of Paxos Consensus and Paxos Commit in Sect. IV.

II. Related Work

There are several production systems that use Paxos Consensus [12], like Google's distributed lock service Chubby [3]. The closest to our work is Etna [13] which provides replicated atomic registers. Etna uses consensus to agree on the replica membership set. It does not provide transactional semantics on multiple data items.

[1] Inconsistencies might still happen if multiple nodes join between two existing nodes [19].

Dynamo [5] is a large-scale key/value store. In contrast to Scalaris [16], Dynamo favours availability instead of strong consistency. It provides eventual consistency and no transactions.

We describe an improved transaction commit protocol which reduces the number of message delays in the failure-free case by two compared to our previous protocol [14].

III. Scalaris: Replicated Data on Structured Overlays

Scalaris [16] is a distributed, transactional key/value store with replicated items. It uses symmetric data replication [8] on top of a structured overlay like Chord [20] or Chord$^{\#}$ [17]. In contrast to many other key/value stores, Scalaris provides strong data consistency. It uses the same transaction mechanism for providing replica synchronization as well as transactional semantics on multiple data items.

In the following, we describe the DHT layer and replication layer.

A. Structured Overlay Networks

Distributed hash tables (DHTs) provide a scalable means for storing and retrieving data items in decentralized systems. They are usually implemented on top of structured overlay networks which provide robustness in dynamic environments with unreliable hosts. A DHT has a simple API for storing, retrieving and deleting key/value pairs: *put(key,value), get(key)*, and *delete(key)*.

We use the structured overlay protocol Chord$^{\#}$ [17] for storing and retrieving key/value pairs in nodes that are arranged in a virtual ring. This ring defines a key space where all values can be stored according to the associated key. Nodes can be placed at arbitrary places on the ring and are responsible for all data between their predecessor and themselves. The placement policy ensures even distribution of load over the nodes.

In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to other DHTs like Chord [20], Kademlia and Pastry, Chord$^{\#}$ stores the keys in lexicographical order. This enables range queries and it gives control over the placement of data on the ring structure, which is necessary when deploying a Chord$^{\#}$ ring over datacenters to have better control over latencies. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way [17] that successive fingers in the routing table jump over an exponentially increasing number of nodes in the ring.

To access the node responsible for a given key k, a *DHT lookup* with an average of $0.5\log_b N$ routing hops is performed. The base b can be chosen according to the application requirements, e.g. faster lookup versus lower space requirements [1].

Due to churn, nodes can join and leave at any time, and the ring must be repaired. Stabilization routines run periodically, check the ring healthiness and repair the routing tables according to the finger placement algorithm. If the ring becomes partitioned, a bad pointer list keeps information on nodes on the other part of the ring and a merge algorithm [18], [11] can be used to rejoin them again.

B. Data Replication

To prevent loss of data in the case of failing nodes, the key/value pairs are replicated over r nodes. Several schemes like successor list replication or symmetric replication [8] exist. Symmetric replication stores each item under r keys. A globally known function places the keys $\{k_1,\ldots,k_r\}$ symmetrically in the key space. Read and write operations are performed on a majority of replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes. This scheme is shown to ensure key consistency for data lookups under realistic networking conditions [19].

IV. Paxos Consensus and Paxos Commit

To provide strong consistency over all replicas, transactions are implemented on top of our structured overlay where symmetric replication is employed. We use optimistic concurrency control with a backward validation scheme. Our Scalaris system uses an adapted Paxos Commit for non-blocking atomic commit, which in turn uses Paxos Consensus for each individual data replica to fault-tolerantly agree on prepared or abort for each replica.

We first describe the Paxos Consensus protocol and then discuss the non-blocking atomic commit protocol.

A. Paxos Consensus

In a distributed consensus protocol, all correct (i.e. non-failing) processes eventually choose a single value from a set of proposed values. A process may perform many communication operations during the protocol execution, but it must eventually decide a value by passing it to the client process that invoked the consensus protocol.

Throughout this paper, we assume a fail-stop model where failing processes do not recover. To simulate this behaviour, returning nodes will rejoin with a new identity and empty state.

Algorithm 1 Paxos Consensus: Proposer

1: **initialize**
2: $r =$ any round number greater than all r seen before
3: multicast *prepare(r)* to all acceptors
4: ack_received = $\emptyset$
5: **on receipt of** $ack(r, v_i, rlast_i)$ from acceptor acc_i
6: ack_received = ack_received $\cup$ $(r, v_i, rlast_i)$
7: **if** $|ack\_received| > \frac{n}{2}$ ▷ get index of newest round
8: j = max($rlast_k$: for all k such that $\{r, v_k, rlast_k\} \in$ ack_received)
9: ▷ end of information gathering phase
10: if $v_j = \perp$ ▷ no value agreed yet?
11: v_j = any_value ▷ we propose a value
12: multicast *accept(r, v_j)* to all acceptors

Algorithm 2 Paxos Consensus: Acceptor

1: **initialize**
2: $r_{ack} = 0, r_{accepted} = 0, v = \perp$ ▷ no round acknowledged or accepted yet, no value
3: **on receipt of** *prepare(r)* from *proposer*
4: if $r > r_{ack} \wedge r > r_{accepted}$ ▷ new round?
5: $r_{ack} = r$ ▷ memorize that we saw round r
6: send *ack(r, v, $r_{accepted}$)* to *proposer*
7: **on receipt of** *accept(r, w)* from *proposer*
8: if $r \geq r_{ack} \wedge r > r_{accepted}$ ▷ latest round?
9: $r_{accepted} = r$ ▷ memorize that we accepted in round r
10: $v = w$
11: send *accepted($r_{accepted}$, v)* to *learners*
12: **on receipt of** *decided(v)* from *learner*
13: cleanup()

Algorithm 3 Paxos Consensus: Learner

1: **on receipt of** *accepted(r,v)* from a majority of acceptors
2: multicast *decided(v)* ▷ v is consensus

Lamport's *Paxos Consensus* [12], [15] is a non-blocking consensus protocol for asynchronous distributed systems. Alternative algorithms were proposed by Chandra and Toueg [4] and by Dwork [6]. Paxos implements a *uniform consensus* which achieves agreement even when a minority of processes should fail. Uniform consensus has the following properties [10]:

- *Termination:* Every correct process eventually decides some value.
- *Validity:* If a process decides v, then v was proposed by some process.
- *Integrity:* No process decides twice.
- *Agreement:* No two processes decide differently.

1) Outline of the algorithm: Each process may take the role of a *proposer*, an *acceptor*, or a *learner*, or any combination thereof. A proposer attempts to get a consensus on a value. This value is either its own proposal or the resulting value of a previously achieved consensus. The acceptors altogether act as a collective memory on the consensus status achieved so far. The number of acceptors must be known in advance and must not increase during runtime, as it defines the size of the majority set m required to be able to achieve consensus. The decision, whether a consensus is reached, is announced by a learner.

Proposers trigger the protocol by initiating a new *round*. Acceptors react on requests from proposers. By holding the current state of accepted proposals, the acceptors collectively provide a distributed, fault-tolerant memory for the consensus. In essence, a majority of acceptors together 'know' whether an agreement is already achieved, while the proposers are necessary to trigger the consensus process and to 'read' the distributed memory.

Each round is marked by a distinct round number r. Round numbers are used as a mean of decentralized tokens. The protocol does not limit the number of concurrent proposers: There may be multiple proposers at the same time with different round numbers r. The proposer with the highest r holds the token for achieving consensus. Only messages with the highest round number ever seen by each acceptor, will be processed by that acceptor. All others will be ignored. If at any round, a majority of the acceptors accepted a proposal with value v, it will again be chosen by all subsequent rounds. This ensures the *validity* and *integrity* properties.

Alg. 1, 2, and 3 depict the protocols of the proposer, acceptor, and learner, respectively. The algorithm can be split into two phases: (1) an information gathering phase to check whether there was already an agreement in previous rounds, and (2) a consolidation phase to distribute the consensus to a majority of acceptors and thereby to agree on the decision. In the best case, consensus may be achieved in a single round. In the worst case, the decision may be arbitrarily long delayed by interleaving proposers with successively increasing round numbers (token stealing by each other).

2) Information gathering phase: A proposer starts a new round (lines 1–3 of Alg. 1) by selecting a round number r greater than any round number seen before. At start time, an arbitrary round number is chosen. The only restriction on round numbers is that they must be unique across all possible proposers. This can be achieved, for example, by appending the proposer's identifier. If any

new round number happens to be smaller than an earlier one, the round will be detected as outdated and will be ignored.

The proposer sends its round number with a *prepare(r)* message to the acceptors and starts a timeout (timeouts are not shown in the algorithms). If it does not get an *ack* message from a majority of the acceptors within the timeout, it starts from the beginning with a higher round number and retries with a slightly increased timeout. The timeout implements an eventually perfect failure detector $\diamond\mathcal{P}$ on an arbitrary majority of acceptors.

When an acceptor receives a *prepare(r)* message (lines 3–6 of Alg. 2), it checks whether the given round r is newer than any previously seen round. If the received r is greater, the acceptor memorizes the round and acknowledges with *ack(r, v,* $r_{accepted}$*)* where v is the value accepted previously in round $r_{accepted}$.

Note that a proposed value v may be accepted several times by an acceptor in different rounds. If the round number r is outdated, the acceptor does nothing. Alternatively, the acceptor may send *nack(r,* $r_{accepted}$*)* to help the proposer to quickly find a higher number for a new round (this improvement is not shown in the algorithms).

3) Consolidation phase: After collecting a majority of *ack* messages, the proposer checks for the latest value that was accepted by an acceptor (lines 4–9 of Alg. 1). If it is still the initial $\perp$, the proposer chooses a value by itself, otherwise it takes the latest accepted value v_j. The proposer then sends an *accept(r,* v_j*)* request to the acceptors.

An acceptor receiving an *accept(r,* v_j*)* request checks the round. If it is the latest one, it updates its local state and confirms the accept request with *accepted(r, v)* to the learners (lines 7–11 of Alg. 2). Otherwise the acceptor does nothing or sends *naccepted()* to the proposer.

When a learner receives *accepted(r, v)* messages from a majority of the acceptors, the consensus is finished with value v.

4) Discussion: When a proposer crashes, any other process (or even multiple processes) may take the role of a proposer. The new proposer(s) may retrieve the so far achieved consensus (if any) from the acceptors by triggering a new round.

Since the acceptors have no indication on whether a consensus has been achieved already, they must run forever, always being prepared to take new *accept(r,w)* messages from other proposers. When a new *accept(r,w)* with a higher round number r comes in, they are obliged to accept and store the new value w. As an improvement, the application may decide that a consensus was achieved and consumed and hence the acceptors may be terminated.

B. Paxos Commit

Gray and Lamport [9] describe a commit protocol based on Paxos Consensus. Instead of using a simple version with a single Paxos Consensus as a stable storage, they propose a variant that needs more messages but one less message delay. It performs a Paxos Consensus for each item (TP) involved in the transaction.

In the simple variant, the transaction manager (TM) is responsible to make the decision. It works as follows: The TM asks all TPs whether they are prepared to commit the requested transaction and TPs answer with either prepared or abort. If all TPs are prepared, the TM initiates a Paxos Consensus and takes the role of a proposer by sending *accept(prepared)* to the acceptors, otherwise by sending *accept(abort)*. The acceptors answer *accepted* and on a majority of such answers the TM sends the final decision (commit or abort) to all TPs for execution. This procedure involves 5 message delays.

The Paxos Commit proposed in [9] needs one fewer message delay. It does so with a separate Paxos Consensus instance for each TP. As before, the TM asks all TPs whether they are prepared to commit the requested transaction. This time, however, the TPs do not reply to the TM directly, but initiate a Paxos Consensus for their decision by taking the role of a proposer and sending their proposal *accept(prepared)* or *accept(abort)* to the acceptors for stable storage. After consensus is achieved, they reply with the outcome to the TM in its role as a learner, which then combines the results and sends the final decision to all TPs for execution. This requires 4 message delays and $N(2F+3)-1$ messages for N TPs, and $2F+1$ acceptors.

If the TM or a TP fails in the decision process, any replicated transaction manager (RTM) may read the decision from the acceptors, or propose to abort if there was no consensus yet.

V. Transactions in Scalaris

Scalaris supports transactional semantics. A client connected to the system can issue a sequence of operations including reads and writes within a transactional context, i.e. *begin trans ... end trans*. This sequence of operations is executed by a local transaction manager TM associated with the overlay node to which the client is connected. The transaction will appear to be executed atomically if successful, or not executed at all if the transaction aborts.

A. System Architecture

Transactions in Scalaris are executed optimistically. This implies that each transaction is executed completely locally at the client in a read-phase. If the read phase is successful the TM tries to commit the transaction permanently in a commit phase, and permanently stores the modified data at the responsible overlay nodes. Concurrency control is performed as part of this latter phase. A transaction t will abort only if: (1) other transactions hold the majority of locks of some overlapping data items (simultaneous validation); or (2) other successful transactions have already modified data that is accessed in transaction t (version conflict).

Each item is assigned a version number. Read/write operations work on a majority of replicas to obtain the highest version number and thereby the latest value. A read operation selects the data value with highest version number, and a write operation increments the highest version number of the item.

The commit phase employs an adapted version of the Paxos atomic commit protocol [9], which is non-blocking. In contrast to the 3-Phase-Commit protocol used in distributed database systems, the Paxos Commit protocol still works in the majority part of a network that became partitioned due to some network failure. It employs a group of replicated transaction managers (RTMs) rather than a single transaction manager. Together they form a set of acceptors with the TM acting as the leader.

B. Transaction Validation with Paxos Commit

Scalaris executes the following four steps in the failure-free case (Fig. 1).

1) Prerequisites: For a fast transaction validation, each node in the overlay permanently maintains a list of $r-1$ other nodes, that can be used as *Replicated Transaction Managers (RTMs)*. The location of these nodes could be according to the scheme of symmetric replication. Once these nodes are located, they are maintained through the use of failure detection.

Step 1. The client contacts an arbitrary node in the Scalaris ring with a transaction log (*translog*) of read and write operations for the validation phase. This node becomes the *Transaction Manager (TM)*. The TM chooses a *transaction identifier (Tid)* and a *Paxos Consensus identifier (P_i)* for each replica of each item. It sends an *init_RTM* message with the translog, the Tid, all P_i, and the addresses of all RTMs to each RTM. Additionally, the TM sends to all *Transaction Participants (TP)* an *init_TP* message with the translog, Tid, RTMs, and the individual P_i for each TP.

Step 2. Each TP initiates a Fast Paxos Consensus with the received P_i. Each TP proposes either *prepared* or *abort* with an *accept* message to the acceptors according to its local validation strategy (see later).
As the TP is the only initial proposer, it uses the lowest round number by default and thereby skips the information gathering phase ('Fast Paxos Consensus'). The proposal is sent to the TM and RTMs.
If the TP decided *prepared* it locks its replica. When a TM or RTM receives an accept message from a TP, it also learns the address of the TP to be used later in the protocol.

Step 3. The TM will take the role of a learner in each consensus instance. To allow the TM to calculate each consensus instance, each RTM sends a list of *accepted* messages to the TM. As soon as the TM received a majority of *accepted* messages for a given consensus instance P_i it decides on i.

Step 4. The TM will decide the transaction to *commit* if for each item a majority of the consensus instances have decided prepared, otherwise it will decide *abort*. After having received the decision from the TM, the TPs execute the changes, release the locks and finish.

2) Discussion: As a precondition, we assume that a majority of RTMs plus TM and a majority of replicas for each item are correct. The following failures may happen:

When the TM fails, any RTM may take its role by initiating a new round for every Paxos Consensus involved. In Scalaris, the RTMs' failure detectors have different timeouts, so that multiple RTMs will never compete for leadership and no explicit leader election algorithm is necessary. The new TM is able to continue with the protocol, because the current status on the consensus is safely stored at the RTMs (acceptors).

When an RTM fails, the protocol continues with the rest of the RTMs.

When a TP fails in step 2, the TM or some RTM does not receive an *accept* message from the TP within the specified timeout. The TM or RTM then takes the role of a proposer and proposes *abort* for the corresponding consensus instance with a round number > 1, if no consensus was already achieved before the TP crashed. Until only a minority of the Paxos Consensus for the

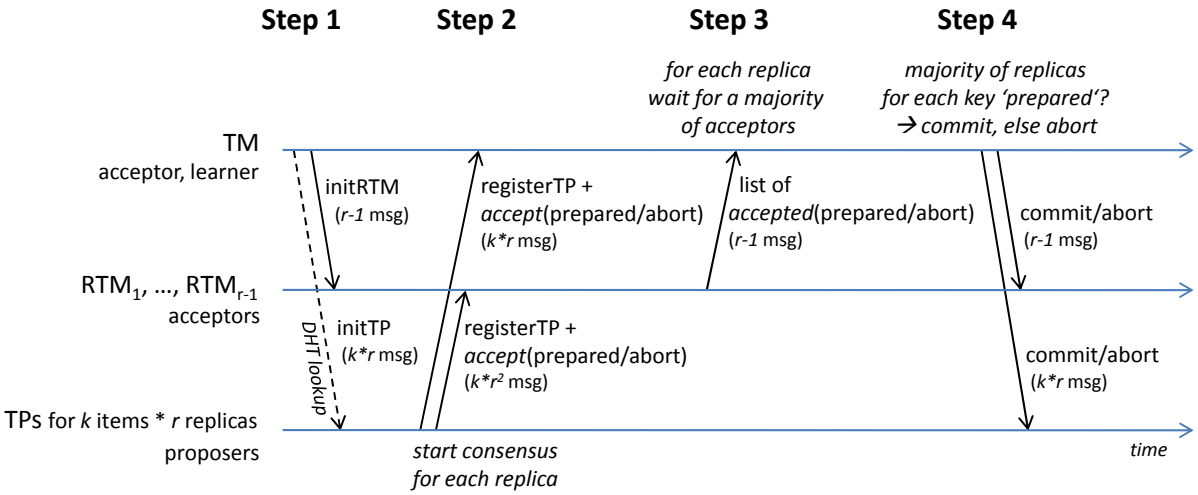


Fig. 1. Timeline diagram of a Scalaris commit.

replicas of a given item votes *abort* and a majority of them votes *prepared* the transaction still can be committed. This can be safely done, as in contrast to Paxos Commit, we operate on replicated items.

C. Working Phase: Building a Translog in Scalaris

We now describe the working phase in which Scalaris builds a translog with all items that are to be updated in an atomic operation. Alg. 4 shows an example of a client code for a money transfer from bank account A to account B. The money transfer should be executed atomically—if the balance in account A allows to. In the example, each account is replicated over three keys $key_{A_1}, \ldots, key_{A_3}$ and $key_{B_1}, \ldots, key_{B_3}$. Fig. 2 shows the corresponding Scalaris ring with the replicas.

The client code shown in Alg. 4 is formulated in the functional programming language Erlang [2]. It works as follows. First, it defines a function F, that will perform the working phase of the transaction (lines 2-12). It then executes this function to retrieve a transaction log (line 13) and thereafter attempts to validate it by calling scalaris:commit() on the outcome of the working phase (line 14).

The working phase is 'read only' and does not modify any values or locks. It stores only the relevant data for each accessed key in the transaction log *translog*. Each translog entry is a 5-tuple consisting of: (1) the performed operation, (2) the key involved, (3) a status flag indicating success or failure, (4) the corresponding value, and (5) the corresponding version.

A *read* request for a key k triggers a quorum read on the replicas, if k is not yet included in the translog.

Algorithm 4 Example of a Scalaris transaction in Erlang.

```
1:  my_transaction() ->
2:    F = fun (TransLog) ->
3:      {X, TL1} = scalaris:read(TransLog, "Acc A"),
4:      {Y, TL2} = scalaris:read(TL1, "Acc B"),
5:      if X > 100 ->
6:        TL3 = scalaris:write(TL2, "Acc A", X - 100),
7:        TL4 = scalaris:write(TL3, "Acc B", Y + 100),
8:        {ok, TL4};
9:      true ->
10:       {ok, TL2};
11:     end
12:   end,
13:   MyTransLog = F(EmptyTransLog),
14:   Result = scalaris:commit(MyTransLog) .
```

It returns the read value and the accordingly updated translog as a tuple.

A *write* request for a key k first triggers a quorum read on the replicas, if k is not yet included in the transaction log. Then a new translog entry with the incremented version number and the new value is created or updated accordingly.

The quorum reads for read and write operations require DHT lookups with $O(\log n)$ hops. If a quorum read fails, this is recorded in the corresponding status flag in the translog. If any status flag in the translog is *failed*, the whole transaction will be aborted.

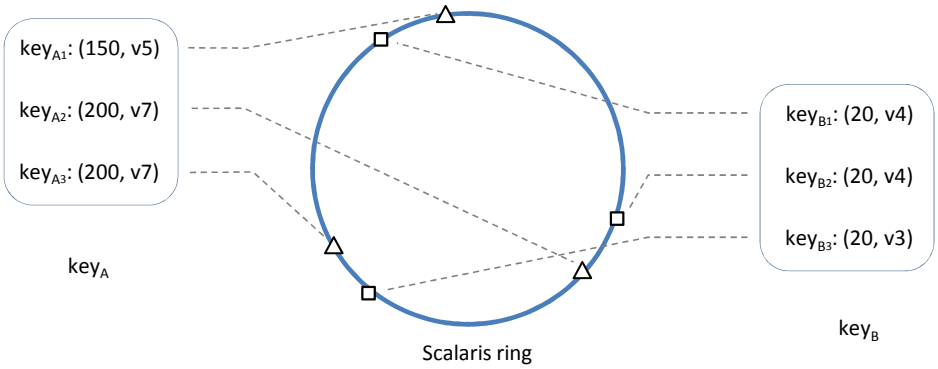


Fig. 2. Scalaris ring with two items key$_A$ and key$_B$.

D. Using the Translog in the Validation Phase

Based on the commit protocol presented in Sec. V-B we now describe the validation strategy in more detail. We show how Scalaris places locks and decides according to the translog.

A TP receives in step 1 of Fig. 1 the corresponding translog entry. To choose between the proposals *prepared* and *abort* it checks the following constraints:

- Is the version number still valid?
  For reads: Is the local version number in the data store the same as the one listed in the translog entry?
  For writes: Is the local version number in the data store one less than the version number stored in the translog?
- Is the lock of the key available?
  For reads: Is no write lock set?
  For writes: Is neither a read lock, nor a write lock set?

If both checks are successful, the TP proposes *prepared* and increments for reads the read lock counter and for writes it sets the write lock. Otherwise it proposes *abort*.

When a TP receives a write commit in step 4 of Fig. 1, it writes the value and version number from the translog into the key.

For read and write operations, independent of commit or abort, the TP releases the locks.

VI. Evaluation

For globally distributed structured overlay systems, latency is an important issue. To reduce the latency in our majority based system, we may assign a majority of the replicas of an item to nodes near the main popularity of that item. This is possible using Chord$^{\#}$ [17] as an overlay, as it allows to arbitrarily assign nodes to ranges of keys and as it does not use hashing but keeps the keys in lexicographical order in the ring.

1) The latency-critical path: In step 1 of our commit protocol, an initialization message is send to each RTM and TP (see Fig. 1). Each TP immediately responds with its *accept* message to the TM and RTMs. So, some *accept* message may arrive at an RTM earlier than the corresponding initialization message. This is not a problem, as the RTM will record it and assign it later via the given transaction and consensus identifiers. Similarly in the case of *accepted* messages from RTMs (step 3) that may arrive earlier at the TM than the *accept* messages from the TPs sent in step 2.

In step 3, each RTM collects an *accept* message for each consensus (each TP) and sends a list of *accepted* in a single message to the TP. While this protocol is optimal with respect to the number of messages sent, the overall latency can be reduced by sending each *accepted* message immediately after receipt of the corresponding *accept*. Then the TM must await a consensus for a majority of the P_i for each item, independent from which RTMs it came. Progress between step 2 and 4 depends on the m lowest latency paths from TPs (via RTMs) to the TM for each item, where $m = r/2+1$ is the size of the majority set.

2) Empirical Results: We compared the performance of simple quorums reads with full transactions on an Intel cluster with 16 nodes. Each node has two Dual-Core Intel Xeons (4 cores in total) running at 2.66 GHz and 8 GB of main memory. The nodes are connected via GigE. On each server we ran s Scalaris nodes distributed over v Erlang virtual machine. We used a replication degree of four, i.e. there are four copies of each key-value pair. For generating load, we started c clients in each Erlang VM and each client performed the function under test i times. We ran the tests with various combinations

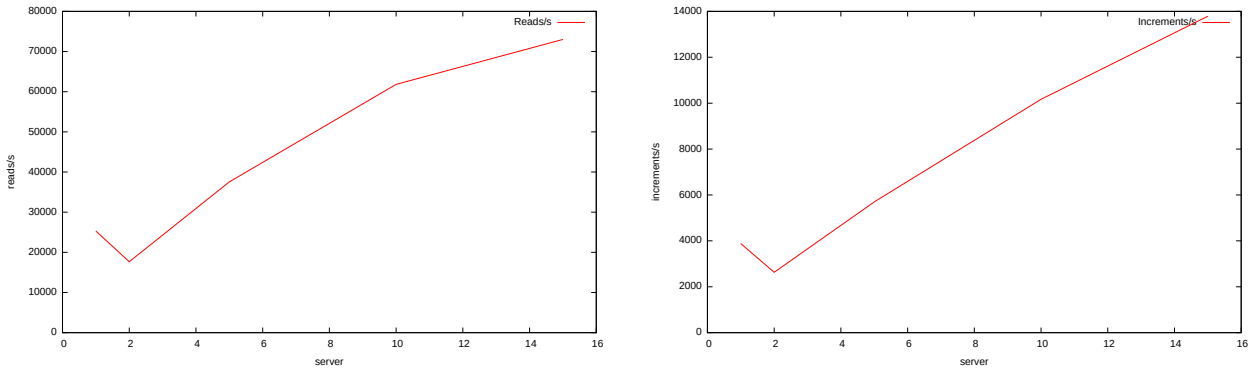


Fig. 3. Performance of quorum reads (left) and transactions with Paxos (right).

for (s,v,c,i). The graphs in Fig. 3 show the aggregated performance over all clients and the number of clients per VM of the best parameter combinations. The best parameter settings usually used 1 VM per server with 16 or 32 Scalaris nodes.

The left graph in Fig. 3 shows the throughput for quorum reads. The maximum of 73,000 lookups is achieved with 15 servers. As the quorum reads are dominated by the lookup, which scales with $\log N$, the curve does not scale linearly. Two servers achieve a lower read performance than one because of the additional TCP overhead.

The right graph in Fig. 3 shows the performance of read-modify-write transactions with Paxos. 15 servers are capable of handling almost 14,000 transactions per second. More importantly, the curve scales almost linearly with an increasing number of servers.

VII. Conclusion

We presented an atomic transaction protocol that has been efficiently embedded into a DHT and uses four communication steps only. It makes progress as long as a majority of TPs for each item and a majority of RTMs (including the TM) are correct (non-failing).

The transaction protocol was used to implement Scalaris [16], a fault-tolerant key/value store with replicated items on a DHT. The DHT ensures scalability while the enhanced Paxos commit protocol provides data consistency. The implementation comprises a total of 9,700 lines of Erlang code: 7,000 for the P2P layer with replication and basic system infrastructure and 2,700 lines for the transaction layer.

Acknowledgements

This work would not have been possible without the great help of the Scalaris team. Funding was provided by the EU projects SELFMAN and XtreemOS.

References

[1] L. Alima, S. El-Ansary, P. Brand and S. Haridi. DKS(N,k,f): A family of low-communication, scalable and fault-tolerant infrastructures for P2P applications. *Workshop on Global and P2P Computing*, CCGRID 2003, May 2003.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006.

[4] T.D. Chandra, S. Toueg. Unreliable failure detector for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: Amazon's highly available key-value store. *SOSP*, Oct. 2007.

[6] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[7] A. Ghodsi. Distributed k-ary system: Algorithms for distributed hash tables. *PhD Thesis*, Royal Institute of Technology, 2006.

[8] A. Ghodsi, L. Alima, S. Haridi. Symmetric replication for structured Peer-to-Peer systems. *DBISP2P*, Aug. 2005.

[9] J. Gray, L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[10] R. Guerraoui, L. Rodrigues. Introduction to reliable distributed programming. Springer-Verlag, 2006.
[11] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. ESOA, 2005.
[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
[13] A. Muthitacharoen, S. Gilbert, R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, 2005.
[14] M. Moser, S. Haridi. Atomic commitment in transactional DHTs. 1st CoreGRID Symposium, Aug. 2007.
[15] R. D. Prisco, B. W. Lampson, N. A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.
[16] T. Schütt, F. Schintke, A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. *ACM SIGPLAN Erlang Workshop*. 2008.
[17] T. Schütt, F. Schintke, A. Reinefeld. Structured overlay without consistent hashing: Empirical results. *GP2PC'06*, May. 2006.
[18] T. M. Shafaat, A. Ghodsi, S. Haridi. Handling Network Partitions and Mergers in Structured Overlay Networks. P2P 2007.
[19] T.M. Shafaat, M. Moser, T. Schütt, A. Reinefeld, A. Ghodsi, S. Haridi. Key-based consistency and availability in structured overlay networks. Infoscale, June 2008.
[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A scalable Peer-to-Peer lookup service for Internet applications. ACM SIGCOMM 2001.

A.6 Towards Explicit Data Placement in Scalable Key/Valaue-stores

Towards Explicit Data Placement in Scalable Key/Value Stores *

Mikael Högqvist
hoegqvist@zib.de

Stefan Plantikow
plantikow@zib.de

Zuse Institute Berlin

Abstract

Distributed key/value-stores are a key component of many large-scale applications. Traditionally they have been designed using Distributed Hash Tables (DHTs). DHTs, however, setup a tight coupling between the naming of nodes and assignment of keys to nodes which limits application control over data placement.

We propose using small amounts of shared state in a semi-centralized architecture for more flexible data placement by introducing explicit mapping between keys and nodes via an indirection layer (blockspace). Our design is based on a membership layer that provides $O(1)$ routing thereby targeting interactive applications. We evaluate a centralized and decentralized approach showing that both have relatively low overhead and provide efficient load balancing.

1 Introduction

Distributed key/value stores [7, 8, 5] provide decentralized, scalable storage as basis for applications like caching layers, indirection services, and activity tracking systems. Such applications require high scalability, high availability, autonomic failure management, and certain consistency and security guarantees from their storage layer.

Beyond this, storage systems powering large-scale web applications and services additionally demand (1) low latency for interactive access [7], (2) fine-grained control over data placement to honor legal restrictions, react to variations in resource pricing, or move/replicate data according to usage patterns [1], and (3) flexible data models with support for both exact-match and range queries.

Current state of the art in autonomic key/value-stores are based on DHTs such as [12, 9]. DHTs provide algorithms for node management (join/leave/fail) and a routing layer that maps keys to the system nodes. Key-lookup requires $O(\log N)$ routing steps and per-node routing table entries.

*This work was funded by the EU FP6 project SELFMAN, IST-34084

DHTs partition the keyspace horizontally by mapping it directly to the node identifier space. This does not require any additional state for finding the location of a key. However, the static assignment makes it difficult to control or introduce new policies for the placement of items within the system. Furthermore, look-up with $O(\log N)$ routing hops may not be sufficient to meet the latency demands of interactive applications [4].

Based on above considerations, we propose a system design that (1) introduces an intermediary blockspace which decouples the keyspace from the nodespace to control data placement and load balancing, (2) uses a routing and membership layer with $O(1)$ hops to provide low lookup latencies. This design implies that each node has to maintain state for the blockspace as well as membership information about all other nodes in the system.

In addition to describing the system design, we contribute algorithms for managing the index-structure storing key/value-pairs. This includes, the assignment of blocks to nodes, block splitting and merging that adapts to the item distribution, and the atomic reassignment of blocks. In the evaluation, we compare two different block assignment strategies in terms of load balancing and maintenance traffic overhead.

2 System Design

Background A key/value store is a database management system for storing key/value-pairs or items. Keys are elements of the *keyspace* range $[K_{min}, K_{max}]$, values are arbitrary binary objects. Items are grouped in *indices* and sorted by key. The store provides a simple interface for creation, retrieval, update and deletion of index items. Data retrieval is achieved by issuing an exact-match query that returns a value for a given key or via range/prefix queries that return all matching key/value-pairs.

Distributed key/value-stores consists of a set of N nodes which are fully connected via a network. Each node stores items in a local database and is assigned an ID in the discrete range $[N_{min}, N_{max})$. This range, the *nodespace*, wraps around at N_{max}, and can be seen as a ring.

Items are assigned to nodes according to some data placement strategy. Many systems implement this through some deterministic and constant function based on keys only, e.g. by hashing into the nodespace. However, such functions can not be directly influenced by the application or system management components. For example, when using hashing, load balancing requires moving nodes by changing their ID. Alternatively, distributing a direct key-to-node assignment to all system nodes does not scale to a large number of keys.

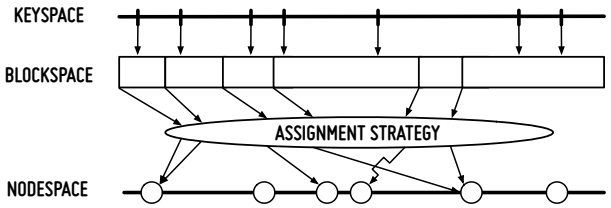


Figure 1. Using an intermediary blockspace enables more flexible data placement

Approach The key insight is that decoupling the nodespace from the keyspace enables the system to control data placement. This separation is achieved by introducing a new indirection layer that we call the *blockspace* (see Fig. 1). The blockspace covers the same identifier space as the keyspace but is partitioned into blocks containing a range of keys. Each block is mapped to the nodespace using an assignment strategy. Distributing the block-to-node assignment to all system nodes is scalable, since the size of the state is several orders of magnitudes smaller compared to a key-to-node assignment. Since we aim for $O(1)$ routing, this mapping needs to be synchronized between all system nodes. This requires a mechanism for membership management and the distribution of global state.

Membership Management Each node needs a consistent view regarding membership information, blockspace partitioning and assignment. We accomplish this by using Census [6], a recently suggested membership service protocol. Census provides each node with a replica of all membership information (shared global state). To achieve this, Census operates in epochs (periods of time). At the start of each epoch, a set of leader nodes broadcasts *changes* to the global state of that epoch to all other nodes. Therefore, during an epoch, all nodes have the same consistent view on the state of the system. At any time, nodes may send updates to leaders for distribution in the next epoch. Leaders agree on the set of updates for the next epoch using a consensus protocol or master-slave replication and leader election.

For broadcasting updates, non-leader nodes are organized as a set of multicast trees. The key to Census is that these trees are constructed deterministically based on the global state of the current epoch, i.e. require no additional communication between nodes. By only propagating updates of the global state, overall communication cost is minimized.

For correctness, census only requires synchronized clock *rates* of participating nodes with an upper bound on the maximal deviation. This is sufficient for local nodes to determine if their current view could still be valid from the leader's point of view.

Indices An index consists of a *blockspace* and a *keyspace*, both confined within a pre-defined discrete range $R = [R_{min}, R_{max})$ (see Fig. 1). The index range is divided into a set of B disjoint blocks. Each block, b_i, has a position i and a start identifier, b_i^{ID}. A block, b_i, ends at the identifier for the next block, b_{i+1}^{ID}. We say that b_i *covers* the range $[b_i^{ID}, b_{i+1}^{ID})$. All blocks combined cover the entire range $[R_{min}, R_{max})$. The first block, b_0, starts at R_{min} and the last block, b_{N-1}, ends at R_{max}.

An item consists of a variable-length key and value. The key identifies the position of the item in the index range. An item with a key outside the range R cannot be stored by the system. A block b_i is responsible for all keys $k \geq b_i^{ID}$ and $k < b_{i+1}^{ID}$. There is always only one such block per key k.

Look-ups The goal of a look-up is to find the node currently responsible for storing an item. In order to perform a look-up, first, the block responsible for the key is determined. Second, the associated node is looked up in the block-to-node assignment table. Both of these indirections are performed locally at each node using the consistent state which contains the block-to-node assignment table. Therefore each look-up requires exactly 1 overlay network hop. A range query, $[a, b]$ is executed by calculating all blocks covering the given range. All blocks are retrieved in parallel by issuing separate look-ups for each block.

Summary Our design uses a combination of a decoupled key- and nodespace with a consistent membership view to allow for flexible data placement and low look-up latency suitable for interactive applications. By introducing a leader-based membership service and efficient application-level multicast, the nodes are provided with low-cost updates to the global state. The leader is not part of read and write operations to the key/value-store and is thereby kept out of the clients fast path. In the following section we introduce the algorithms for managing the blockspace.

3 Index Management

This section presents the algorithms used for managing the blocks in a single index. We describe a self-tuning protocol which adapts the block sizes to a supplied target size in order to support efficient storage of different key distributions and value sizes. Based on this, we introduce two approaches for assigning blocks to nodes. Finally, we outline a protocol for atomic re-assignment of blocks.

Split and Merge The goal of the split and merge algorithm is to dynamically partition the blockspace according to the distribution of the stored keys and the number of stored keys per block. The target size of blocks, L_t, is a system defined parameter. Using a fixed block size is useful for estimating the result set size of a range query. We define two operations used by the algorithm: *split* which divides a single block into two parts and *merge* which merges two consecutive blocks into a single block. Note that the operations described here are binary but they can easily be generalized to n-ary split and merge.

Nodes decide locally if a block that it stores should be split or merged. A split is triggered when a block's load is larger than a factor δ of the target load, assuming $\delta > 1$. A merge is performed when the block load is less than a factor $\frac{1}{\delta}$ of the target load. This ensures that the block load varies in the interval $[\frac{L_t}{\delta}, \delta L_t]$. Using this interval avoids oscillations which can occur in threshold-based schemes. We split the block at the median item to achieve storage load balance but this can be done arbitrarily with application specific policies.

When a node performs an operation on the blockspace, all other nodes must find out about the modification in order for look-ups to be directed to the correct node. A blockspace change is done in two phases. First, a node informs the leader that it wants to perform an update. In a following epoch, the leader forwards the change to all nodes including the initiating node.

Block Assignment The split and merge algorithm partitions the blockspace according to the key distribution. However, for look-ups to work, all nodes need to know the current partitioning including the mapping from the blockspace to the nodespace. The assignment strategy is used to place blocks at different nodes. We present two alternatives, a centralized approach where blocks are explicitly assigned to nodes and a decentralized approach that uses consistent hashing. The main trade-off between the strategies is the extra load and maintenance costs vs. the assignment flexibility.

In the centralized approach, all nodes maintain a data structure containing a mapping from blocks to nodes. *Changes* to the mapping table which are induced by block split and merges as well as node churn are reported to the leader who distributes them to all nodes at the beginning of the next epoch. Thus the mapping tables of all blocks are kept synchronized. A look-up is performed locally on a node by finding the block responsible for a key using the block-node mapping.

In the decentralized approach, the nodes only maintain the current partitioning as created by split and merge. Instead, blocks are assigned to nodes by applying a hash-function mapping their starting identifier, b_i^{ID}, to the nodespace. A block belongs to the node which is closest to the block according to a distance function, $d(x, y)$. Unlike for example Chord [12], we use the euclidean distance $d(a, b) = min((a - b) \bmod ID_{max}, (b - a) \bmod ID_{max})$. This has the advantage that when a node fails, the blocks it was responsible for are divided between its two closest neighbors instead of a single successor.

In both strategies, the state at each node is bounded by the number of blocks in the blockspace. The main difference is the network usage and CPU costs. Since in centralized, the mapping table include both blocks and nodes, it is dependent on the churn rate and the item insert and removal rate. The additional CPU costs comes from the recomputation of the block to node mapping at each epoch. However, performing this at the leader can also save overall CPU costs by not repeating an expensive calculation at each node.

Atomic Block Reassignment When re-assigning a block from one node to another, it is crucial to avoid different nodes disagreeing about who is responsible for a block (at a fixed point in global time). This problem could arise due to delayed message delivery and is related to look-up inconsistency in structured overlay networks [11].

While such structural inconsistency may be dealt with through the use of replication at the cost of higher average latency, as an alternative, we propose a forward-till-timeout-approach to achieve atomic block re-assignment. To handover a block b at epoch e from node n_0 to node n_1, a forward entry is added to the global state at epoch $e + 1$. This forward requires that all requests to block b are sent to n_1 who will forward them to n_0 until n_0 either is removed from the system or acknowledges successful handover to n_1 along with the last version of b. n_0 will signal this only after it is guaranteed that either each system node is in epoch $e + 1$ or must have failed. This knowledge is available to n_0 based on assumptions about epoch duration, leader behavior, accuracy of clock rate synchronization, and the maximal timeout after which a node that has not received the next epoch state will cease processing requests in its current epoch. In any case, eventually n_1 can be sure that no more requests will be sent to n_0 and will finish the re-assignment in the following epoch by removing

the forward entry from the global state. We are currently working on a more detailed and formal description of this protocol.

4 Evaluation

By introducing a leader in the system, it is the most likely bottleneck even though it is only mediating control traffic and not any data traffic. We evaluate the trade-off between maintenance costs and the ability to balance the storage load using the centralized and decentralized approaches. The experiments are performed using a discrete time event-based simulator.

An experiment is initialized with a leader and 1000 nodes. Each node has a mean time to failure (MTTF) and a recovery delay in order to simulate churn. The MTTF is drawn from an exponential distribution with an average set to 1 hour. Block updates are sent to the leader which forwards the changes in each epoch (30s) to all nodes. We use an insert-only workload with a constant rate of 10000 items per epoch and a block size of 10000 items. For simplicity, messages are sent directly to the nodes without loss. Thus, we only measure the overhead of the centralized and decentralized approach without fault tolerance.

Maintenance Costs In this experiment, we measure the maintenance overhead for an index. That is, the updates propagated by the leader that are used by the nodes to maintain their internal block-node mapping table.

Figure 2(a) shows the aggregated number of updates after 86400s (1 day). The significantly higher maintenance cost for centralized block assignment can mainly be attributed to the churn rate, since, unlike decentralized assignment, each join and leave require explicit updates to the block-node mapping table. The average number of updates per epoch for centralized is 26.4, while for decentralized it is 2.3. Note that the rate of splits is much higher in the starting phase. This is because the system starts with a single block and is under a uniform insert-only workload which trigger splits as soon as a block is full. With more blocks in the system, it takes longer time before a block is full.

Block Imbalance Using the centralized approach, the maintenance costs are significantly higher compared to the decentralized approach. However, by letting a leader decide the block to node assignment, we have more control of how to balance the storage load. In this experiment, we quantify this trade-off by measuring the storage imbalance resulting from a central algorithm vs. consistent hashing.

We define the block storage imbalance as $\frac{L_{max}}{L_{avg}}$. L_{max} is the maximum number of blocks stored at any node while L_{avg} is the average number of blocks per node. The central algorithm assigns each unassigned block in an epoch to the nodes with the least number of blocks. A block becomes unassigned when the node responsible for the block leaves the system or when a new block is created through a split.

Figure 2(b) shows that the imbalance for the centralized algorithm approaches 1 with increasing number of blocks, while the hash-based algorithm is able to balance the system within a factor 5 to 10. We conclude that the increased control gained from the centralized algorithm makes it possible to significantly improve the storage imbalance. In addition, the leader-based approach can more easily be extended to consider further parameters such as inter-node latencies or application-specific placement policies.

5 Related Work

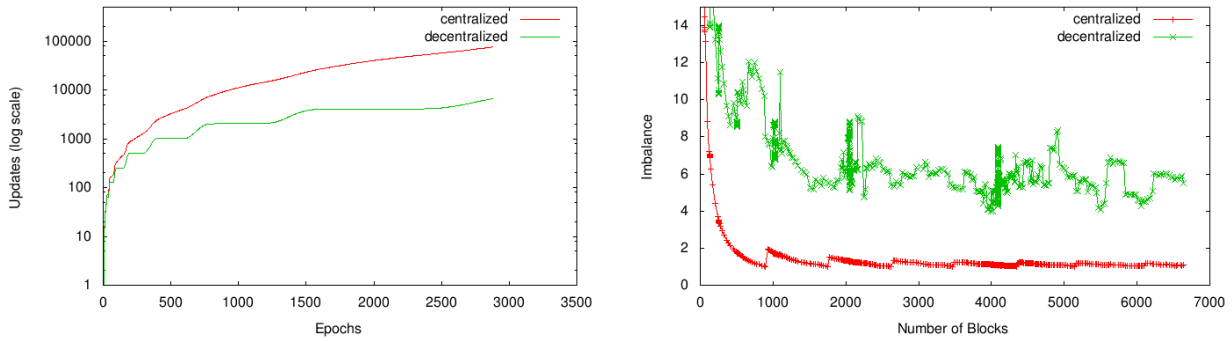
DHTs Classic DHTs such as Chord or Pastry are mapping keys directly to the identifier space. This makes it difficult to support range queries without explicit storage load balancing. Therefore several approaches that layer additional indexing on top of DHTs in order to achieve complex queries such as range- or prefix-queries has been developed. We discuss two of these approaches, however, unlike our system which is layered on a one-hop membership service, they all assume a DHT as underlay. In [13], Zheng et. al. present a binary tree structure called Distributed Segment Tree (DST) that support range- and cover-queries. A DST is a binary tree where a tree node represents a key interval. The DST is mapped to a DHT by hashing the node's interval into the nodespace. The insert-cost in a DST corresponds to the height of the tree since a parent tree node also cover the interval of its children.

In RIPPNET [10], Ryeng et. al., use a CAN-like system to index fragments of a range instead of the individual keys. We are also decoupling the keyspace from the nodespace by introducing fragments or blocks, but do not require a specialized multi-dimensional DHT.

Distributed Key/Value-stores Amazon's Dynamo [7], uses consistent hashing for partitioning to keyspace. Each physical node maintains a set of virtual nodes used for load balancing. Nodes have an eventually consistent view of the members, including the virtual nodes, of the system which is updated through gossiping.

Google [3], uses Chubby [2], a highly available and persistent distributed lock service, to handle node membership. A central master is responsible for assigning tablets, blocks of data, to the nodes.

PNUTS from Yahoo! [5] take a similar approach by storing key/value-pairs in blocks (tablets). Unlike Dynamo, they support exact-match and range queries. The mapping of tablets to data nodes is done with a centralized "tablet controller" for increased control. Routers, used by clients



(a) Maintenance costs

(b) Block storage imbalance

Figure 2. Maintenance costs and storage imbalance with centralized and decentralized.

to find the location of a key or range, update their internal tablet state regularly via the tablet controller.

Both Yahoo! and Google uses a centralized way of assigning blocks to nodes, while Amazon uses consistent hashing. We plan to further explore this trade-off between the cost of centralized control and decentralized assignment for environments which are not under single administrative control.

6 Conclusions

Revisiting semi-centralized architectures is an interesting option for building low-latency large-scale storage systems. We explored this idea through our system design based on the census membership service. Beyond achieving $O(1)$ routing, a consistent global view opens up the possibility for flexible data placement. Our simulation results indicate that the traffic overhead of update dissemination is considerably low. We conclude, that semi-centralized management of data placement is an interesting design approach for distributed key/value-stores.

References

[1] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. M. Hellerstein. A case study in building layered dht applications. In *SIGCOMM*, pages 97–108. ACM, 2005.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[6] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *USENIX*, San Diego, CA, USA, June 2009.

[7] 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. In *SOSP*, pages 205–220. ACM, 2007.

[8] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *SIGCOMM*, pages 73–84. ACM, 2005.

[9] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[10] N. H. Ryeng and K. Nørvåg. Rippnet: Efficient range indexing in peer-to-peer networks. In *ICDIM*, pages 184–191. IEEE, 2008.

[11] T. M. Shafaat, M. Moser, T. Schütt, A. Reinefeld, A. Ghodsi, and S. Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proc. of Infoscale'08*. ACM, June 2008.

[12] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[13] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed Segment Tree: Support of Range Query and Cover Query over DHT. In *IPTPS*, Santa Barbara, USA, Feb. 2006.

A.7 Active/Passive Load Balancing with Informed Node Placement in DHTs

Passive/Active Load Balancing with Informed Node Placement in DHTs

Mikael Högqvist and Nico Kruber

Zuse Institute Berlin
Takustr. 7, 14195, Berlin, Germany
hoegqvist@zib.de, kruber@zib.de

Abstract. Distributed key/value stores are a basic building block for large-scale Internet services. Support for range queries introduces new challenges to load balancing since both the key and workload distribution can be non-uniform.

We build on previous work based on the power of choice to present algorithms suitable for active and passive load balancing that adapt to both the key and workload distribution. The algorithms are evaluated in a simulated environment, focusing on the impact of load balancing on scalability under normal conditions and in an overloaded system.

1 Introduction

Distributed key/value stores [1,2,3] are used in applications which require high throughput, low latency and have a simple data model. Examples of such applications are caching layers and indirection services. Federated key/value-stores, where the nodes are user contributed, require minimal management overhead for the participants. Furthermore, the system must be able to deal with large numbers of nodes which are often unreliable and have varying network bandwidth and storage capacities. We also aim to support both exact-match and range queries to increase flexibility for applications and match the functionality of local key/value-stores such as Berkeley DB and Tokyo Cabinet.

Ring-based Structured Overlay Networks (SONs) provide algorithms for node membership (join/leave/fail) and to find the node responsible for a key within $O(\log N)$ steps, where N is the number of nodes. One of the main advantages of SONs for large-scale services is that each node only has to maintain state of a small number of other nodes, typically $O(\log N)$. Most SONs also define a static partitioning strategy over the data items where each node is responsible for the range of keys from itself to its predecessor.

At first glance SONs may therefore seem to be a good fit for distributed key/value stores. However, the static assignment of data items to nodes in combination with the dynamic nature of user-donated resources make the design of the data storage layer especially challenging in terms of reliability [4] and load balancing.

The goal of load balancing is to improve the fairness regarding storage as well as network and CPU-time usage between the nodes. Imbalance mainly occurs

T. Spyropoulos and K.A. Hummel (Eds.): IWSOS 2009, LNCS 5918, pp. 101–112, 2009.

due to: 1) non-uniform key distribution, 2) skewed access frequency of keys and 3) node heterogeneity. First, by supporting range-queries, an order-preserving hash function is used to map keys to the overlay's identifier space. With a non-uniform key distribution a node can become responsible for an unfair amount of items. Second, keys are typically accessed with different popularity which creates uneven workload on the nodes. The third issue, node capacity differences, also impacts the imbalance. For example, a low capacity node gets overloaded faster than a high capacity node. We assume that nodes are homogeneous or have unit size, where a single physical node can run several overlay nodes.

Our main contribution is a self-adaptive balancing algorithm which is aware of both the key distribution and the item load, i.e. used storage and access-frequency. The algorithm has two modes: *active*, which triggers a node already part of the overlay to balance with other nodes and *passive*, which places a joining node at a position that reduces the overall system imbalance. In both the passive and active mode, a set of nodes are sampled and the algorithm balance using the node with the highest load.

Our target application is a federated URL redirection service. This service allow users to translate a long URL, from for example Google Maps, to a short URL. The redirection service supports look-ups of single URLs as well as statistics gathering and retrieval over time which motivates the need for range queries to execute aggregates. Popular URL redirection providers such as `tinyurl.com` have over 60 million requests per day and close to 300 million indirections.

Section 2 contains the model, assumptions and definitions that are used for the load balancing algorithm presented in Section 3. In Section 4, we evaluate the system using a simulated environment. Results from the simulation show that the algorithm improves the load imbalance within a factor 2-3 in a system with 1000 nodes. In addition, we also show that load balancing reduces the storage capacity overhead necessary in an overloaded system from a factor 10 to 8.

2 System Model

A ring-based DHT consists of N nodes and an identifier space in the range $[0, 1)$. This range wraps around at 1.0 and can be seen as a ring. A node, n_i, at position i has an identifier n_i^{ID} in the ID space. Each node n_i has a *successor*-pointer to the next node in clockwise direction, n_{i+1}, and a *predecessor*-pointer to the first counter-clockwise node, n_{i-1}. The last node, n_{N-1}, has the first node, n_0 as successor. Thus, the nodes and their pointers create a double linked list where the first and last node are linked. We define the distance between two identifiers as $d(x, y) = |y - x| \bmod 1.0$.

Nodes can fail and join the system at any time. When a node joins, it takes over the range from its own ID to the predecessor of its successor. Similarly, when a node n_i fails, its predecessor becomes predecessor of n_i's successor. We model churn by giving each node a mean time to failure (MTTF). To maintain the system size, a failed node is replaced after a recovery time-out.

Storage: When a key/value-pair or item is inserted in the system it is assigned an ID using an order-preserving hash-function in the same range as the node IDs, i.e. $[0, 1)$. Each node in the system stores the subset of items that falls within its responsibility range. That is, a node n_i is *responsible* for a key iff it falls within the node's key range $(n_{i-1}^{ID}, n_i^{ID}]$.

Each item is replicated with a replication factor f. The replicas are assigned replica keys according to symmetric replication where the identifier of an item replica is derived from the key and the replica factor using the formula $r(k, i) = k + (i-1) * \frac{1}{f} \bmod N$, k is the item ID and i is the ith replica [5]. An advantage of symmetric replication is that the replica keys are based on the item key. This makes it possible to look-up any replica by knowing the original key. In other approaches such as successor-list replication [6] the node responsible for the key must first be located in order to find the replicas.

A replica maintenance protocol ensures that a node stores the items and the respective replicas it is responsible for. The protocol consist of two phases; the synchronization phase and the data transfer phase. In the synchronization phase, a node determines which items should be stored at the node using the symmetric replication scheme. And if they are not stored or not up-to-date, which replicas need to be retrieved. The retrieval is performed during the data transfer phase by issuing a read for each item.

Load and Capacity: Each node has a workload and a storage capacity. The workload can be defined arbitrarily, but for a key/value-store this is typically the request rate. Each stored item has a workload and a storage cost. A node cannot store more items than its storage capacity allows. The workload, on the other hand, is limited by for example bandwidth, and a node can decide if a request should be ignored or not. We model the probability of a request failure as $P(fail) = 1 - \frac{1}{\mu}$, where μ is the current node utilization, i.e. the measured workload divided by the workload capacity.

Imbalance: We define the system imbalance of a load attribute (storage or workload) as the ratio between the highest loaded node and the system average. For example, for the storage, the imbalance is calculated as $\frac{L_{max}}{L_{avg}}$. L_{max} is the maximum number of items stored by a node and L_{avg} is the average number of items per node.

3 Load Balancing Algorithm

The only way to change the imbalance in our model is to change the responsibility of the nodes. A node's responsibility changes either when another node joins between itself and its predecessor, or when the predecessor fails. Thus, we can balance the system either *actively* by triggering a node to fail and re-join or *passively* by placing a new node at an overloaded node when joining. Passive balancing uses the system churn, while active induces churn and extra data transfers. We first present the passive/active balancing algorithm followed by the placement function.

```
1  def placement():
2      balanced_ID = ⊥
3      current_distance = ∞
4      for item in (n_{i-1}^ID, n_i^ID]:
5          distance = f(item^ID) # the placement function
6          if distance < current_distance:
7              balanced_ID = item^ID + d(item^ID, next(item^ID))/2
8              current_distance = distance
9
10     return balanced_ID
11
12 def sample():
13     samples = [(n.load(), n)
14                for n in random_nodes(k)]
15     return max(samples)
16
17 def passive():
18     (n_load, n) = sample()
19     join(n)
20
21 def active():
22     (n_load, n) = sample()
23     if n_load > local_load * ε:
24         leave()
25         join(n.placement())
```

Fig. 1. Passive and Active load balancing

The passive/active balancing algorithm presented in Figure 1 uses only local knowledge and can be divided into three parts. 1) sample a set of k random nodes to balance with using e.g. [7], 2) decide the placement of a potential new predecessor and 3) select one of the k-nodes that reduce the imbalance the most. We assume that there is a join function which is used to join the overlay given an ID. `passive` is called before a node is joining and `active` is called periodically. `active` is inspired by Karger's [8] balancing algorithm, but we only consider the case where the node has a factor ϵ less load than the remote node. The ϵ is used to avoid oscillations by creating a relative load range where nodes do not trigger a re-join. `sample` calls a function `random_nodes` that uses a random walk or generates random IDs to find a set of k nodes. The node with the highest load is returned.

Placement Function

The goal of the placement function is to find the ID in a node's responsibility range that splits the range in two equal halves considering both workload and key distribution. When defining the cost for a single load attribute, it is optimal to always divide the attribute in half [9]. We use this principle for each attribute by calculating the ratio between the range to the left of the identifier x and the remaining range up to the node's ID. The optimal position is where this ratio approaches 1. A ratio therefore increases slowly from 0 towards 1 until the optimal value of x is reached, and after 1 the value approaches the total cost for the attribute.

First, let $l_r(a,b) = \sum_{i=0}^{items \in (a,b]} l(item_i)$ be a function returning the load of the items in the range $(a,b]$. $l(item_i)$ is the load of a single item and is defined arbitrarily depending on the load attribute. Second, let n_i be the node at which we want to find the best ID, then the ratio function is defined as follows

$$r(x) = \frac{l_r(n_{i-1}^{ID}, x)}{l_r(x, n_i^{ID})}$$

The workload ratio, $r_w(x)$, could for example be defined using $l(item_i) = weight(item_i) + (rate_{access}(item_i) \times weight(item_i))$. The weight is the total bytes of the item and the access rate is estimated with an exponentially weighted moving mean. For the key distribution ratio, $r_{ks}(x)$, the load function is $l(item_i) = 1$. This means that $r_{ks}(x) = 1$ for the median element in n_i's responsibility range. An interesting aspect of the ratio definitions is that they can be weighted in order to ignore load attributes that changes fast or taking on extreme values.

In order to construct a placement function acknowledging different load attributes, we calculate the product of their respective ratio function. The point x where this product is closest to 1 is where all attributes are being balanced equally. Note that when it equals 1, it means that the load attributes have their optimal point at the same ID.

The placement function we use here considers both the key-space and workload distribution and is more formally described as

$$f(x) = |1 - r_w(x) \times r_{ks}(x)|$$

where x is the ID and n_j is the joining node. The ratio product value is subtracted from 1 and the absolute value of this is used since we are interested in the ratio product value "closest" to 1. Finally, when the smallest value of $f(x)$ is found, a node is placed at the ID between the item, $item_i$ preceding x and the subsequent item, $item_{i+1}$. That is, the resulting ID is $item_i^{ID} + d(item_i^{ID}, item_{i+1}^{ID})/2$.

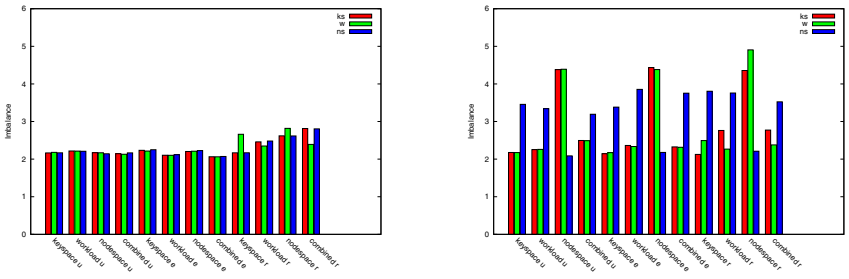
4 Evaluation

This section present simulation results of the passive and active algorithms. The goal of this section is to 1) show the effects of different access-load and key distributions, 2) show the scalability of the balancing strategies when increasing the system size and 3) determine the impact of imbalance in a system close to its capacity limits. Table 1 summarizes the parameters used for the different experiments.

Effect of Workloads: In this experiment, we quantify the effect that different access-loads and key distributions have on the system imbalance. The results from this experiment motivate the use of a multi-attribute placement function. Specifically, we measure the imbalance of the nodespace (ns), keyspace (ks) and the access workload (w).

Table 1. Parameters of the different experiments

	Nodes	Items	Replicas	k	MTTF	Storage	Item Size
Effect of Workloads	256	32768	7	7	∞	∞	1
Network costs	256	8192	7	7	1h	∞	1-1MB
Size of k	256	8192	7	0-20	1h	∞	1
System size	64-1024	2^{15}-2^{18}	3	7	1h	∞	1
Churn	256	8192	7	7	1h-1d	∞	1
Overload	256	8192	7	7	1h	$128 * 7$-$1024 * 7$	1



(a) Uniform key distribution (b) Dictionary key distribution

Fig. 2. The effect of different access workloads and key distributions

Four different placement functions are used (x-axis in Fig. 2)

nodespace places a new node in the middle between the node and its predecessor, i.e. $n_i + \frac{d(n_{i-1},n_i)}{2}$.
keyspace places the node according to the median item, $f(x) = |1 - r_{ks}(x)|$.
workload halves the load of the node, i.e $f(x) = |1 - r_w(x)|$
combined uses the placement function defined in section 3.

The simulation is running an active balancing algorithm with $\epsilon = 0.15$.

Workload is generated using three scenarios; uniform (u), exponential (e) and range (r). In the uniform and exponential cases, the items receive a load from either a uniform or exponential distribution at simulation start-up. The range workload is generated by assigning successive ranges of items with random loads taken from an exponential distribution. We expect this type of workload from the URL redirection service when, for example, summarizing data of a URL for the last week.

From the results shown in Figure 2, we can see that the imbalance when using the different placement strategies are dependent on the load type. Figure 2(a) clearly shows that a uniform hash-function is efficient to balance all three metrics under both uniform and exponential workload. In the latter case, this is because the items are assigned the load independently. However, for the range workload, the imbalances are showing much higher variation depending on the placement

function. We conclude that in a system supporting range queries, the placement function should consider several balancing attributes for fair resource usage.

Size of *k*: In this experiment, we try to find a reasonable value of the number of nodes to sample, k. A larger k implies more messages used for sampling, but also reduces the imbalance more. The results in figure 3 imply that the value of k is important for smaller values of between 2-10. However, the balance improvement becomes smaller and smaller for each increase of k, similar to the law of diminishing returns. In the remaining experiments we use $k = 7$.

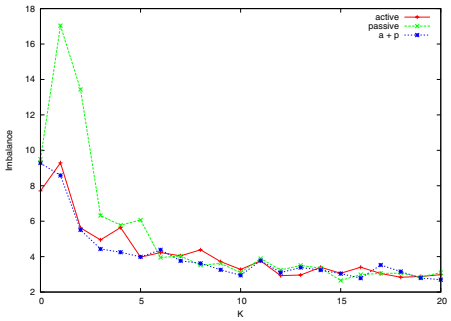
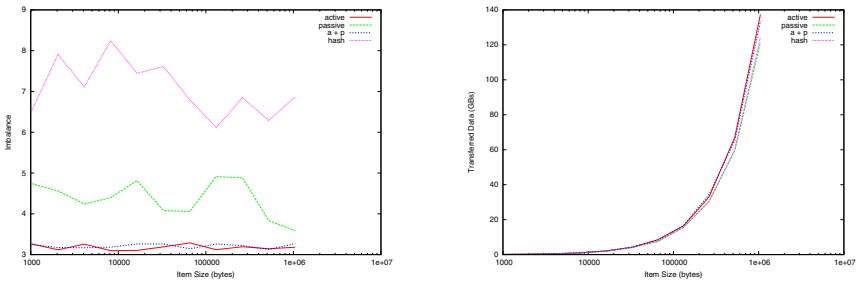


Fig. 3. Imbalance when increasing the number of sampled nodes

Network costs: We define cost as the total amount of data transferred in the system up to a given iteration. This cost is increased by the item size each time an item is transferred. Since there is no application traffic in the simulation environment, the cost is only coming from replica maintenance. That is, item transfers are used to ensure that replicas are stored according to the current node responsibilities. Active load balancing creates traffic when a node decides to leave and re-join the system.

We measure the keyspace imbalance and the transfer cost at the end of the simulation, which is run for 86400s (1 day). Each simulation has 8192 items with 7 replicas and the size of the items is increased from 2^{10} to 2^{20}. The item size has minor impact on the imbalance (Fig. 4(a)). Interestingly, the overhead when using the hash-based balancing strategy as a reference, of active and passive (a+p in the figure) and active only is 5-15% (Fig. 4(b)). The passive strategy does not show a significant difference. Noteworthy is also that in a system storing around 56 GB of total data (including replicas), over 1 TB aggregated data is transferred. This can be explained with the rather short node lifetime of 3600s.

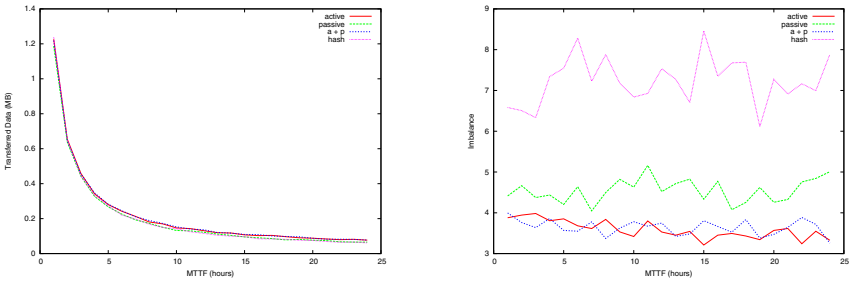
Churn: A node joining and leaving (churn) changes the range of responsibility for a node in the system. Increasing the rate of churn influences the cost of replica maintenance since item repairs are triggered more frequently. In this experiment, we quantify the impact of churn on transferred item cost and the storage imbalance.



(a) Imbalance vs. Item size (b) Transferred bytes

Fig. 4. Imbalance and cost of balancing for increasing item size

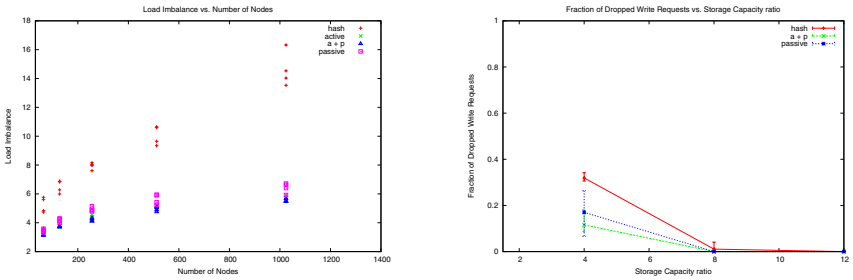
In figure 5(a) the node MTTF is varied from 1 to 24 hours. As expected the amount of data transferred is decreasing when the MTTF is increasing. Also as noted in the network costs experiment, the different schemes for load balancing have a minor impact on the total amount of transferred data. Figure 5(b) shows that churn has in principle no impact on the imbalance for the different strategies. This is also the case for the passive approach which only relies on churn to balance the system.



(a) Bytes transferred with increasing MTTF (b) Imbalance with varying MTTF

Fig. 5. Imbalance and network cost for varying levels of churn (MTTF)

System size: The imbalance in a system with hash-based balancing was shown theoretically to be bounded by $O(\log N)$, where N is the number of nodes in the system [10]. However, this assumes that both the nodes and the keys are assigned IDs from a uniform hash-function. In this experiment, we try to determine the efficiency of the placement function with an increasing number of nodes and items.



(a) Increasing nodes and items (b) Capacity

Fig. 6. Imbalance of the system using different balancing strategies while increasing the system size. The right figure shows the influence of load balancing in an overloaded system.

We measure the keyspace imbalance for an increasing number of nodes between 2^5 and 2^{10}. In addition, for each system size we vary the number of items from 2^{15} to 2^{18}. Keys are generated from a dictionary and nodes are balanced using the combined placement function. Four different balancing strategies are compared; 1) IDs generated by a uniform hash-function 2) active without any passive placement, 3) passive without any active and 4) active and passive together (a+p). For the last three, 7 nodes are sampled when selecting which node to join at or whether to balance at all.

Figure 6(a) shows that the hash-based approach performs significantly worse with an imbalance up to 2-3 times higher compared to the other balancing strategies. Interestingly, the difference in load imbalance when varying the number of items is also growing slightly with larger system sizes. All three variants of the passive/active algorithm show similar performance. The imbalance grows slowly with increasing system size and the difference for different number of items is small. Thus, we draw the conclusion that these strategies are only minimally influenced by system size and number of items. However, note that we need to perform further experiments varying other parameters such as k to validate these results.

Overload: In a perfectly balanced system where at most one consecutive node can fail, nodes can use at most up to 50% of their capacity to avoid becoming overloaded when a predecessor fails. This type of overload leads to dropped write requests when there is insufficient storage capacity and dropped read request with insufficent bandwidth and processing capacity. Since a replica cannot be recreated when a write is dropped, this influences the data reliability. The goal of this experiment is to better understand the storage capacity overhead to avoid dropped writes.

We start the experiment such that the sum of the item weights equals the aggregated storage capacity of all nodes. Then by increasing the node's storage

capacity we decrease their fill-ratio and thereby the probability of a dropped write. The system is under churn and lost replicas are re-created using a replica maintenance algorithm executed periodically at each node. The y-axis in Figure 6(b) shows the fraction of dropped write requests and the x-axis shows the storage capacity ratio. We do not add any data to the system which means that a write request is dropped when a replica cannot be created at the responsible node because of insufficient storage capacity. We measured the difference with hash-based balancing vs. the active and active + passive with 7 sampled nodes and the combined placement function.

Figure 6(b) shows that a system must have at least 10x the storage capacity over the total storage load to avoid dropped write requests when using hash-based balancing. Active and active-passive delays the effect of overload and a system with at least 8x storage capacity exhibits a low fraction of dropped requests.

5 Related Work

Karger et al. [8] and Ganesan et al. [11] both present active algorithms aiming at reducing the imbalance of item load. Karger uses a randomized sampling-based algorithm which balances when the relative load value between two nodes differs by more than a factor ϵ. Ganesan's algorithm triggers a balancing operation when a node's utilization exceeds (falls below) a certain threshold. In that case, balancing is either done with one of its neighbors or the least (most) loaded node found. Aspnes at al. [12] describe an active algorithm that categorizes nodes as closed or open depending on a threshold and groups them in a way so that each closed node has at least one open neighbor. They balance load when an item is to be inserted into a closed node that cannot shed some of its load to an open neighbor without making it closed as well. A rather different approach has been proposed by Charpentier et al. [13] who use mobile agents to gather an estimate of the system's average load and to balance load among the nodes. Those algorithms however do not explicitly define a placement function or use a simple "split loads in half" approach which does not take several load attributes into account.

Byers et. al. [14] proposed to store an item at the k least loaded nodes out of d possible. Similarly, Pitoura et al. [15] replicate an item to k of d possible identifiers when a node storing an item becomes overloaded (in terms of requests). This technique, called the "power of two choices" was picked up by Ledlie et. al [16] who apply it to node IDs and use it to address workload skew, churn and heterogeneous nodes. With their algorithm, k-Choices, they introduce the concept of passive and active balancing. However, their focus is on virtual server-based systems without range-queries. Giakkoupis and Hadzilacos [17] employ this technique to create a passive load balancing algorithm including a weighted version for heterogeneous nodes. There, joining nodes contact a logarithmic (in system size) number of nodes and choose the best position to join at. Their focus on the other hand is on balancing the address-space partition rather than arbitrary

loads. Manku [18] proposes a similar algorithm issuing one random probe and contacting a logarithmic number of its neighbors. An analysis of such algorithms using r random probes each followed by a local probe of size v is given by Kenthapadi and Manku [19]. However, only the nodespace partitioning is examined.

In Mercury [20] each node maintains an approximation of a function describing the load distribution through sampling. This works well for simple distributions, but as was shown in [21] it does not work for more complex cases such as filenames. Instead, [21] introduces OSCAR where the long-range pointers are placed by recursively halving the traversed peer population in each step. Both OSCAR and Mercury balance the in/out-degree of nodes. While this implies that the routing load in the overlay is balanced, it does not account for the placement of nodes according to item characteristics.

6 Conclusions

With the goal of investigating load balancing algorithms for distributed key/value-stores, we presented an active and a passive algorithm. The active algorithm is triggered periodically, while the passive algorithm uses joining nodes to improve system imbalance. We complement these algorithms with a placement function that splits a node's responsibility range according to the current key and workload distribution. Initial simulation results are promising showing that the system works well under churn and scales with increasing system sizes. Ongoing work include quantifying the cost of the algorithms within a prototype implementation of a key/value-store.

Acknowledgments. This work is partially funded by the European Commission through the SELFMAN project with contract number 034084.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP, pp. 205–220. ACM, New York (2007)
2. Rhea, S.C., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: Opendht: a public dht service and its uses. In: SIGCOMM, pp. 73–84. ACM, New York (2005)
3. Reinefeld, A., Schintke, F., Schütt, T., Haridi, S.: Transactional data store for future internet services. Towards the Future Internet - A European Research Perspective (2009)
4. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: Pick two. In: HotOS, USENIX, pp. 1–6 (2003)
5. Ghodsi, A., Alima, L.O., Haridi, S.: Symmetric replication for structured peer-to-peer systems. In: DBISP2P, pp. 74–85 (2005)
6. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM, pp. 149–160 (2001)

7. Vishnumurthy, V., Francis, P.: A comparison of structured and unstructured p2p approaches to heterogeneous random peer selection. In: USENIX, pp. 309–322 (2007)
8. Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: Voelker, G.M., Shenker, S. (eds.) IPTPS 2004. LNCS, vol. 3279, pp. 131–140. Springer, Heidelberg (2005)
9. Wang, X., Loguinov, D.: Load-balancing performance of consistent hashing: asymptotic analysis of random node join. IEEE/ACM Trans. Netw. 15(4), 892–905 (2007)
10. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: ACM Symposium on Theory of Computing, May 1997, pp. 654–663 (1997)
11. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: VLDB, pp. 444–455. Morgan Kaufmann, San Francisco (2004)
12. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range-queriable data structures. In: PODC, pp. 115–124 (2004)
13. Charpentier, M., Padiou, G., Quéinnec, P.: Cooperative mobile agents to gather global information. In: NCA, pp. 271–274. IEEE Computer Society, Los Alamitos (2005)
14. Byers, J.W., Considine, J., Mitzenmacher, M.: Simple load balancing for distributed hash tables. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 80–87. Springer, Heidelberg (2003)
15. Pitoura, T., Ntarmos, N., Triantafillou, P.: Replication, load balancing and efficient range query processing in dhts. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 131–148. Springer, Heidelberg (2006)
16. Ledlie, J., Seltzer, M.I.: Distributed, secure load balancing with skew, heterogeneity and churn. In: INFOCOM, pp. 1419–1430. IEEE, Los Alamitos (2005)
17. Giakkoupis, G., Hadzilacos, V.: A scheme for load balancing in heterogenous distributed hash tables. In: PODC, pp. 302–311. ACM, New York (2005)
18. Manku, G.S.: Balanced binary trees for id management and load balance in distributed hash tables. In: PODC, pp. 197–205 (2004)
19. Kenthapadi, K., Manku, G.S.: Decentralized algorithms using both local and random probes for p2p load balancing. In: SPAA, pp. 135–144. ACM, New York (2005)
20. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM, pp. 353–366. ACM, New York (2004)
21. Girdzijauskas, S., Datta, A., Aberer, K.: Oscar: Small-world overlay for realistic key distributions. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., Ouksel, A.M. (eds.) DBISP2P 2005 and DBISP2P 2006. LNCS, vol. 4125, pp. 247–258. Springer, Heidelberg (2006)

A.8 Generic Self-Healing via Rejuvenation: Challenges, Status Quo, and Solutions

Generic Self-Healing via Rejuvenation: Challenges, Status Quo, and Solutions

Artur Andrzejak
Zuse Institute Berlin
Takustr. 7, D-14195 Berlin, Germany
andrzejak@zib.de

***Abstract*—Software rejuvenation - in its simplest form a restart of a component or a program - is an efficient and universal approach for ad hoc healing of certain complex systems such as SOA components, telecommunication systems, and servers in data centers. Despite of its advantages this technique has not been widely deployed in other scenarios. The reasons are several shortcomings including loss of application availability and loss of working data due to a restart, and a lack of standardized support in operating systems, middleware, and component frameworks. In this position paper we argue that even partial remedies to these problems can turn rejuvenation into a powerful self-healing tool applicable to a larger variety of scenarios. We discuss rejuvenation-related problems, overview existing solutions, and propose a set of efficient architectural approaches which can pave the way to a universal adoption of this technique.**

I. What Is Rejuvenation?

The simplest form of *rejuvenation* [Huang et al., 1995], [Pfening et al., 1996], [Vaidyanathan and Trivedi, 2005] is a restart of a software component, an application, or the operating system. The purpose of this technique is the cleansing of internal data structures whose corruption (due to software or hardware faults) has caused a malfunction or performance degradation. More advanced forms include replacing a corrupted image by a stand-by replica [Silva et al., 2007] or "cleaning up" of critical data structures during run-time [Demsky and Rinard, 2003]. The most prominent usage of rejuvenation is to fight *software aging* (or rather software *state* aging) [Parnas, 1994]. The latter is a phenomenon of gradual decrease in system performance due to memory leaks, non-scheduled threads, accumulated rounding errors and other causes.

The idea to cleanse a system or its components in face of a malfunction instead of attempting to fix it has been exploited in other contexts as well. Examples include killing and restarting lightweight threads in the Erlang programming language [Wikstrom, 1994], and re-issuing web service requests if the response time exceeds a limit. Rejuvenation is also one of the fundamental forms of self-healing in organisms: the programmed cell-death (PCD) is triggered upon feedback from neighbors, stress or DNA-damage e.g. in order to prevent cancer. This mechanism is complemented by creation of new cells to ensure organic rejuvenation.

Even if rejuvenation is not a permanent solution to recurring problems and does not eliminate the need for diagnosing and repair of the root problems, this method is very attractive for broad classes of application scenarios and defect types:

> Rejuvenation is the only feasible "healing" approach in many complex software systems built of Commercial Off-The-Shelf Software (COTS), legacy code, or very complex frameworks / middleware / applications. This applies especially in production environments, where repairing an aging application is impossible (no source code), too costly, or if the real system and the failure conditions cannot be reproduced in a testing environment due to complexity.
>
> Rejuvenation is sufficient to "cure" faults which were caused by very rare (possibly non-deterministic) conditions or events. The complexity of today's systems (caused by size of applications, bloated communication protocols and overhead of component-based frameworks) can be enormous, preventing elimination of rare bugs in the testing phase. Paired with the non-determinism induced by distribution and concurrent execution, diagnosis of certain malfunctions can be almost impossible[1]. On the other hand, this does not preclude that malfunctions are rare in complex systems - they just have different causes. In such cases, rejuvenation is the most effective means to remove the corruption of the application state and restore its full performance and functionality.

In our opinion rejuvenation is rapidly gaining importance due to growing complexity and scale of today's software systems. These characteristics make it hard to quickly inspect systems in order to fix the root causes, and simultaneously cause the emergence of rare or non-deterministic faults.

In this position paper we argue that by solving several key problems rejuvenation can become a primary self-healing tool for a variety of scenarios. We discuss the mentioned obstacles, the status quo of solutions, and propose a set of efficient architectural approaches which might pave the way to a universal adoption of this technique.

II. Shortcomings of Rejuvenation and the Status Quo

Despite its effectiveness and universality, the use of rejuvenation is currently limited to specific environments and

[1] According to unofficial statements from Microsoft it is not uncommon that diagnosis of a *single* bug in a Windows device driver takes *weeks* for a skilled developer.

it is implemented in a hand-crafted way. Examples of most successful uses of this technique include (stateless) web and application servers, high-availability telecommunications hardware [Networks, 2008] and server healing in Microsoft data centers [Isard, 2007]. The lack of support for healing of generic software components and applications can be explained by several shortcomings of this technique and a non-existent infrastructural support for it.

Loss of availability. A restart implies a temporary loss of availability. Several approaches can reduce or partially removed this disadvantage. A lot of research has been done on the question *when* to rejuvenate [Dohi et al., 2000], [Andrzejak and Silva, 2007] (termed *adaptive rejuvenation*). However, these approaches neither shorten nor prevent the non-availability. Several solutions proposed in the recent years within the Recovery Oriented Computing (ROC) project [Candea et al., 2004] have gained most attention. They include *recursive rejuvenation* [Candea and Fox, 2001] and *microrebooting* [Candea and Fox, 2004]. These approaches can significantly shorten the loss of availability but do not completely eliminate it; moreover, they require changes in the architecture and implementation of a system. In [Silva et al., 2007] we proposed an approach that replicates aging application server based on virtualization techniques. It does not require code changes. While the decision to trigger the rejuvenation and replace an application by a hot-standby replica is based on a simple performance threshold, on optimization approach to schedule rejuvenation of multiple concurrently running replicas is presented in [Andrzejak et al., 2007].

Loss of working data during restart. This disadvantage limits the applicability of current approaches to (essentially) stateless components or applications. The recovery of data after a crash has been extensively studied in the context of database transactions [Haerder and Reuter, 1983]. Similar applies to recovery of distributed applications [Elnozahy et al., 2002]. However, these solutions require profound changes in the application architecture and cannot be used in a generic way. Both the Tandem / NonStop systems [Gray, 1990] as well as process groups [Birman, 1993] use multiple redundant processors simultaneously or pseudo-synchronously and deploy solutions to synchronize and transfer working data between them. These approaches require a significant redundancy of resources, a proprietary operating system or deep changes of applications and do not provide protections against permanent (or repeatable) software errors. Similar approaches including execution redundancy have been implemented in the IBM zSeries systems [Bartlett et al., 2004].

Lack of support in operating systems and component frameworks. The most serious obstacle (and a consequence of the above two shortcomings) to the widespread use of rejuvenation is the lack of support in the component frameworks, mainstream operating systems, middleware, or virtual machine managers (including browser engines). Hence currently each implementation carries a great cost of developing and infrastructure and algorithms for component / application analysis, decision taking, image replication, and transfer of working data (if applicable). The only existing support in OS's can be found in Solaris 10. It is limited to management of dependencies between system processes [Shapiro, 2004] to enable a partial reboot of the OS. In the domain of middleware, the framework presented in [Silva et al., 2007] allows for a transparent replication and restart of SOA servers. Besides of being a prototype, it has the disadvantage that the measurement and decision parts need to be adapted to each application. It also introduces a high resource overhead since a complete operating system (and not only a process) is virtualized.

III. Making Rejuvenation Cheap and Painless

The thesis of this paper is the following one.

> Rejuvenation can become one of the most generic and efficient solutions for self-healing of software components, applications and operating / distributed systems. The prerequisites to achieve this vision are:
>
> the introduction of *non-intrusive*, *transparent* support for rejuvenation in the mainstream operating systems / component frameworks / virtual machine managers / middleware, and
>
> practical solutions to the problems of loss of availability and loss of working data.

By *non-intrusive* and *transparent* we mean that the rejuvenation frameworks and methods should not require changes of the source code[2]. An even stronger constraint is that these approaches should not need knowledge about the " interior"of the component or an application. This is in contrast to the techniques introduced in the ROC project [Candea et al., 2004] e.g. microrebooting which demand changes or even complete rewriting of components. While these requirements surely limit the spectrum of available approaches, we believe that they are the key factors yet to ensure a widespread acceptance and effortless deployment of rejuvenation. Furthermore, they guarantee that the approach remains very universal and works regardless of the causes of errors.

We discuss in the following possible approaches towards the above goals, especially essential support in the OS's, methods for transfer of working data between process replica, and techniques for state cleansing without a restart.

A. Rejuvenation support in operating systems

An ideal support of rejuvenation in an OS or a component framework such as J2EE would provide a standardized API to specify the following functions and parameters: setting for each component or process whether it should be protected by rejuvenation; conditions or routines for triggering rejuvenation according to metrics provided by the OS; policies for process replication in faces of limited resources (cores, memory); buffering of inputs and messages in order to recompute working data (see Section III-B). In order to enable this scenario,

[2]This does not preclude methods such as binary code rewriting or AOP.

some sophisticated techniques need to be investigated and implemented.

Identifying component/process errors. A necessary functionality is the detection of potential state corruption. For the case of crashes this is trivially solvable, yet much harder in case of partial malfunctions. In case of performance degradation, on-line monitoring and performance modeling [Andrzejak and Silva, 2008] coupled with scheduling of restarts [Andrzejak and Silva, 2007], [Andrzejak et al., 2007] can be used.

Adaptive replication of processes. Rejuvenation without loss of availability can be achieved via process replication [Silva et al., 2007], [Andrzejak et al., 2007]. Today's hardware (multiple cores, several GB of memory) make this approach feasible - users are more likely to "sacrifice" a processor core that to experience availability outage. Such replication of processes can be achieved by lightweight virtualization [Yu et al., 2008] and techniques for rapid replication of virtual resources [Lagar-Cavilla et al., 2009]. In addition it is necessary to develop policies for controlling the number replicas depending on the available resources, workload, and the "importance" of protected processes.

B. Transfer of working data

A major shortcoming of rejuvenation is a possible loss of working data of *stateful* components or applications. This is usually caused by a (forced) restart of a component / application or its substitution by an uncorrupted replica. Several issues make this problem difficult.

First, in many cases the essential working data is hard to tell apart from the corrupted parts of the data. For example, identifying loitering objects ("memory leaks") in Java is a non-trivial problem [Mitchell and Sevitsky, 2003]. Consequently, migrating the whole component / application state indiscriminately is likely to lead to an anew corruption. Approaches outlined below attempt to tackle this core problem of state transfer. Another, more intrusive approach is based on invariants of data structures (see Section III-C).

Second, application components (understood here as OS processes) have both an "internal state" represented by the local data structures, and "external state" expressed by entries in OS tables, file / network handles, and references held by other components (e.g. process ID, open socket connections). While the transfer of the internal state should require least or no changes in application code, the external state requires changes in the hosting OS, such as intercepting of the input and output function calls and transparent replacement of file / network handles. This functionality (partially provided by lightweight virtualization [Yu et al., 2008]) is clearly to be implemented as a part of the OS support discussed in Section III-A, and requires deep changes in the process management.

Finally, in today's multi-component, multi-threaded applications, an unintentionally changed behavior (due to state changes or even new timing patterns) of one component might break the consistency of the whole application. As a consequence, partial rejuvenation in such complex applications might cleanse a component but cause a corruption on the application level. As a partial solution, OS supporting rejuvenation should provide an API which allows to specify component dependencies (e.g. maximum response latencies) for multi-process applications.

Following the arguments from Section III on importance of non-intrusiveness we focus here on several solutions for transferring of "internal state" which require least amount of changes in the application code. However, also more invasive (yet highly effective) solutions exist[3]. One of them are restrictions on the component development, e.g. requiring programmers to implement methods for saving the working data (similarly to methods `onStop()` / `onRestart()` of the `Activity` class in the Android APIs [Open Handset Alliance, 2009]).

Creating a state without working data. Some applications are stateful yet provide mechanisms for persistent storing of their working data, and for retrieval of them. This category include EJB components (via Container Managed Persistence), text editors, and web browsers which can save session data upon closing them. In these cases a state without "unsaved" working data can be achieved easily; then a restart or replacement of the component with an subsequent retrieval of the working data can be enforced. For example, in e.g. text editors or browsers saving of the current state can be enforced without code modifications - via simulation of user input or OLE automation in Windows. In application servers, EJB can be forced to save their state before requests are redirected to a replica initialized with this saved state [Silva et al., 2007]. One of the difficulties here is ensuring availability after the main process no longer accept inputs, but the replica is not yet active or up-to-date.

State mirroring with input filtering. A more general but costly mechanism is on-line mirroring of working memory of a process. This can be complemented by the buffering of user input and incoming messages. Techniques applied for the rapid replication of virtual machines [Lagar-Cavilla et al., 2009] can be used to obtain an efficient implementation. After a rejuvenation or substitution of the corrupted image with a replica, the buffered input is fed to the application in order to achieve consistency. While this approach resolves rare errors due to non-determinism, it might fail if the replica inherits the same corrupted data as the original process. In some cases the last problem can be addressed by the using methods of Delta Debugging [Zeller, 2002]. In brief, recorded inputs can be selectively "replayed" in order to observe which ones produce a state corruption or a crash. Those are then subsequently filtered out to avoid repeated corruption.

C. State cleansing without restart

An alternative to a component restart is cleansing of internal data structures during the run-time. This method reduced the need to reboot a part of large telecommunication system from

[3]As pointed out by an anonymous reviewer.

once per week to about twice per year [Singhal, 2007]. While this approach avoids a loss of availability and issues of state transfer, it is restricted to components / applications with known source code or to environments which allow inspection and change of code and data (such as JVM). Following approaches might be investigated in this context:

Swapping of leaking objects. Performance degradation due to memory leaks is only noticeable when available memory is low. By using technologies such as LeakBot [Mitchell and Sevitsky, 2003] it is possible to identify automatically leaking objects in Java. These objects can be then swapped to hard drive, delaying (in terms of time or work done) the need for restart by a factor or larger.

Automatic repairing of data structures during runtime. Existing techniques for cleansing of data structures [Demsky and Rinard, 2003] require a programmer to specify invariants. This approach is time consuming and not possible without knowledge of the source code. In order to automatize the creation such invariants statistical methods could be used to create invariants automatically, or at least propose to the developer potential invariants. To this aim variations of methods such as statistical debugging [Zheng et al., 2006] could prove useful.

IV. Conclusion

In this paper we gave a brief overview on software rejuvenation, a technique to (temporarily) remedy state corruption and performance problems in complex software systems. We argued that it has a largely unexploited potential to become a primary self-healing solution in such systems. We have also discussed the major obstacles for the widespread acceptance of this technique - notably, a lack of support in OS's / component frameworks, loss of availability and loss of working data - and outlined several possible solutions to these problems.

As next steps we plan to evaluate some of the presented approaches for transparent rejuvenation and working data transfer within the J2EE environment using the RUBiS benchmarking framework [Pugh and Spacco, 2004].

V. Acknowledgements

This research work is carried out in part under the SELFMAN project (contract 034084, FP6) funded by the EC.

References

[Andrzejak et al., 2007] Andrzejak, A., Moser, M., and Silva, L. (2007). Managing performance of aging applications via synchronized replica rejuvenation. In *DSOM 2007*, Silicon Valley, CA, USA.

[Andrzejak and Silva, 2007] Andrzejak, A. and Silva, L. (2007). Deterministic models of software aging and optimal rejuvenation schedules. In *10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany.

[Andrzejak and Silva, 2008] Andrzejak, A. and Silva, L. (2008). Using machine learning for non-intrusive modeling and prediction of software aging. In *IEEE/IFIP Network Operations & Management Symposium (NOMS 2008)*, Salvador de Bahia, Brazil.

[Bartlett et al., 2004] Bartlett, W., Society, I. C., and Spainhower, L. (2004). Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1:2004.

[Birman, 1993] Birman, K. P. (1993). The process group approach to reliable distributed computing. *Communications of the ACM*, 36:37–53.

[Candea et al., 2004] Candea, G., Brown, A. B., Fox, A., and Patterson, D. A. (2004). Recovery-oriented computing: Building multitier dependability. *IEEE Computer*, 37(11):60–67.

[Candea and Fox, 2001] Candea, G. and Fox, A. (2001). Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS*, pages 125–130. IEEE Computer Society.

[Candea and Fox, 2004] Candea, G. and Fox, A. (2004). End-user effects of microreboots in three-tiered internet systems. *CoRR*, cs.OS/0403007.

[Demsky and Rinard, 2003] Demsky, B. and Rinard, M. C. (2003). Automatic detection and repair of errors in data structures. In *OOPSLA 2003*, pages 78–95.

[Dohi et al., 2000] Dohi, T., Goseva-Popstojanova, K., and Trivedi, K. S. (2000). Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In *PRDC 2000*, pages 77–84. IEEE Computer Soc. Press.

[Elnozahy et al., 2002] Elnozahy, Alvisi, Wang, and Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *CSURV: Computing Surveys*, 34.

[Gray, 1990] Gray, J. (1990). A census of tandem system availability between 1985 and 1990. Technical Report 90.1, Tandem Computers.

[Haerder and Reuter, 1983] Haerder, T. and Reuter, A. (1983). Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–317.

[Huang et al., 1995] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. (1995). Software rejuvenation: Analysis, module and applications. In *Proceedings of Fault-Tolerant Computing Symposium FTCS-25*.

[Isard, 2007] Isard, M. (2007). Autopilot: automatic data center management. *Operating Systems Review*, 41(2).

[Lagar-Cavilla et al., 2009] Lagar-Cavilla, H. A., Whitney, J., Scannell, A., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. (2009). Snowflock: Rapid virtual machine cloning for cloud computing. In *3rd European Conference on Computer Systems (Eurosys)*, pages 1–12, Nuremberg, Germany.

[Mitchell and Sevitsky, 2003] Mitchell, N. and Sevitsky, G. (2003). LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. *Lecture Notes in Computer Science*, 2743:351–377.

[Networks, 2008] Networks, J. (2008). *JUNOS 9.3 Configuration Guides - High Availability*. Manual.

[Open Handset Alliance, 2009] Open Handset Alliance (2009). *Android SDK Reference*.

[Parnas, 1994] Parnas, D. L. (1994). Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287.

[Pfening et al., 1996] Pfening, A., Garg, S., Puliafito, A., Telek, M., and Trivedi, K. S. (1996). Optimal software rejuvenation for tolerating soft failures. *Perform. Eval*, 27/28(4):491–506.

[Pugh and Spacco, 2004] Pugh, B. and Spacco, J. (2004). RUBiS revisited: why J2EE benchmarking is hard. *ACM SIGPLAN Notices*, 39(10):204–205.

[Shapiro, 2004] Shapiro, M. W. (2004). Self-healing in modern operating systems. *ACM Queue*, 2(9):66–75.

[Silva et al., 2007] Silva, L. M., Alonso, J., Silva, P., Torres, J., and Andrzejak, A. (2007). Using virtualization to improve software rejuvenation. In *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, Cambridge, MA, USA.

[Singhal, 2007] Singhal, S. (2007). Private communication. HP Labs.

[Vaidyanathan and Trivedi, 2005] Vaidyanathan, K. and Trivedi, K. S. (2005). A comprehensive model for software rejuvenation. *IEEE Trans. Dependanble and Secure Computing*, 2(2):1–14.

[Wikstrom, 1994] Wikstrom, C. (1994). Distributed computing in Erlang. In *First International Symposium on Parallel and Symbolic Computation*.

[Yu et al., 2008] Yu, Y., Kolam, H., Lam, L.-C., and Chiueh, T. (2008). Applications of a feather-weight virtual machine. In Gregg, D., Adve, V. S., and Bershad, B. N., editors, *VEE 2008*, pages 171–180, Seattle, WA, USA.

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA. ACM Press.

[Zheng et al., 2006] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. (2006). Statistical debugging: simultaneous identification of multiple bugs. In Cohen, W. W. and Moore, A., editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM.

A.9 DHT Load Balancing with Estimated Global Information

DHT Load Balancing with Estimated Global Information

Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Submitted by: Nico Kruber

Supervisor: Prof. Dr. Alexander Reinefeld
Second reader: Prof. Dr. Miroslaw Malek

Berlin, 25th September 2009

Declaration of own work

I, Nico Kruber, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Place, Date:

Abstract

One of the biggest impacts on the performance of a Distributed Hash Table (DHT), once established, is its ability to balance *load* among its nodes. DHTs supporting range queries for example suffer from a potentially huge skew in the distribution of their items since techniques such as *consistent hashing* [29] can not be applied. Thus explicit load balancing schemes need to be deployed. Several such schemes have been developed and are part of recent research, most of them using only information locally available in order to scale to arbitrary systems.

Gossiping techniques however allow the retrieval of fairly good estimates of global information with low overhead. Such information can then be added to existing load balancing algorithms that can use the additional knowledge to improve their performance. Within this thesis several schemes are developed that use global information like the average load and the standard deviation of the load among the nodes to primarily reduce the number of items an algorithm moves to achieve a certain balance. Two novel load balancing algorithms have then been equipped with implementations of those schemes and have been simulated on several scenarios. Most of these variants show better balance results and move far less items than the algorithms they are based on.

The best of the developed algorithms achieves a $15 - 30\%$ better balance and moves only about $50 - 70\%$ of the number of items its underlying algorithm moves. This variation is also very robust to erroneous estimates and scales linearly with the system size and system load. Further experiments with self-tuning algorithms that set an algorithm's parameter according to the system's state show that even more improvements can be gained if additionally applied. Such a variant based on the algorithm described by Karger and Ruhl [30] shows the same balance improvements of $15 - 30\%$ as the variant above but reduces the number of item movements further to $40 - 65\%$.

Contents

1. Introduction

1.1. Context

Distributed Hash Tables (DHTs) store key/value-pairs on several nodes of a network and provide means for inserting, retrieving and deleting a value associated with a key. Each node is assigned a unique node ID in a given ID space uniformly at random and is then responsible for all values with keys near its ID (keys are also mapped to this ID space). By using a technique called *consistent hashing* [29], the DHT then spreads the stored items uniformly over the node ID space which achieves a fair balance without any further effort. More precisely, nodes will have loads varying by $O(\log n)$ times the average load in terms of stored items in a system of n nodes [36, 25]. However, DHTs with range queries like Scalaris [39] cannot use hash functions to spread their items because they need to stay in the order given by their keys. Therefore more effort is needed to balance items among the nodes in such storages.

Consider the following example: Articles are to be stored in a range-query-based DHT with 100 nodes and the key under which an article is stored is its heading. In case of keys in (American) English and nodes responsible for equidistant key ranges, items would then be distributed as shown in Figure 1.1.

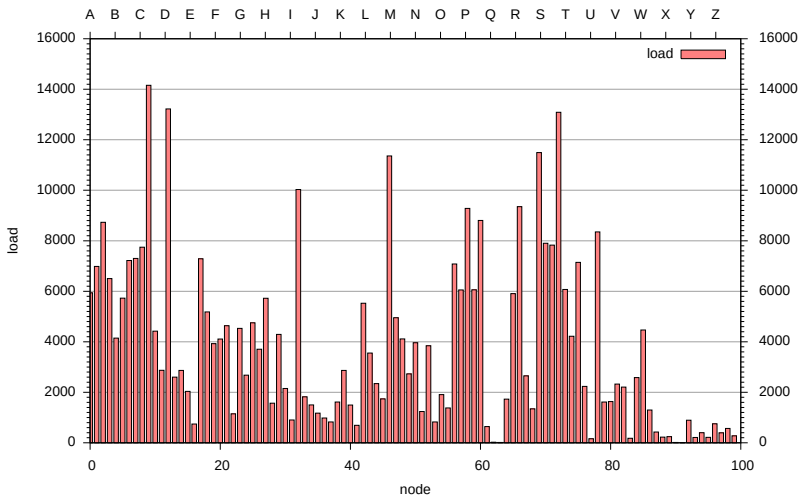


Figure 1.1.: *Item distribution of US-English words on 100 nodes with equidistant key ranges. (list of words aggregated from [11])*

To even out such skewed load distributions load balancing algorithms are required which change the nodes' responsibilities that in turn reduces their load. Such algorithms try to balance an arbitrarily defined *load* at each node and should only use information locally available in order to scale to large systems. Several of such algorithms will be introduced in the following chapters, some of them using different definitions of *load*, e.g. the number of keys a node is responsible for, the number of items a node actually stores or the access-popularity of a node's items. Some also weight load depending on a node's capacities and therefore adapt to heterogeneous environments.

1.2. Aims & Objectives

This thesis aims at improving such load balancing algorithms in terms of moved items and reached balance by adding estimates of global information. These values can be retrieved with high confidence and low overhead using gossiping techniques [23, 28] and include approximations of values like the minimum, maximum and average load as well as the standard deviation and system size.

A first approach will try to use the average load of all nodes in its decision on whether to balance two nodes and how much to transfer from one node to another. Most algorithms simply balance two nodes that have been matched by trying to equalise their loads. This does however not involve a node's ideal load - the average load. Therefore several items are transferred multiple times during the algorithm's task to balance the load at each node of the system, especially if a node's load after a balance operation is sufficiently higher than the average load. By knowing the target load and integrating it appropriately, a much better performance can be expected. Preliminary results of an algorithm using the average load already show some improvements compared with its underlying algorithm without that change [27].

Further variations will be introduced into ordinary load balancing algorithms also including some of the other information mentioned above. The resulting algorithms' performance will then be evaluated on a set of given scenarios like the alphabetical distribution of Figure 1.1. It is assumed that with the right use of such information, any algorithm can be significantly improved.

1.3. Methods

In order to evaluate any of the introduced algorithms, a simulation will be implemented that emulates a simple DHT with range queries and starts with an initial system load distributed among the nodes according to a given scenario. This emulation will disregard node joins and deletions as well as any other side-effects, e.g. network maintenance,

node failures, network delay and bandwidth etc., to eliminate any other influences when assessing an algorithm. It will however allow analysing the algorithm under different aspects such as different scenarios with different numbers of items and nodes, different choices for an algorithm's parameters and a different accurateness of the estimated global information. It will also provide the ability to run multiple simulations with the same set of parameters in order to allow the evaluation of randomised algorithms that show (slightly) different behaviour in each simulation.

The program will follow the strategy of being easily extensible and will in particular allow additional algorithms and scenarios to be deployed separately and added dynamically via a plugin-based infrastructure. It will also provide means of comparing different algorithms with varying parameters on multiple scenarios. A graphical user interface and a command line client will be created that allow fast evaluations of the algorithms as well as batch-jobs for more time-consuming simulations.

1.4. Achievements

At first a survey of the field of Distributed Hash Tables has been given by presenting their concepts and examining their mode of operation including DHTs that support range queries. Additionally a very thorough overview of load balancing schemes that can be applied to (arbitrary) DHTs has been given and several novel load balancing algorithms have been presented. Gossiping algorithms have also been introduced to present a way estimates of global information can be retrieved in Distributed Hash Tables.

Secondly several algorithm variations have been introduced that make use of estimated global information in order to minimise the item movements an algorithm performs as well as the imbalance it reaches. Those variations have then been applied to the load balancing schemes by Karger and Ruhl [30] and Bharambe et al. [12] and have been evaluated by performing simulations with different load distribution scenarios. These variations use estimated values of the system's average and maximum load, the standard deviation of the load among the nodes and the system size.

The best algorithm among those variants limits the original algorithm's item movements in a way that nodes that have a load smaller than the (estimated) average load will not reach a load above this bound. Additionally it only performs such balance operations that increase the standard deviation by at least a factor s/n with n being the system size and s a configurable parameter that has been set to 2.0 and 3.0 for the algorithms by Karger and Ruhl and Bharambe et al. respectively. This variation achieves an up to 30% lower imbalance than the algorithm applied to would achieve alone and only moves about $50 - 70\%$ of its items.

Further experiments that try to tune the algorithms' parameters according to the

system's state show that even more improvements are possible. Applying such *self-tuning* to the algorithm by Karger and Ruhl for example has shown an up to 30% lower imbalance with only about $40 - 65\%$ of the item movements of the original algorithm. Such good performance has however not been achieved by a similar variant that has been applied to the algorithm by Bharambe et al. Further investigations into the field of *self-tuning* algorithms are thus needed.

1.5. Outline

At first a deeper insight into the techniques behind Distributed Hash Tables (DHTs) will by given in Chapter 2. It will also present several representatives of DHTs and their characteristics and will introduce methods for achieving range-queriable systems. It will then present gossip algorithms followed by several novel load balancing schemes that are available for such DHTs. Chapter 3 will define the system model that is used in this thesis and the algorithms that have been chosen to be equipped with estimated global information. It will conclude with the introduction of the algorithm variants that have been developed. These variants will be evaluated in the following Chapter 4 starting with a detailed description of the evaluation process itself and the scenarios used. It then presents the results that the implemented simulator created for the different algorithms under different aspects of the simulation. Chapter 5 will finally sum up the achievements of the thesis and will provide ideas about possible extensions of the given algorithms and future work.

2. Background / Related Work

This chapter presents Distributed Hash Tables (DHTs), a prominent representative of the class of structured overlay networks, which has been of great interest in research over the past years. The structure of a generic DHT and some of its representatives will be introduced including DHTs that support range queries. This class of DHTs imposes some restrictions on the organisation of the stored resources which need to be considered when designing load balancing algorithms, e.g. stored resources cannot move arbitrarily to different nodes of the network. Before load balancing algorithms are described in the last section of this chapter, gossiping algorithms will be introduced. Those algorithms can be used in peer-to-peer networks to gather estimates of certain global information that is usually not available in such a setting. This information will later be used to improve some of the load balancing algorithms described here.

2.1. Distributed Hash Tables (DHTs)

Distributed Hash Tables provide functionality similar to ordinary hash tables. They store *key/value*-pairs on several nodes of a network and provide look-up facilities for retrieving the value associated with a given key. Several such systems exist, but despite their diversity a reference model can be given which models their approaches in a generic manner [8] and is outlined below.

In this model, a DHT maps peers P and resources R to a common identifier space I using mapping functions $f_P : P \to I$ and $f_R : R \to I$. Furthermore, a closeness metric $d : I \times I \to \mathbb{R}$ is defined on I which can be used by a mapping function $M : I \to 2^P$ that associates identifiers with the peers storing them. The peers themselves are organised in a logical network to allow access to every other peer's resources, i.e. by embedding a graph into the identifier space. Following this notation, differences between several DHTs only exist because of the different choices made for the following aspects:

- *Selection of an identifier space with a closeness metric d:* This serves as an address-space for resources and peers and should be large enough to support large systems.
- *Mappings f_P and f_R:* These functions may satisfy certain distributional properties which can be exploited for load balancing. They can preserve resource semantics

such as closeness/neighbourhood relations or the order under a given key or completely drop them, e.g. when following a uniform distribution in I.

- *Management of the identifier space:* The function $M : I \to 2^P$ assigns each identifier of a resource a set of peers responsible for it. Locating resource r therefore involves finding a peer in $M(f_R(r))$. Note that systems with replication have several peers responsible for each resource.

- *Structure of the logical network:* The logical network can be modelled as a (time-dependent) directed graph $G = (P, E)$ with vertices P (peers) and edges E (direct connections). Also let $N(p)$ be the set of peers a given peer p maintains a connection to, e.g. its neighbours. The overall structure of that graph is then determined by $N(p)$ for every $p \in P$.

- *Routing strategy:* Requests for identifiers need to be routed to their responsible peers. A strategy for that can be described as selecting at a given peer p for an identifier i a set of next peers $R(p, i) \in N(p)$ to which to forward a request. Routing is typically greedy, i.e. $\forall q \in R(p, i) : d(i, f_P(q)) \leq d(i, f_P(p))$, and built on top of the decisions made for the identifier space and its management, e.g. the distance function.

- *Maintenance strategy:* Changes in peer connectivity (referred to as *churn*) may occur quite frequently and create the need for mechanisms to repair the state of the logical network. Since node joins are typically active operations, this task mainly focuses on repairing connections due to node (connection) failures. Maintenance strategy can either follow a proactive approach (heartbeats, periodic probing) or a reactive approach (correction on use, failure or change) or a combination of the two. Functionality of the DHT heavily relies on a consistent network structure making this strategy essential for its operation.

Additionally DHTs provide (supposedly different) implementations for a common set of functionality they expose to their clients.This includes joining and leaving a network, several routing functions, looking up identifiers and getting some administrative information about the local peer and its neighbours. Data management functionality exposes insert, delete and update methods as well as searching for resources using queries of some kind.

Implementations of such structured overlay networks include CAN [37], Pastry [38], Chord [41], Freenet [19], Tapestry [45], Gnutella [5, 1] and more. The following sections will concentrate on the first three which all implement a variant of consistent hashing [29, 33] outlined below. The main focus however is not on a complete description of the different DHTs but to give an overview of their structure and message routing / resource

retrieval algorithms which both is important for load balancing. Also although the descriptions will make use of the introduced terminology and definitions they will not be structured explicitly that way in order to focus on the main aspects of design decisions and establish a better understanding of the techniques. It will however become apparent that the DHTs follow the given model.

2.1.1. Consistent Hashing

While traditional hash tables map objects to a static set of buckets, the number of peers to which resources are mapped constantly changes in DHTs. Karger et al. [29] and Lewin [33] describe a *consistent hash function* that operates on a changing set of buckets and provides some *consistency properties*, e.g. adding a bucket only changes the mappings of a minimum fraction of objects needed to maintain a balanced state.

Using the aforementioned syntax, let P be a set of $n = \|P\|$ peers, I the circular interval $[0, 1) \subset \mathbb{R}$ and f_R a random function that maps resources of R $\log(n)$-way independently[1] and uniformly to I. Now let each real peer p run m "virtual" peers that operate independently from each other. Virtual peers can be modelled by each peer being mapped to m different identifiers instead of just one: $f_P : P \to I_m \subset I, \|I_m\| = m$ (otherwise the same constraints as f_R apply). Also define a function M that maps each resource $r \in R$ to the peer $p \in P$ that has the closest identifier to $f_R(r)$. Each such hash function has the following properties which also hold for large enough arbitrary I:

- *Monotonicity:* If new peers are added to P, resources only move from old peers to new peers, but never between old peers.
- Adding a peer p to P changes the mappings of $O(\|R\|/n)$ resources.
- *Balance:* The probability of a resource $r \in R$ being assigned to peer $p \in P$ is $O\left(\frac{1}{n} \cdot \left(1 + \frac{\log(n)}{m}\right)\right)$.

Thus using $m = \Omega(\log(n))$ virtual servers results in a well-balanced state with each node being responsible for $O\left(\|R\|/n\right)$ resources. Having no virtual nodes however ($m = 1$) yields to some nodes having $O(\log(n))$ times more resources associated with them than others because each node is responsible for $O\left((\log(n) + 1) \cdot \|R\|/n\right)$ resources.

2.1.2. CAN

A basic CAN network [37] uses a virtual d-dimensional Cartesian coordinate space C for its identifiers and places it on a d-torus for routing. Peers are responsible for their

[1] A random mapping function is k-way independent if any k elements are mapped independently. This allows representing real identifiers in $I \subset \mathbb{R}$ with limited precision instead of using an infinite number of bits and also allows for arbitrary large enough discrete I.

individual and distinct zones of this coordinate space which is entirely covered at any point in time. A *key*/*value*-pair is stored in CAN by mapping its *key* to a point $q \in C$ using a uniform hash function and storing it at the peer p responsible for the zone containing q. Similarly querying for a *key* corresponds to routing to the node responsible for the zone containing q. For this every peer maintains a list of immediate neighbours (nodes with zones adjoining their own zone) and routing a message at peer p directed to q is done by forwarding it to the neighbour of p which is responsible for coordinates closest to q (greedy forwarding). Also note that several paths exist and can be used in case of node and connection failures or to deploy a simple request load balancing (see Figure 2.1 for an example). This way using d dimensions and $n = \|P\|$ commensurate zones, each individual node maintains $2d$ neighbours and average routing paths cross $(d/4)(n^{1/d})$ zones (peers).

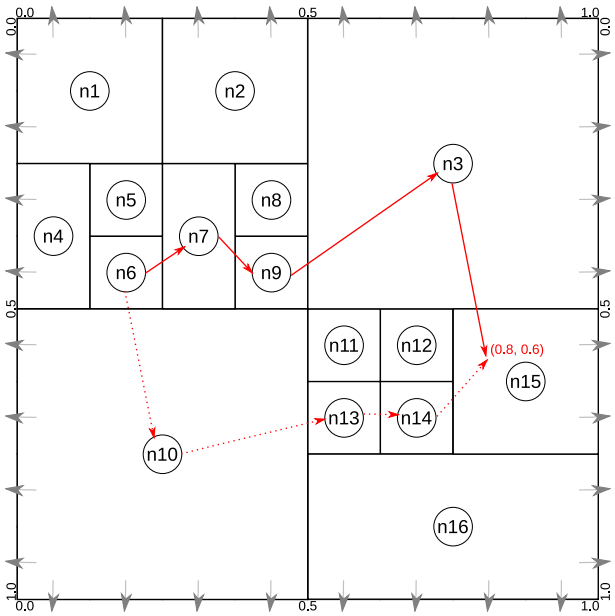


Figure 2.1.: *Planar 2-d CAN with coordinates in range* $[0,1) \times [0,1) \subset \mathbb{R}^2$ *with 16 nodes routing a message from node* $n6$ *to* $q = (0.8, 0.6)$ *(dashed arrows present an alternative route).*

If greedy forwarding fails, an expanding ring search using stateless, controlled flooding may be used to locate peers closer to the destination. From there greedy forwarding will be continued.

In order for a node to join an existing CAN, it needs to take the following steps:

1. Find a CAN node by using some external mechanism, e.g. DNS.
2. (Randomly) choose a point q to join at, contact the peer currently responsible for that point (using normal routing) and split the zone in half assigning one half to the new node.

3. Learn neighbours from the previous occupant and notify them of its arrival.

4. Transfer resources according to the new responsibilities.

These steps involve $O(d)$ existing nodes which need to change their list of neighbours.

A node gracefully leaving the system hands over its zone to a neighbour which is able to join the two zones. If this is not possible the neighbour with the smallest zone will (temporarily) handle both zones. To identify node failures peers normally send periodic update messages to their neighbours. If such message is not received for some time its neighbours each initiate a takeover mechanism and agree on one of them taking over the failed zone, possibly becoming responsible for two zones. To prevent further fragmentation of the coordinate space background zone-reassignment algorithms try to merge zones again.

Further improvements were suggested in order to reduce routing path lengths, i.e. increasing the number of dimensions or using multiple coordinate spaces (*realities*), introducing round-trip-times into routing decisions, caching frequently requested resources or replicating them to their peers' neighbours. See [37] for more details and an evaluation of the various improvements.

2.1.3. Pastry

A different approach to realising a distributed hash table is provided by Pastry [38] which uses identifiers represented by 128-bit numbers. The circular identifier space therefore consists of integers in the range $[0, 2^{128} - 1]$ and the two mapping functions f_P and f_R are expected to distribute their results uniformly and independently among the identifier space. Resource identifiers are for example created by applying a secure hash function, e.g. SHA-1 [7], to the resource's name, content and the resource owner's identifier. Also resource $r \in R$ is stored at the k peers with identifiers numerically closest to $f_R(r)$. k can be set individually for each resource at its insertion, influences its availability in case of failures and provides some means of balancing resource requests.

For routing purposes node identifiers in pastry are sub-divided into separate *levels* of b bits with a *domain* at level l being defined as the bits from position $(b \cdot l)$ to $(b \cdot (l+1) - 1)$. Messages are now forwarded using prefix routing, i.e. messages at peer p with destination $f_R(r)$ matching $f_P(p)$ up to level l will be forwarded to a node whose identifier matches the destination's identifier up to at least level $(l+1)$. Also each pastry node p stores information about other peers in 3 different node sets:

- a *routing table* T which contains information about representatives of different domains at different levels: for each level l it contains IP addresses of $(2^b - 1)$ peers with the same prefix as $f_P(p)$ up to level $(l-1)$ but a different domain at level l (to improve route locality, a representative geographically close to p can be chosen),

- a *namespace set* L that contains identifiers and IP addresses of $\|L\|$ peers that are numerically close and centred around $f_P(p)$ which is used for routing too and
- a *neighbourhood set* M storing identifiers and IP addresses of $\|M\|$ peers that are geographically close to p and which is useful for network maintenance. Note that this set has been dropped in later versions of Pastry [15]. The following descriptions however are based on the original version.

While the choice of b influences the size of the routing table (ca. $\lceil \log_{2^b}(n) \rceil \cdot (2^b - 1)$ entries, $n = \|P\|$) and the average routing path length, the sizes of $\|L\|$ and $\|M\|$ can be chosen arbitrarily and are typically 2^b and 2^{b+1} respectively. Using those tables, a message to resource r arriving at peer p is routed as follows:

if $f_R(r)$ is in the range of the two farthest nodes in p's namespace set $N(p)$:
Forward message to $p_i \in N(p)$ so that $|f_R(r) - f_P(p_i)|$ is minimal (possibly p).
else if p's routing table contains a node that shares a longer prefix than p:
Forward the message to that node.
else:
Forward to a known node (from the routing table, namespace set or neighbourhood set) that shares a prefix at least as long as p but is numerically closer to r.

Although the third case creates a worst-case with linear performance (in the number of nodes), Rowstron and Druschel [38] argue that this is very unlikely due to the uniform distribution of node identifiers and give an average routing path length of $\lceil \log_{2^b}(n) \rceil$ hops.

Nodes joining a Pastry network need to perform the following 6 steps which involve $O(\log_{2^b}(n))$ remote procedure calls:

1. Find a Pastry node p_1 by using some external mechanism.
2. Choose a node identifier (at random), contact the peer p_2 currently responsible for resources with that identifier using normal routing.
3. Update its routing tables using the neighbourhood set of p_1 and the namespace set of p_2 as approximations of its own neighbourhood and namespace sets. Fill the routing table with information from the nodes the *join* message came along.
4. Improve those approximations by requesting the state of the nodes in its routing table and neighbourhood set.
5. Notify peers that need to be aware of the new node and send them its own state.
6. Transfer resources according to the new responsibilities.

Node failures are detected when a node tries to contact another node in its routing table or namespace set. The latter is repaired by using an appropriate node of the namespace set of the live node with the largest identifier in the direction of the failed node. Repairing a representative in the routing table involves contacting another representative at the same level and asking it for the required connection or continuing with requests to nodes at higher levels. The neighbourhood set can be repaired by requesting the neighbourhood sets of the other live nodes in it.

2.1.4. Chord

Chord [41] places identifiers of m bits on a circle modulo 2^m and performs every calculation modulo 2^m. A secure hash function, typically SHA-1 [7], is used for mapping resources and peers to this identifier space ($m = 160$ in case of SHA-1). f_P maps peers to identifiers by hashing their IP address and f_R hashes the key of a resource. Using consistent hashing, M maps a resource $r \in R$ to the peer $p \in P$ whose identifier is equal to or follows r's identifier in the identifier space. That is if $predecessor(p)$ denotes the predecessor of a peer p on the identifier circle, p is responsible for all resources with identifiers within $(f_P(predecessor(p)), f_P(p)]$. Note that Chord does not use an explicit load balancing scheme but instead relies on consistent hashing with the use of virtual servers.

Chord nodes store routing information about m nodes in a so-called *finger table*. The i'th *finger* of peer p's table, $1 \leq i \leq m$, points to the node p' whose identifier succeeds $f_P(p)$ by at least 2^{i-1}, i.e. $p.finger[i] = M(f_P(p) + 2^{i-1})$, $p.finger[1] = successor(p)$. Note that consecutive fingers can point to the same node if there is no peer between their designated identifiers. Additionally to the finger table each node maintains a pointer to its predecessor to simplify node join and leave operations. Figure 2.2 shows such a node's complete routing state (including its finger table) in an exemplified Chord ring.

Routing uses those fingers as shortcuts to reach the destination with fewer hops than using successor links alone (which would suffice for routing correctness and result in $O(n), n = \|P\|$ hops)). If peer p needs to find the node p' which is responsible for key k, it searches its finger table for the node j whose identifier immediately precedes k and asks j for the node it thinks is closest to k. This procedure is repeated until the immediate predecessor of k is found, whose successor is then the node responsible for k. Note that those messages could also be forwarded to the nodes recursively instead of implementing an iterative approach as described here. Because the fingers provide shortcuts half-way, quarter-way,... around the circle and the distance to the destination halves in each step this results in $O(\log(n))$ nodes to contact (with high probability[2]).

[2] *with high probability* means probabilities of at least $(1 - O(n^{-1}))$, n being the system size

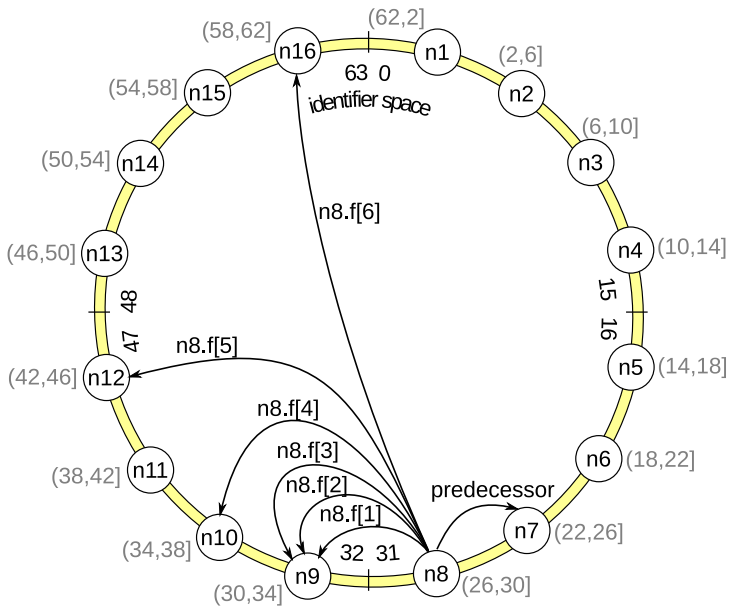


Figure 2.2.: *Chord ring with $m = 6$ and 16 nodes showing the routing pointers of node $n8$ and all nodes' responsibilities in grey (associations with identifiers).*

Nodes joining an existing Chord ring need to take the following steps:

1. Find a Chord node p_1 by using some external mechanism.
2. Initialise its predecessor and finger pointers by asking p_1 to look them up or copy from a neighbour's finger table and find the correct values on its own (the neighbour of the to-be-inserted node p can be retrieved by asking p_1 to look up $M(f_P(p))$).
3. Contact (existing) peers that need to be aware of the new node and update their predecessor and finger pointers.
4. Transfer resources according to the new responsibilities.

Alternatively, step 3 could be omitted if the Chord nodes periodically run a stabilisation protocol that fixes their finger tables. Note that this would also allow Chord to handle concurrent joins.

To deal with node failures first recall that Chord only needs to maintain correct successor pointers in order to work properly. To overcome failures of successor pointers, each peer stores an additional list of r successors and uses the first live node in that list in such case. The stabilisation protocol mentioned above also ensures that the finger tables are corrected in case of node failures. Meanwhile, alternative nodes to forward routing messages to are found in the finger table (using the preceding finger to the failed one) or in the successor-list.

Both, a node joining and leaving a Chord ring will require $O\left(\log^2(n)\right)$ messages to be exchanged in order to re-establish the routing state of affected nodes.

2.1.5. Conclusion

The previously presented DHTs show that realising a distributed hash table can be done in many different ways while still maintaining the goal of efficient resource look-ups (in terms of visited nodes) with only a small fraction of the system known to a node. CAN puts its resources and peers in a d-dimensional coordinate space and requires each node to maintain $2d$ neighbour links. By using simple forwarding based on the geometric distance of a node to a target resource, it achieves average routing path lengths of $(d/4)(n^{1/d})$ hops and allows simple request load balancing by routing requests through different nodes in the direction of a target. Pastry and Chord both map their nodes and resources to a one-dimensional circular name space with addresses between 0 and $2^m - 1$. Pastry further sub-divides those identifiers into levels of b bits and requires a node to maintain a routing table of size $\lceil \log_{2^b}(n) \rceil \cdot (2^b - 1)$ as well as a namespace and neighbourhood set of fixed sizes. Using prefix-routing the average number of hops during routing is $\lceil \log_{2^b}(n) \rceil$ with a worst-case of $O(n)$. Additionally Pastry allows replication on a per-resource level which can be set at a resource's insertion and also provides request load balancing. Chord on the other hand uses finger tables of size m to point to nodes responsible for an exponentially increasing key distance from the nodes' own keys and achieves routing path lengths of $O(\log(n))$ with high probability.

Except for the request load balancing provided by CAN and Pastry, those three DHTs do not implement any explicit load balancing algorithms to balance the load among the nodes but instead rely on the (passive) load balancing provided by consistent hashing with the help of virtual servers (Chord). Without virtual servers this results in each node being responsible for an $O((\log(n) + 1) \cdot 1/n)$ fraction of the available resources which is brought down to $O(1/n)$ using virtual servers. Further improvements (even without virtual servers) are possible using explicit load balancing algorithms (ref. Section 2.4).

Despite their differences, those DHTs all provide a common set of functionality which allows them to be deployed to the needs of the user and be replaced by one another. However, they only allow simple requests like retrieving resources for a set of single keys and lack support for further extensions such as range-queries covered below.

2.2. DHTs with Range Queries

One way of implementing range queries is to build them on top of ordinary DHTs such as the ones described above. Multiple dimensions, i.e. possible attributes in range queries, would be reduced to one dimension using space-filling curves and then split into several

ranges which each serve as a single key that is then stored in the DHT [9, 17, 22] (also see Figure 2.3). Such partition needs to be implemented with care because too few fractions lead to poor load balance and too many will increase look-up costs as several of them may need to be retrieved in order to answer a single range query. The same happens for large multi-dimensional range queries. Another disadvantage is the increased maintenance cost this additional layer imposes on the network.

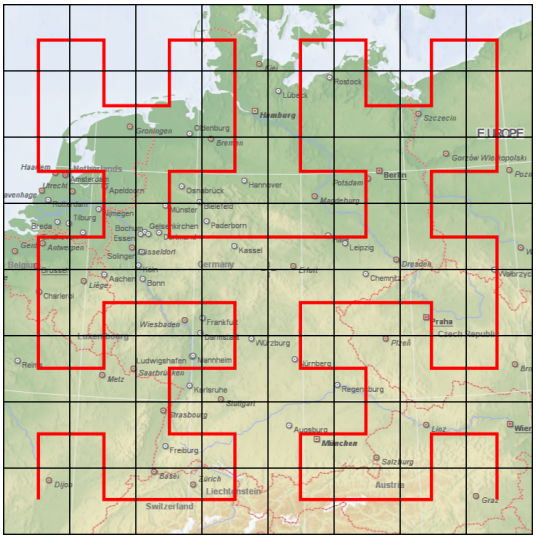


Figure 2.3.: *Example of using a space-filling curve: Patches group coordinates and are then mapped to one dimension according to the progression of the Hilbert curve (approximation level 4) drawn in red. (map: Marble [2], curve: Wikipedia [6])*

Because of these disadvantages, specific DHTs were created that support range queries out of the box. Mercury [12] and Chord$^{\#}$ [39], for example, use key-order-preserving hash functions which allow significantly lower overhead in design complexity and better query performance (in terms of visited nodes per range query) compared to the method depicted above. The following sections will give a short overview of those two implementations which work a bit differently than ordinary DHTs and need to deal with a new set of problems, e.g. significant load imbalance based on the key distribution of their resources.

2.2.1. Mercury

Resources in Mercury [12] consist of a list of $(attribute, value)$ pairs with attributes supporting `int`, `char`, `float` and `string` data types. Queries can be created using multiple filters on given attributes which together form a conjunction (disjunctions of filters need to be emulated by issuing a single query for each of them).

Mercury partitions its peers into several *attribute hubs* H_a each denoting a group

that is responsible for a single attribute a. The number of hubs should be reasonably small but nodes can be part of several such hubs. Within a single hub H_a each node is responsible for a contiguous range of an attribute a and together the nodes form a circular overlay based on that attribute. This range is assigned to a node when it joins the network. Furthermore, resource r is stored at each node that is responsible for any of its attributes in any hub and is thus sent to every hub H_a with $a \in r$ when it is inserted into the network.

Processing a query first involves a selection of a (single) hub through which the message is routed. Within that hub the query is processed by all nodes which have potential matches. Selecting a hub is therefore crucial for getting a good routing performance and should be done by evaluating the selectivity of each filter of a query. In-hub-routing works by sending the query to the node that is responsible for the first value of the hub's attribute and forwarding it to subsequent nodes still within the range of the query. For that nodes store links to their predecessor and successor nodes within each hub and links to (at least) one node in every other hub. For better robustness to node failures, nodes could alternatively store a (small) number of those links instead of just one.

Similarly to Chord this system would result in $O(n)$, $n = \|P\|$ hops required for processing a query. To provide more efficient routing, k long-distance links are added to the nodes' state (also see the example given in Figure 2.4). Note that k could be different for each node but let's assume that each node contains no more than $2k$ of such links whose construction is given as follows. For each link l_i a node p responsible for the range $[a_l, a_r)$ of attribute a draws a number $x \in [1/n, 1] = J$ using the harmonic probability distribution function $p(x) = (n \cdot \log(x))^{-1}$ for $x \in J$ and stores the node that is responsible for the value $(a_r + (a_{max} - a_{min}) \cdot x)$ within H_a. Queries are then forwarded to the node among the long-distance links that minimises the (clockwise) distance to the requested attribute value. Assuming node ranges are uniform, a node responsible for the first value in a given range can be reached with $O\left(\log^2(n) \cdot 1/k\right)$ hops (including the first hop which decides the hub to route in).

Constructing $O(\log(n))$ long-distance links in that manner also enables Mercury to allow uniform random sampling of nodes which is used to gather histograms of system statistics, e.g. load distribution, node-counts and so on. This provides information needed to create the links at all (the number of nodes) and may also be used for implementing a load balancing scheme. Mercury uses a load balancing algorithm similar to the one presented by Karger and Ruhl [30]. Both are described in Section 2.4.2.

Nodes joining Mercury need to complete the following steps:

1. Find a Mercury node p_1 by using some external mechanism.

2. Obtain a list of representatives of each hub by querying p_1.

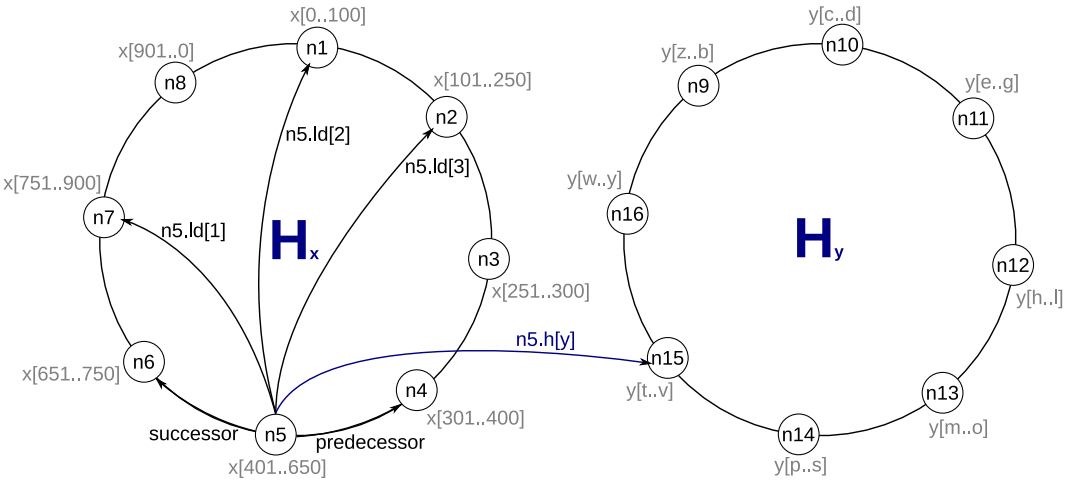


Figure 2.4.: *Mercury network with Hubs H_x and H_y showing node n5's predecessor, successor, cross-hub (h) and $k = 3$ long-distance (ld) links and each node's responsibilities inside its hub.*

3. (Randomly) choose a hub to join at, contact one of its members (p_2) and become its predecessor taking half of its values.

4. Copy routing state of p_2, create its own long-distance links and get hub representatives different to the ones from p_2.

When nodes fail or leave the network, repairing successor and predecessor links is done by using the successor and predecessor link lists mentioned before. Long-distance link failures can be repaired by simply creating new links. Alternatively (and to deal with many link failures) nodes can periodically re-create all links when the number of nodes in the system changes substantially. Finally repairing cross-hub links can be achieved by using a backup link, asking a neighbouring node for its links or (if both fails) using the external mechanism used for node joins.

2.2.2. Chord# / Scalaris

Chord# [39] is a variation of the Chord protocol described in Section 2.1.4 and has been implemented in Scalaris [3]. In its basic form it supports one-dimensional range queries but can also be extended for multiple dimensions as described in [39]. It derives from Chord by removing consistent hashing and instead using a key-order preserving hash function to map resources to the identifier space, e.g. by storing the keys in lexicographical order. Nodes are placed at such points of the identifier space that achieve well-enough load distribution. This placement is managed by an explicit load balancing mechanism which constantly changes the nodes' responsibilities according to the current system load. Schütt et al. suggest to use the algorithm presented by Karger and Ruhl [30]

but any of the algorithms described in Section 2.4 is suitable.

In order to keep the routing performance (number of hops required to reach a node responsible for a random resource) at $O(\log(n)), n = \|P\|$, the finger table is constructed differently and operates in the node space rather than the key space. The first finger is the node's successor as in Chord and the i'th finger is created by asking the node at finger $(i-1)$ for its $(i-1)$'th finger. This step is repeated as long as fingers point to nodes succeeding the previous finger and not exceeding the current node. The resulting finger table then contains at most $\lceil \log(n) \rceil$ fingers with the longest finger pointing half-way around the node circle, the second longest quarter-way, and so on. It is also guaranteed that no two fingers point to the same node (refer to Figure 2.5 for an example).

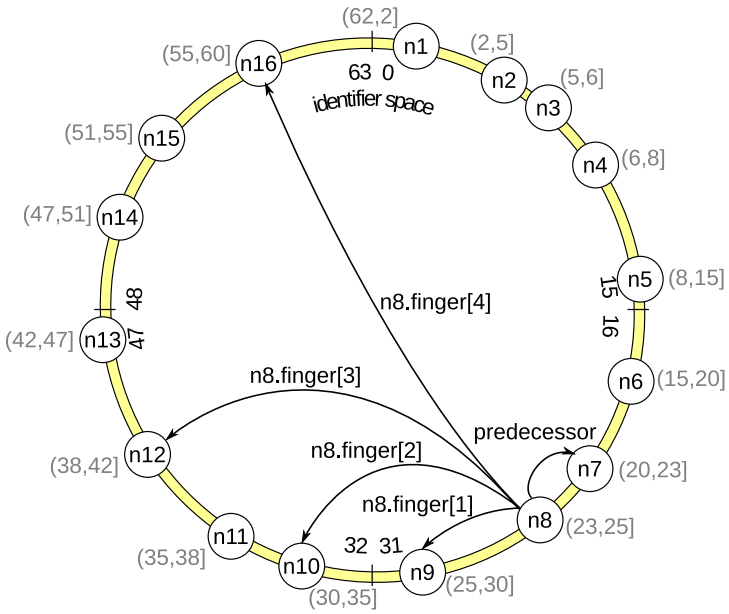


Figure 2.5.: *Chord$^{\#}$ ring with 2^6 possible IDs and 16 nodes distributed to balance a distribution of resources with hot spots around 6, 24 and 36 (node responsibilities in grey).*

Although the routing algorithm stays the same as in Chord, the number of hops required to reach a desired node is now *guaranteed* to be $O(\log(n))$. This is achieved because fingers in Chord$^{\#}$ definitely decrease the distance to a target node by factor 2 each routing step and not just with high probability. Also (re-)building the finger table requires only $O(\log(n))$ messages compared to $O(\log^2(n))$ in Chord.

2.2.3. Conclusion

The aforementioned DHTs show that support for range queries can be achieved with little less or no overhead to ordinary DHTs. In fact, Chord$^{\#}$ even improves Chord's performance by guaranteeing routing performance of $O(\log(n))$ hops and changing only

little compared to Chord. Mercury supports multi-dimensional range queries and uses a so-called *Hub* for each of a resource's attributes. Using k long-distance links it reaches a designated node within $O\left(\log^2(n) \cdot 1/k\right)$ hops. Furthermore by the way those links are generated, Mercury supports random sampling of nodes which it uses to gather system statistics such as estimates of the average load and the number of nodes.

Both DHTs may use arbitrary load balancing algorithms in order to even out the load imbalance that is inherent in the use of order-preserving hash functions. Chord$^{\#}$ suggests an algorithm proposed by Karger and Ruhl [30] while Mercury implements its own variant of this algorithm. Refer to Section 2.4 for a description of those algorithms.

2.3. Gossiping

Gossip algorithms can be advantageous for Distributed Hash Tables in several ways. They can for example provide another way of learning random nodes and can be used to adapt the topology of the overlay network to changes. Both is provided by the Cyclon framework [43]. They can also be used to aggregate (global) information with high confidence and low overhead which is of more interest here. Such information includes approximations of values like the minimum, maximum and average load, network size, variance and standard deviation [28].

A generic proactive algorithm calculating those values could for example work by letting each node periodically select another node to exchange information about its local estimate of the desired attribute. Both nodes update their state according to an aggregation-specific *update* function that improves a node's estimate with the help of the other node's estimate. In case of average load computation the nodes could start with local estimates such as their own load. The update function would receive the two estimates avg_p and avg_q of the nodes p and q and both nodes will update their local estimates to $(avg_p + avg_q)/2$ thus achieving a better estimate. Note that the sum of all estimates remains the same as the sum of all nodes' loads and can thus be used to further aggregate the average load the same way. Similarly the minimum and maximum can be calculated by returning $\min(avg_p, avg_q)$ and $\max(avg_p, avg_q)$ respectively and can also be used to collect information about the k minimum/maximum loads (and the nodes holding those values). The variance can be computed by calculating the averages of the nodes' loads and their squares since $\mathrm{Var}(l) = \mathrm{avg}(l^2) - \mathrm{avg}(l)^2$, same for the standard deviation $\sigma_l = \sqrt{\mathrm{Var}(l)}$.

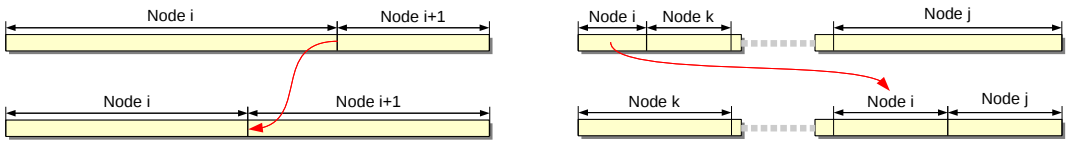
This method provides exponential convergence to the desired value at each node, but best performance can only be guaranteed if the node selection is truly random, e.g. uniform. Nevertheless, this protocol also works by (randomly) selecting nodes from a list of neighbours that is based on the topology of its network or by making random

walks. Experiments conducted in [28] show that the more uniform the random sampling is the faster this algorithm converges.

The following chapters will make use of gossiping algorithms only to retrieve the aforementioned estimates of global information in order to improve load balancing. For an overview of further uses of such algorithms on structured overlay networks refer to [23] and the papers referred there.

2.4. Load Balancing in DHTs

As depicted above, some distributed hash tables include (simple) load balancing techniques with some even being immanent in their design, e.g. by using consistent hashing. Their ability to balance load among the system however varies greatly and can generally be improved by deploying a different load balancing algorithm that suits a specific need. That might include a better partition of the address space among the nodes or, more generally, a better balance of an arbitrary *load* like the number of stored resources, a machine's workload including computing power or bandwidth or any other. Also, although not explicitly considered here, one might include node heterogeneity in any balance decision. Other DHTs, in particular those supporting range queries, heavily rely on explicit load balancing mechanisms because the distribution of the stored resources is retained and may be highly skewed.



(a) Slide (b) Jump

Figure 2.6.: *Supported load balancing operations in arbitrary distributed hash tables.*

Note that generic load balancing algorithms can only make use of techniques supported by every DHT and cannot use features specific to a single one. It is for example possible to adjust the responsibility of two neighbouring nodes so that one node takes some resources or responsibilities off of the other. This process is called *sliding* and may be supported directly by the DHT or by removing one of the two nodes and inserting it at an identifier that will result in the desired behaviour. The second generic load balancing primitive is *jumping*, that is a node leaves its current position dropping off all its load and responsibilities to its successor and joining somewhere else in order to take off some of other node's load. Examples for both are presented in Figure 2.6 for a ring-based DHT also showing the changes of every affected node.

The following sections will present several such load balancing algorithms which are

structured as follows. Section 2.4.1 will describe algorithms that try to balance the amount of identifier space each node is responsible for, followed by algorithms trying to balance the actual number of resources in Section 2.4.2. Load balancing algorithms relying on virtual servers or using replication are covered in sections 2.4.3 and 2.4.4. Note that some algorithms' classifications can be ambiguous in which case their main aspects determine the section they are presented in. Further categorisations could be made in order to differentiate between passive and active algorithms, i.e. those that only act on node or item inserts or deletes and those that actively probe the network every once in a while to search for nodes to balance. This additional classification is included in the overview of all presented algorithms given in Section 2.4.5

Note that (until otherwise stated) algorithm descriptions in the following sections will be restricted to ring-based DHTs like Chord which can be done without loss of generality (special care only needs to be taken with the multiple dimensions of a CAN network).

2.4.1. Address-Based Load Balancing

Address-based load balancing algorithms aim at partitioning the identifier space uniformly among the participating nodes so that each node is responsible for an equal range of identifiers. This is mostly useful for DHTs using consistent hashing (see Section 2.1.1) where resource identifiers are spread uniformly among the identifier space as well and do not follow a particular distribution. Recall that using uniform and independent hash functions for both nodes and resources still results in an $O(\log(n))$ imbalance. Using virtual servers reduces this imbalance but introduces higher maintenance costs due to the increased number of connections each real host manages. The following sections will describe several address-based load balancing algorithms that will try to reduce the imbalance without using virtual servers.

Karger and Ruhl

Karger and Ruhl's address-space balancing scheme [30] first adds an ordering to addresses of the form $\frac{x}{y} = \frac{2b+1}{2^a}$ in the circular identifier interval $I = [0, 1]$ such that $\frac{x}{y} < \frac{x'}{y'} \Leftrightarrow (y < y')$ *or* $(y = y'$ *and* $x < x')$. Equation 2.1 shows the order of some addresses with this specification.

$$0 = 1 < \frac{1}{2} < \frac{1}{4} < \frac{3}{4} < \frac{1}{8} < \frac{3}{8} < \frac{5}{8} < \frac{7}{8} < \frac{1}{16} < \frac{3}{16} < \frac{5}{16} < \frac{7}{16} \ldots \tag{2.1}$$

Secondly, each node maintains a set of $O(\log(n))$ potential positions it can place itself at (solely dependent on the node itself, e.g. on its IP address). It now occasionally checks the address range between each such position and the succeeding active node on

the ring and places itself at the position with the range that covers the smallest address under the given ordering. It can be observed that nodes place themselves at positions close to all small addresses (under the given ordering) which distributes them nearly uniformly among the address space (each node is responsible for an $O(1/n)$ fraction) with high probability thus achieving a ratio between the largest and smallest interval of $O(1)$.

Bienkowski et al.

Bienkowski et al. [13] give a load balancing algorithm for ring-based DHTs which estimates the total number of nodes by having each node maintain an additional connection to a random position in the ring (a *marker*) and count the number of markers that fall into the interval of the node itself and some of its successors. Let i and m be the length of the encountered interval and the number of encountered markers respectively, then a node continues to add the succeeding node's data (interval length and number of markers) as long as $m < \log(1/i)$. At the end, i is decreased so that $m = \log(1/i)$ using the information of the last visited node. Let n_i be the solution of $\log(x) - \log(\log(x)) = \log(1/i)$. It follows that with high probability n_i is within constant factors of the real number of nodes n and there are global constants v, u so that $v \cdot n_i \leq n \leq u \cdot n_i$.

Bienkowski et al. now use these values to define three categories of intervals: *short* intervals of length at most $\frac{4}{v \cdot n_i}$, *long* intervals of length at least $\frac{12 \cdot u}{v^2 \cdot n_i}$ and *middle* intervals of lengths in between. Note that the given interval definitions have been chosen so that middle and long intervals have lengths of at least $4/n$ and halving long intervals never creates short intervals.

In the algorithm, nodes with short intervals whose predecessors also cover short intervals try to contact nodes with long intervals with probability $1/2$ and move to a position which splits those nodes' intervals into halves. The search for suitable partner nodes starts at a random position on the ring and continues to look at up to $6 \cdot \log(u \cdot n_i)$ of the succeeding nodes. If routing messages to random destinations is of complexity R then this algorithm achieves a constant ratio between the largest and smallest interval in $O(1)$ rounds with each node incurring a communication cost of $O(R + \log(n))$ per round.

Manku et al.

Manku [34] describes an algorithm for choosing appropriate node identifiers upon insertion by contacting the node responsible for a random identifier as well as $c \cdot \log(n)$ of its neighbours (using a small constant c) and selecting an identifier so that the largest covered interval among those nodes is split into halves. Node departures are handled

similarly by moving at most one node of the $c \cdot \log(n)$ neighbours of the departing node taking into account the intervals they cover. This algorithm achieves a ratio between the largest and smallest node interval of at most 4 using $\Theta(R + \log(n))$ messages, R being the number of messages needed to contact a random node of the used DHT, and can be tuned to achieve a ratio of $(1 + \epsilon), \epsilon > 0$ at the cost of re-assigning $O(1/\epsilon)$ nodes instead of one node and an increased message cost.

Later Kenthapadi and Manku [31] generalise the scheme of using random and local probes describing algorithms that conduct r random probes each followed by a local probe discovering v of its neighbours and selecting an identifier to split the largest of those intervals. They state that with $r \cdot v \geq c \cdot \log(n)$ the ratio between the largest and smallest interval is at most 8 with high probability where c is a small constant. n can be estimated from the first random probe to ensure that condition. Such schemes use $\Theta(r \cdot R + v)$ messages which allows fine-tuning of the number of local and random probes with respect to the message cost.

Giakkoupis and Hadzilacos

Giakkoupis and Hadzilacos [24] employ the *power of multiple random choices* paradigm to create a load balancing algorithm they extend to support heterogeneous nodes. Their algorithm ensures that each key interval a node is responsible for has a length of $1/2^d$ for some constant $d \in \mathbb{N}$ and its endpoints are integer multiples of its length. It adjusts node responsibilities only at *join* and *leave* operations and works as follows: Nodes joining the system first contact the nodes responsible for a logarithmic (in system size) number of points selected uniformly and independently at random. If $1/2^d$ is the length of the interval the node contacts to join the DHT, then $\lceil a_{join} \cdot d + b_{join} \rceil$ identifiers are looked up for some positive system-wide parameters a_{join} and b_{join}. The node then splits the largest interval in halves. Similar to this nodes leaving the system will again issue a logarithmic number of requests for nodes ($\lceil a_{leave} \cdot (d + 1) + b_{leave} \rceil$ identifiers if $1/2^d$ is the length of the node's interval), merge the smallest interval with its adjacent interval and assign the leaving node's interval to the node removed due to this merge. As in the algorithm by Manku, a ratio between the largest and smallest node interval of at most 4 is reached but $O(R \cdot \log(n))$ messages need to be exchanged.

In the weighted version of the protocol nodes have an associated *weight* (an integer power of 2 with a system-wide upper bound W) proportional to their power, e.g. computing power, bandwidth or storage capacity, and are organised in groups containing adjacent nodes. The same technique as in the unweighted version is now used to balance the intervals of those groups while an additional *group management protocol* handles the balance inside a group and splits or merges groups in order to keep the sum of all weights of its nodes between W and $(2W - 1)$. Therefore the ratio between the largest and smal-

lest group interval is 4 and the overall balance depends on this protocol. It could either achieve a perfect balance inside each group, requiring that up to all its nodes change their responsibilities, or settle for only a few changes to the nodes' associations and achieve an adequate ratio of its nodes' largest and smallest intervals.

2.4.2. Item-Based Load Balancing

Item-based load balancing algorithms try to balance the actual distribution of the resources among the nodes and do not rely on a uniform resource distribution in the identifier space. This makes them particularly suitable for range-queriable DHTs that use order-preserving hash functions. Exemplary distributions of resources that result from an alphabetical storage can be seen in Figure 1.1 on page 9 and Figure 4.1 on page 55.

Although some of the depicted address-based load balancing algorithms may be extended to support item-based load balancing as well, e.g the random and local probes used by Manku et al. and the power of multiple random choices paradigm incorporated by Giakkoupis and Hadzilacos could use the node's actual load instead of the covered address space, there are also some specific algorithms handling this category of load balancing which are introduced in the following sections. Also note that item-based load balancing generally allows arbitrary definitions of *load* that could for example take into account a node's capacity, a resource's size and popularity, network latency and more or combine any of those.

Karger and Ruhl

Karger and Ruhl's item balancing algorithm [30] is a randomised load balancing algorithm that lets each node n_i occasionally contact another node n_j at random and tries to balance those nodes if their load differs by at least a factor of $0 < \epsilon < 1/4$ which is a system-wide constant. It uses the two generic load balancing primitives *slide* and *jump* introduced above to adjust how an interval is split between two neighbouring nodes and to move a node in order to capture some other node's resources respectively. Note that when node n_i is removed due to such a move, n_i's successor n_{i+1} gets all of n_i's resources which can be a severe burden for n_{i+1}.

After n_i and n_j establish contact, Karger and Ruhl's algorithm first checks whether n_j is n_i's successor in which case the two can be directly balanced. Otherwise it tries to balance the most loaded node of the three involved nodes n_i, n_j and n_{j+1} (lines 16-21 in listing 2.1) by either balancing n_j with its successor or moving n_j to a position that would result in n_j receiving half of n_i's resources.

Karger and Ruhl prove that if each node contacts $\Omega(\log(n))$ random nodes (n being

```
1  karger_item(DHT d, double ε) {
2   foreach (Node n_i ∈ d) {
3    Node n_j = d.getRandomNodeExcept(n_i); // get another random node
4    if (load(n_i) ≤ ε · load(n_j)) {
5     balance(n_j, n_i);
6    } else if (load(n_j) ≤ ε · load(n_i)) {
7     balance(n_i, n_j);
8    }
9   }
10 }
11
12 balance(Node n_i, Node n_j) { // load(n_i) > load(n_j)
13  if (n_i == n_{j+1}) {
14   slide(n_i, n_j); // equalise load of n_i, n_j
15  } else {
16   if (load(n_{j+1}) > load(n_i)) {
17    slide(n_j, n_{j+1}); // equalise load of n_j, n_{j+1}
18   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
19    jump(n_j, n_i); // move n_j to take half of n_i's resources
20                    // n_j's resources are moved to n_{j+1}
21   }
22  }
23 }
```

Listing 2.1: *Item-based load balancing by Karger and Ruhl.*

the number of nodes in the system, l_{avg} the nodes' average load), this will result in every node having a load of at most $(16/\epsilon) \cdot l_{avg}$ with high probability. Contacting another $\Omega(\log(n))$ nodes will bring all nodes' loads to at least $(\epsilon/16) \cdot l_{avg}$. Increasing the number of nodes to contact when searching for a node to balance with will increase the probability of those bounds.

Bharambe et al. (Mercury)

In Mercury [12] (ref. Section 2.2.1) a variant of Karger and Ruhl's algorithm is used based on the histograms the DHT provides. Firstly, the *local load* $l_{local}(n_i)$ of a node n_i is defined to be the average load of itself, its successor and its predecessor. Secondly Mercury's histograms are used to retrieve an estimate of the system's average load l_{avg}. A node n_i is then said to be *light* if $l_{local}(n_i)/l_{avg} < 1/\alpha$ and *heavy* if this ratio is greater than α. This ensures that light nodes have only light neighbours with high probability. If a light node's neighbour is heavy, the two nodes need to balance their load. Additionally heavy nodes probe the system for light nodes which (if found) leave their current position and move to the heavy node in order to take some of its resources. Listing 2.2 shows an

implementation of this algorithm.

```
1  mercury(DHT d, double α) {
2   foreach (Node n_i ∈ d) {
3    if (isLight(n_i)) {
4     if (isHeavy(n_{i+1})) {
5      slide(n_i, n_{i+1}); // equalise load of n_i, n_{i+1}
6     } else if (isHeavy(n_{i-1})) {
7      slide(n_{i-1}, n_i); // equalise load of n_{i-1}, n_i
8     }
9    } else if (isHeavy(n_i)) {
10    Node n_j = d.getRandomNodeExcept(n_i); //get another random node
11    if (isLight(n_j)) {
12     // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
13     Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
14     if (n_j.isNeighbourOf(n'_i)) {
15      slide(n'_i, n_j); // equalise load of n'_i, n_j
16     } else if (n'_i ≠ n_j) {
17      jump(n_j, n'_i); // move n_j to take half of n'_i's resources
18                       // n_j's resources are moved to n_{j+1}
19     }
20    }
21   }
22  }
23 }
```

Listing 2.2: *Example implementation of the load balancing algorithm by Bharambe et al.*

Provided that $\alpha \geq \sqrt{2}$, the ratio between the highest and average load (as well as the ratio between the average and lowest load) is bound by a factor of α. Also note that by tolerating a small skew, i.e. by setting α appropriately, unnecessary item movements due to balance operations during load oscillations can be prevented or at least reduced.

Ganesan et al.

Ganesan et al. [21] describe a load balancing algorithm for range-partitioned data that can also be applied to DHTs. It tries to balance load among nodes whenever the load at a node increases or decreases by a certain factor δ and is called the *Threshold Algorithm*. More precisely they define a sequence of thresholds $T_i = \lfloor c \cdot \delta^i \rfloor, i \geq 1$ for some constant $c > 0$ and whenever a node's load increases to a $(T_m + 1)$ the algorithm tries to adjust its load as follows. If one of its neighbours has a load of at most T_{m-1} the node balances its load with the neighbour with the smallest load. Otherwise the node (let it be n) searches for the least-loaded node n_k and, if n_k's load is at most T_{m-2}, tells it to leave its current position (moving all its items to its successor) in order to take over half of

n's load. This might result in further recursive invocations of this adjustment at the affected nodes. See listing 2.3 for a complete description of the load balancing procedure performed if such interval is exceeded due to a resource's insertion.

```
1  adjustLoad(Node n_i) { // let load(n_i) ∈ (T_m, T_{m+1}]
2   Node n_j = minLoadedNeighbour(n_i); // least loaded neighbour
3   if (load(n_j) ≤ T_{m-1}) { // adjust neighbours
4    slide(n_i, n_j); // equalise load of n_i, n_j
5    adjustLoad(n_j);
6    adjustLoad(n_i);
7   } else {
8    Node n_k = findLeastLoadedNode();
9    if (load(n_k) ≤ T_{m-2}) {
10    jump(n_k, n_i); // move n_k to take half of n_i's resources
11                    // n_k's resources are moved to n_{k+1}
12    adjustLoad(n_{k+1}); // adjust load of n_k's old successor
13   }
14  }
15 }
```

Listing 2.3: *Method used to adjust load in the* Threshold Algorithm *by Ganesan et al. when a resource insertion results in node n_i's load exceeding a threshold.*

Load adjustments due to resource deletions are handled accordingly: if the node's load drops below a threshold T_m it tries to balance with the highest-loaded neighbour with a load of at least T_{m+1} or tries to move to the highest-loaded node of the system to take half of its elements if that node has a load of at least T_{m+2}. This definition reduces too hasty load balancing operations - and thus resource movements - in case of oscillating loads around the threshold T_m.

Ganesan et al. show that each $\delta \geq \Phi := (\sqrt{5}+1)/2 \approx 1.62$ can be chosen achieving a ratio of δ^3 between the highest and lowest load. They also state that finding the least and most loaded nodes (line 8 of the *adjustLoad* method in listing 2.3) is not necessary for their results to hold true. Instead the node could move to any node that violates a *GlobalBalance* condition, i.e. for node n_i with a load in the interval $(T_{r-1}, T_r]$ find a node n_k with a load not in the interval $(T_{r-3}, T_{r+2}]$.

Aspnes et al.

Aspnes et al. [10] describe a load balancing algorithm for range-queriable data structures that uses arbitrary definitions of *load* and groups keys into buckets with each peer storing some of them (similar to virtual servers). A "free-list" of buckets is maintained, e.g. by using a separate overlay network or storing all such buckets near a fixed key, and is used to take load off of heavily loaded nodes. Buckets are further divided into *closed* and *open*

buckets depending on a certain threshold based on their load. They are furthermore partitioned into groups of two (closed, open) or three buckets (closed, open, closed) and retain this structure by transferring resources as needed, e.g. when a resource is to be inserted into a closed bucket, it moves one of its resources to the neighbouring open bucket and accepts the new resource. If an insertion makes an open bucket closed, one of the buckets from the free list is taken and inserted accordingly (this may transform a group of two into a group of three or a group of three into two groups of two buckets). Deleting resources works similarly and may lead to empty buckets which are returned to the free list during re-structuring.

A bucket's size can be changed by adjusting the threshold that classifies a bucket as closed and requires re-structuring the bucket groups. Such changes will be enforced when the overall system load increases or decreases sufficiently. A centralised version of this algorithm can for example double this threshold when the free list becomes empty and halve it when half the number of nodes is in that list. This results in a worst-case maximum load of 4 times the average load but requires a global controller to adjust the bucket size. It also creates heavy load movements during such migrations.

Aspnes et al. also describe a distributed version which resizes the buckets based on an estimate of the system's average load (gained by gossiping techniques) and prevents simultaneous resource migrations. In order to achieve the latter, buckets are again organised into groups of two pairs (closed, open) or one triple (closed, open, closed). Let M be the load of a closed bucket and $1/4 < e_g < 1/2$, $1/8 < c_g < 1/4$ be random expansion and congestion thresholds for a group g of buckets. g performs localised expansion (doubling its threshold to classify nodes as closed) if its estimated average load l satisfies $l > e_g \cdot M$ and performs localised contraction (halving the threshold) if $l < c_g \cdot M$ allowing each bucket group to migrate separately.

Charpentier et al.

An alternative to using gossiping to gather approximations of global knowledge is to use cooperative mobile agents. Those agents, while moving from one node to another, could also be used to initiate load balancing operations. Charpentier et al. [16] use that technique which is solely sketched here to present a rather different approach using techniques from the research field of mobile agents. In their algorithm agents gather approximations of the system's average load. They first start in an initialisation phase which tries to estimate the average load to a certain degree of accuracy. Several agents can be supplied and cooperate with each other to improve their estimates and speed-up their initialisation phase by exchanging their data. In their second phase agents order nodes with loads higher than their calculated average to migrate some of their resources to either or both of their neighbours thus achieving some load balancing.

2.4.3. Virtual-Server-Based Load Balancing

Several research papers focus on DHTs that use multiple virtual servers on each real peer and balance load by moving those virtual servers. As depicted above, using consistent hashing and deploying $\Omega(\log(n))$ virtual servers (in a system with n real peers) randomly along the identifier space will lead to each peer being responsible for an $O(1/n)$ fraction of the stored resources (ref. Section 2.1.1). Another advantage is that each peer can easily adjust its load by moving some of its virtual servers to any other peer instead of just being able to shed load to its neighbours or move itself. This however comes at the cost of maintaining $\Omega(\log(n))$ additional network connections, an increased number of routing hops while looking up random resources as well as increased churn on node failures which are generally the reasons why the use of virtual servers is not preferred. Nevertheless the following sections show some algorithms that make use of this technique as it has been of interest in past research and still is.

Rao et al.

Rao et al. [36] present three different load balancing schemes that try to move virtual servers from heavily to lightly loaded nodes. *Load* in their case can be any single resource, e.g. storage, bandwidth or CPU capacity, so this algorithm could also be classified as an item-based algorithm. Nodes are considered heavy if their current load exceeds their target load and are otherwise light. Balancing a heavy peer p_h with a light peer p_l will move the virtual server v to p_l that does not make p_l heavy and is the lightest virtual server making p_h light or the heaviest virtual server in case p_h cannot be made light that way. The three schemes now differ in how heavy and light nodes are matched in order to start balance operations.

The *One-to-One Scheme* lets light nodes occasionally probe other nodes at random and virtual servers are transferred if the probed node is heavy. Letting only light nodes try to contact heavy nodes (instead of heavy nodes trying to contact light nodes) will not introduce additional workload on heavy nodes and will therefore not increase the risk of highly loaded systems getting overloaded or trashed due to unnecessary and unsuccessful probes. The second scheme implements a *One-to-Many* matching technique by letting light nodes register with one of d system-wide *directories* which are also maintained by the DHT, e.g. by storing a directory at the node responsible for its key. Heavy nodes may now look at such directories and pick the least loaded node to shed some of their virtual servers to. This scheme is now extended by registering heavy nodes with those directories as well and letting the node that stores a directory occasionally match heavy and light nodes in a *Many-to-Many* fashion to optimise the load balance even more.

Rao et al. [36] now analyse their algorithms in a static setting, i.e. an initial load is

distributed among a fixed number of nodes. Neither new resources are added or deleted, nor are nodes. Using the total number of moved load and the number of probes the algorithm needs to achieve a state with only light nodes, they conclude that the amount of load moved is not dependent on the used scheme although the one-to-one scheme needs more probes in order to succeed. Also in the one-to-many scheme with 16 nodes per directory most heavy nodes succeed in getting light by contacting only one directory.

Godfrey et al.

Godfrey et al. [25] further extend the many-to-many scheme introduced by Rao et al. and analyse it in dynamic networks, that is nodes and resources are dynamically added and removed. In their version, each node initially contacts a random directory and sends its load and capacity information and repeats to send this data to another random directory whenever it transfers any load. Additionally if a node becomes heavy, i.e. its current load is above a given emergency threshold, it contacts its chosen directory and tries to shed load immediately. Directories on the other hand create schedules for transferring load among all their known nodes and execute them periodically. They also perform the immediate balance requests issued by their nodes. Virtual servers are transferred based on a greedy algorithm that moves each heavy node's lightest server to a common pool and matches each of those servers (starting from the heaviest) to the node that suffers the least impact of such transfer relative to its capacity.

Their evaluation shows that using periodic load balancing with an emergency threshold allows selecting significantly larger execution periods and thus achieves better node utilisation with less load movement. They also show that the number of directories deployed does not severely affect the achieved node utilisation and that by using 16 directories node utilisation is only 3% higher than in a centralised approach (1 directory).

Chen and Tsai

In a recent paper, Chen and Tsai [18] try to improve the many-to-many scheme depicted above by introducing ant system heuristics (ASH) to re-assign the virtual servers. Their algorithm, called *Dual-Space Local Search* (DSLS), describes an iterative procedure consisting of three stages:

1. Construct an initial solution for the current iteration using the ASH algorithm.

2. Improve this solution by evaluating further local solutions in its neighbourhood both in terms of load balance and movement cost.

3. Update pheromone trails if a better solution was found.

In step 1, pheromone variables τ_{ij} are used to denote node j's desire of taking virtual server i (with a given maximum desire τ_{max} used if i is already assigned to j). Now with probability p_0 server i is assigned to another node k with enough capacity and a maximal τ_{ik} and with probability $(1 - p_0)$ it is assigned to another node with enough capacity under the probability function $p_{ij} = \frac{\tau_{ij}}{\sum_{k \in P} \tau_{ik}}, \forall j \in P$. p_0 thus allows fine-tuning the algorithm between exploiting already found solutions (high values) or explore new variations (low values). If all nodes are fully occupied, a random assignment is used.

Step 2 first tries to find a good solution (if the first step hasn't found any yet) by shifting load from overloaded nodes to their neighbours and eventually further. If a feasible solution is found, i.e. no node exceeds its capacity, a cost-reducing function tries to minimise movement costs by moving servers back to their original location if possible.

Chen and Tsai's evaluation shows that their variation achieves better load balance than the previous implementation of the many-to-many scheme and moves only little more resources than is necessary in order to balance their scenarios. In contrast to the previous work though the number of deployed directories has a severe affect on their algorithm which performs better with less directories.

Ledlie and Seltzer

Ledlie and Seltzer [32] use the multiple random choices paradigm to deploy an algorithm that generates k different verifiable identifiers for each node at which the node can create virtual servers. This algorithm, called *k-Choices*, primarily works at node joins where a node chooses a target load and a maximum number of $k/2$ virtual servers to create. It then creates new virtual servers as long as its target load and the total number of servers are not exceeded. Each such join happens at the one of the still available identifiers which results in the lowest cost in terms of difference between the target and real loads of the two affected nodes. Additionally to this *passive* implementation, k-Choices can also work *actively* and re-select identifiers at any time (not just at node joins) if the change induces a big enough benefit, e.g. a node is over- or underloaded. Also nodes could create more virtual servers or delete some.

During their experiments, Ledlie and Seltzer show that their active algorithm achieves a good load balance for $k = 8$ identifiers. It was still able to do so under highly dynamic networks and with large amounts of skewed load.

Godfrey and Stoica

A technique described by Godfrey and Stoica [26] can reduce the additional cost induced by virtual servers by placing each peer's k virtual servers in a $\Theta(k/n)$ fraction of the identifier space instead of spreading them randomly. This way they can share a single set

of network links. Godfrey and Stoica show that the ratio between highest and average load can be reduced to $(1 + \epsilon)$ for any $\epsilon > 0$ although increasing route lengths and number of links to maintain by only a constant factor (if applied to an arbitrary DHT). They further evaluate an implementation based on Chord using $2 \cdot \log(n)$ virtual servers and achieve a ratio of less than 4.

2.4.4. Load Balancing using Replication

Sometimes replication is not only used to ensure resource availability in such highly dynamic networks as DHTs, but is also suggested to carry out simple load balancing by placing replicas at lightly loaded nodes thus evening out the overall imbalance. However replication is not in the main focus of this work, so the following sections only briefly describe some of the available techniques.

Byers et al.

Byers et al. [14] combine load balancing and replication techniques by making use of the *power of two choices* paradigm. Each resource is assigned d different identifiers using d different hash functions. It is then associated with the k most lightly loaded peers responsible for any of the identifiers. The rest of the peers may store redirection pointers to those storage locations to simplify searches (resulting in increased maintenance costs). Otherwise searches will be carried out for all possible identifiers in parallel.

Xu and Bhuyan

Xu and Bhuyan [44] collect information about the stored resources' access history and use this as their definition of *load* to balance the impact of requests to popular resources among the (possibly heterogeneous) nodes. They first describe a static load distribution algorithm which splits a node's zone into two halves depending on its load (unlike its key range) when a new node arrives. Secondly a dynamic load distribution algorithm steps in when nodes become overloaded and balances their load among neighbouring nodes (possibly including their neighbours as well, and so on). In a final step they specify a replication scheme which enhances their access history with network topology information and replicates resources to peers near a group of peers with the highest request rates. Requests from those peers can now be redirected to the replicas to reduce the access latency and the original node's (access) load.

Pitoura et al.

Pitoura et al. [35] design a DHT which uses replication for efficient range query processing and load balancing of resource accesses. For this they use a so-called *multi-rotation hash*

function that assigns a resource multiple identifiers of an identifier ring. Whenever a node becomes overloaded (in terms of request access load), it will add additional replicas at some of the available identifiers and issue replication requests to its neighbours' resources as well. Replicating whole arcs of the identifier space will improve performance of range-queries that start on a replicated resource location and continue at its original location's successor because its data is replicated to the current location's successor as well. It should also be mentioned that the underlying algorithm of this system can be applied to different DHTs as well.

2.4.5. Conclusion

As can be seen from the previously described algorithms, the goal of balancing *load* in a distributed hash table can be tackled from several angles. There are at first algorithms which try to balance the address-space, i.e. give each node responsibility for an equal part of the identifier space. Those algorithms rely on the uniform distribution of the resources' identifiers in order to give each node an equal amount of resources to store. They are therefore not suitable for range-queriable DHTs that are based on order-preserving hash functions. Another kind of algorithms tries to balance an arbitrarily defined *load* by moving items (resources) and nodes accordingly and are thus called *item-based*. Those algorithms mostly concentrate on *load* being the size of all resources stored by a node or the stored resources' popularity (number of requests). Heterogeneous algorithms set those into relation to a node's capacity. An overview of all presented algorithms and their classifications is given in Table 2.1.

Algorithm	**item/addr.**	**active/passive**	**Notes**
Karger & Ruhl (1) [30]	address-based	active	
Bienkowski et al. [13]	address-based	active	estimates the network's size
Manku [34]	address-based	passive (node)	
Kenthapadi & Manku [31]	address-based	passive (node)	
Giakkoupis & Hadzilacos [24]	address-based	passive (node)	a weighted version exists
Karger & Ruhl (2) [30]	item-based	active	
Bharambe et al. [12]	item-based	active	uses estimate of average load
Ganesan et al. [21]	item-based	passive (item)	uses least/most loaded nodes
Aspnes et al. [10]	item-based	passive (item)	uses estimate of average load
Charpentier et al. [16]	item-based	active	uses mobile agents, average load
Rao et al. [36]	item-based	active	uses virtual servers (VS-based)
Godfrey et al. [25]	item-based	act+pass(item)	VS-based
Chen & Tsai [18]	item-based	act+pass(item)	VS-based, ant system heuristics
Ledlie & Seltzer [32]	item-based	act+pass(node)	VS-based
Godfrey & Stoica [26]	address-based	passive (node)	VS-based
Byers et al. [14]	item-based	passive (item)	uses replication
Xu & Bhuyan [44]	item-based	active	replication, file access history
Pitoura et al. [35]	item-based	active	uses replication

Table 2.1.: *Overview of the presented load balancing algorithms.*

Comparing the algorithms' performance with each other is however not that simple. At first, several metrics for *performance* exist. Some use the ratio between the highest and the average load of the system, some the ratio between the highest and lowest load. Others measure the variance of the fraction of address-space the nodes are responsible for or the deviation from the average load. Secondly values are sometimes given in Landau notation thus hiding constant factors that matter when comparing otherwise equally well performing algorithms. Additionally the costs of achieving a certain performance, e.g. item movements or the number of interchanged messages, are often not mentioned either or are not comparable.

Further impairing the lack of comparisons is the fact that no common test scenarios have been agreed upon which each algorithm can be tested with and that resemble the different use cases of DHTs. For example some papers evaluate their algorithm(s) by simulating them in a static setting, i.e. an initial load is distributed among a fixed number of nodes and neither new resources nor nodes are added or deleted. Those simulations mostly use load distributions that follow a certain probability distribution, e.g. normal or exponential distributions. Other algorithms, especially passive ones, need dynamic simulations as they only act when nodes or items are inserted or deleted. This provides even more flexibility in setting up a test scenario. Furthermore most algorithms also allow some fine-tuning by setting their parameters according to the scenario and the needs of the user.

3. Improving load balancing algorithms with global information

This chapter will present the general concepts of adding estimates of global information to existing load balance algorithms in order to improve their performance. It will start introducing the underlying system model that describes the DHT the algorithms can work on, the operations it supports and any further assumptions. It will then present the (original) algorithms that have been chosen to exemplify how such information is integrated and which affect it has. The final section will cover the changes that were made to those algorithms and the ideas behind them. Detailed descriptions of all mentioned algorithms in pseudo-code can be found in appendix A.

3.1. System Model

Let d be an arbitrary DHT that operates in the identifier space $I = [0, m) \subset \mathbb{N}$ that wraps around at the end and forms a ring. On this ring a *clockwise* direction is defined as going towards increasing keys possibly wrapping around at $m \to 0$. An *interval* $(a, b]$ of this ring includes all keys that are encountered when traversing the ring clockwise starting at (but excluding) a and stopping at b (inclusive). Note that it is possible that $a > b$ in which case $(a, b]$ covers all keys greater than a and less than or equal to b.

Let d consist of n homogeneous nodes (peers) $p \in P$, each being responsible for an interval $(a_p, b_p] \subset I$ so that exactly one peer is responsible for any identifier $id \in I$:

$$\begin{aligned} \forall p_i, p_j \in P,\ p_i \neq p_j : \quad (a_{p_i}, b_{p_i}] \cap (a_{p_j}, b_{p_j}] &= \emptyset \\ \bigcup_{p \in P} (a_p, b_p] &= I \end{aligned}$$

The successor of peer p_i is the peer p_j which is responsible for the following interval, i.e. $a_{p_j} = b_{p_i}$. A predecessor is defined analogously. From the previous definitions follows that there is exactly one successor and predecessor for each peer. Connections between those are maintained so that predecessor and successor pointers form a double-linked list. Additional connections to further peers exist in order to allow efficient routing from any peer of the network to any other.

The DHT stores (arbitrary) resources, i.e. items, that are mapped to I using an order-preserving hash function and stored at the peer which is responsible for their identifier. Each peer has a load $l(p)$ equal to the number of items it stores. The imbalance of the DHT is defined as the standard deviation of its nodes' loads, i.e.

$$\begin{aligned} \mathit{imbalance}(d) &:= \sigma_l \\ &= \sqrt{\frac{\sum_{p\in P}\left(l(p)-\mu\right)^2}{n}}, \quad \mu = \frac{\sum_{p\in P} l(p)}{n} \\ &= \sqrt{\frac{\sum_{p\in P} l(p)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2} \\ &=: \sqrt{\mathrm{avg}(l^2) - \mathrm{avg}(l)^2} \end{aligned}$$

The first goal of a load balancing scheme is to reduce this imbalance. For this, two types of operations are supported: *slide* and *jump*, previously described in Section 2.4. They effectively adjust some nodes' responsibilities which implies item movements. The second goal of a balance algorithm is thus to reduce the number of moved items in order to reach a certain imbalance. There is a trade-off between these two goals because a smaller imbalance can generally only be reached by moving more items.

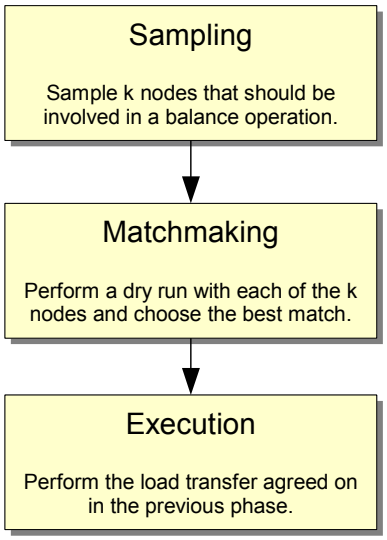
Further operations include the retrieval of a node's successor and predecessor and the ability to get a random node of the whole system. The latter might be natively supported by the DHT or can be implemented by using random walks (suggested in [12]) or gossip algorithms [23] or by generating a random ID and returning the node responsible for it (assuming uniform node responsibilities). Balance algorithms heavily rely on those methods. Some also need to retrieve the node that is responsible for a given key but this is not the case for the algorithms in this section.

Every node also runs a gossiping algorithm that continuously calculates estimates of certain global information such as the system's size, average load and standard deviation (ref. Section 2.3). It is assumed that those estimates are within a certain *error rate* (in percent) of the exact values and that they are re-calculated every once in a while in order to stay within this bound.

During the following sections it will also be assumed that the system is static, i.e. the total number of items and nodes in the system stays constant. Algorithms will thus start their operation on a DHT whose nodes' responsibilities are uniformly distributed among the identifier space. Stored resources follow a certain distribution the algorithms don't know about. The static nature of the system requires that the algorithms actively probe for other nodes to balance with and cannot operate passively.

3.2. Algorithms

In order to analyse the effect of adding estimated global information to existing load balancing schemes, two novel active item-based balance algorithms were chosen and equipped with this knowledge. Those algorithms include the item-balancing scheme introduced by Karger and Ruhl [30] (from here on referred to as "`karger`") and the algorithm used in Mercury [12] ("`mercury`"), both described in Section 2.4.2.



At each execution and for each node, both sample one random node and then decide whether to balance with it and how many items to transfer between the nodes. At most three nodes are involved in such a decision and only they can change their load: the two neighbouring nodes in case of a *slide* operation and in case of a *jump* the moved node, its (original) successor and the node jumped to. Karger and Ruhl also suggest to perform multiple random samples and balance with the best node among them but do not describe how to decide for the best. Here, algorithms with multiple randomly sampled nodes will operate in three phases as presented in the figure to the right. The first step involves sampling a given number of (unique) nodes uniformly at random. With each such candidate node, the balance algorithm is simulated and the best among them (or none) is chosen. It follows the execution of the algorithm with this node (if there is one). Listing 3.1 shows the modified `karger` algorithm and the changes compared to the original scheme described by Karger and Ruhl. Detailed algorithm descriptions in pseudo-code are given in appendix A.

The way the *best match* among the candidates is chosen is crucial for the algorithm. Here it is defined as the node that improves the standard deviation of the system's load the most. If no such node exists, i.e. no balance operation would decrease the standard deviation, no operation is performed. Note that this method will also be used if only one random sample is requested. Also note that only local knowledge is required to take that decision. This can be easily deduced from the following observations.

First recall that $\sigma_l = \sqrt{\frac{\sum_{p\in P} l(p)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2}$ and that the sum of the loads as well as the number of nodes, n, is not changed by performing either *slide* or *jump*. Thus only the first sum needs to be examined. The candidate node that reduces this sum the most will reduce the standard deviation the most. If, for example, the involved nodes' loads $l(p_i), l(p_j)$ change to $l'(p_i), l'(p_j)$, then the change of the sum can be determined as $l'(p_i)^2 + l'(p_j)^2 - l(p_i)^2 - l(p_j)^2$ (similarly with three nodes).

```
1  karger_item(DHT d, double ε, int samples) {
2   foreach (Node n_i ∈ d) {
3    // get k unique random nodes that are not equal to n_i:
4    Nodes candidates = d.getUniqueRandomNodes(n_i, samples);
5    // get best candidate by simulating the algorithm:
6    Node n_j = getBest(d, ε, n_i, candidates);
7    if (d.exists(n_j)) {
8     if (load(n_i) ≤ ε · load(n_j)) {
9      karger_balance(n_j, n_i);
10    } else if (load(n_j) ≤ ε · load(n_i)) {
11     karger_balance(n_i, n_j);
12    }
13   }
14  }
15 }
16
17 karger_balance(Node n_i, Node n_j) { // load(n_i) > load(n_j)
18  if (n_i == n_{j+1}) {
19   slide(n_i, n_j); // equalise load of n_i, n_j
20  } else {
21   if (load(n_{j+1}) > load(n_i)) {
22    slide(n_j, n_{j+1}); // equalise load of n_j, n_{j+1}
23   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
24    jump(n_j, n_i); // move n_j to take half of n_i's resources
25   }
26  }
27 }
```

Listing 3.1: *karger with multiple samples, changes to the original algorithm in red*

3.3. Adding global information

The two previously introduced algorithms are now equipped with additional knowledge about estimated global information. The accurateness of it is unknown to them though. The following sections will introduce each single estimate that is added, how it is used and the ideas behind. A final section will describe how several of those estimates can be used together and will present the idea of self-tuning algorithms.

3.3.1. Average load

The key to reducing the number of moved items is to understand which of them do at least have to be moved in order to reach a state with minimal imbalance. In such a state every node has the same load as the average load among all nodes (only then is $\sigma_l = 0$). This ideal state however does sometimes not exist for a given total amount

of load and a number of nodes, e.g. if the total load is not divisible by the number of nodes. If it exists and arbitrary item movements are possible, an optimal number of load transfers can be reached by moving items only from overloaded nodes (those with a load higher than the average) to underloaded nodes (less load than the average) and never make them overloaded. Also overloaded nodes would only move so many items in order to become balanced. Figure 3.1 shows the load that would get moved in such case. However, it is not possible to apply this algorithm to DHTs following the given model since, for example, items can not be moved arbitrarily.

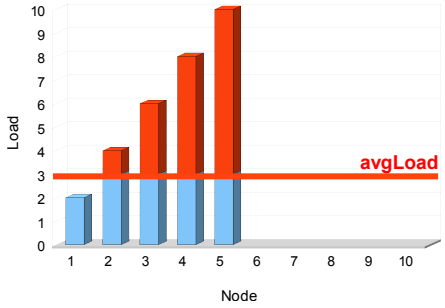


Figure 3.1.: *Load that needs to be moved in order to reach a minimal imbalance (in red).*

The following sections will present some ways, information about the average load can be added to existing load balance algorithms. They try to incorporate some of the ideas of the optimal item movement pictured above.

Variant `avg1`

As one of the first things, one might observe that existing load balancing schemes often try to even out the balance of two neighbouring nodes. This moves load off of overloaded nodes quickly but often results in unnecessary item movements, e.g. transferred items would need to be moved again if the receiving node gets overloaded after the balance operation. Those movements can be reduced if the amount of items that can be moved is lower than or equal to the (estimated) average load.

This variant will thus hook into the algorithms to replace the decision about how much load - and thus items - is to be moved from one node to another. It will never move more than the (estimated) average load from the heavier loaded node to the lighter one. Algorithms with this implementation will have `_avg1` appended to their name. It is the hope that this will reduce the number of item movements while not changing the algorithm's operations too much and thus keeping the balance results the original algorithms expose.

Variant `avg2`

A natural extension to `avg1` would further restrict item movements when balancing neighbouring nodes and never make the light node heavy. Additionally this variant will only move items from heavy nodes (those with a load higher than the average) to light nodes (less load than the average) and limit the number of transferred items to the minimum required in order to make the heavy node balanced (load equal to the average). Algorithms with this implementation will have `_avg2` appended to their name.

While less item movements can be expected from this variation, evaluations will show whether those changes are too invasive in order to reach the same imbalance.

Variant `avg3j`

Both of the previous variations will not not have any influence on the decision whether a node is jumping to another position or not. In `avg2`, for example, this results in some light nodes becoming heavy nonetheless because a jumping node's items are transferred to them. This is not the case in an ideal item movement though.

A third implementation will thus restrict jumping so that this does not happen. If a node makes its successor heavy by jumping or if the successor is already heavy, no balance operations is performed. This variant can be combined with either of the previous two variants (which are not already included!) and appends `_avg3j` to the algorithm's base name. It will reduce unnecessary item movements by restricting jumps. However without jumping, reducing the system's imbalance would take significantly longer and is sometimes not possible when further restrictions apply. It is unclear whether the same imbalance can be reached as with the original algorithm, especially since a jump is normally doing more good than bad.

3.3.2. Standard deviation and system size

As the standard deviation measures the system's imbalance, it can also be used to quantify the quality of a single load balance operation. Assuming the average load does not change and the current standard deviation σ_l, the system's size n and the load changes of the current balance operation are known, the new standard deviation can be calculated as follows. Let p_i and p_j be the peers that changed their load from $l(p_i), l(p_j)$ to $l'(p_i), l'(p_j)$, then:

$$\begin{aligned}\sigma'_l &= \sqrt{\frac{\sum_{p\in P\setminus\{p_i,p_j\}} l(p)^2}{n} + \frac{l'(p_i)^2}{n} + \frac{l'(p_j)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2} \\ &= \sqrt{\sigma_l^2 - \frac{l(p_i)^2}{n} - \frac{l(p_j)^2}{n} + \frac{l'(p_i)^2}{n} + \frac{l'(p_j)^2}{n}}\end{aligned}$$

Variant `stddev2`

This variation will hook into the algorithms to replace the decision about which node from a list of candidates is the best and thus also decides whether to balance or not. Aside from the original algorithm implementations mentioned above, an estimate of the standard deviation of the nodes' loads as well as the system's size is retrieved and used to calculate an estimate of the new standard deviation. Additionally, the algorithms are equipped with one more parameter, `s`, and the best node among some candidates is then determined as the node that, if used in a balance operation, would improve the standard deviation the most and by at least a factor s/n. If no such candidate is available, no balance operation is performed. Algorithms with this implementation will have `_stddev2` appended to their name.

This rationale behind this idea is to omit balance operations that do not improve the overall balance enough compared to the big picture. It will thus tolerate a small skew as balance algorithms mostly already do. This way the algorithm can concentrate on bigger improvements and will (maybe) handle the smaller ones at a later time when they are responsible for the imbalance. With this restriction it can be expected that almost the same balance can be reached while moving fewer items. However special care has to be taken on the decision of the value of s which needs to be adapted to the algorithm this variant is applied to (and possibly also to the load distribution in the DHT, in particular the overall total load). If the algorithm generally only moves a few items, the affect on the imbalance can't be as high as an algorithm moving more items and thus s needs to be lower in order not to block too many operations. This will be evaluated as well.

3.3.3. Combined variants

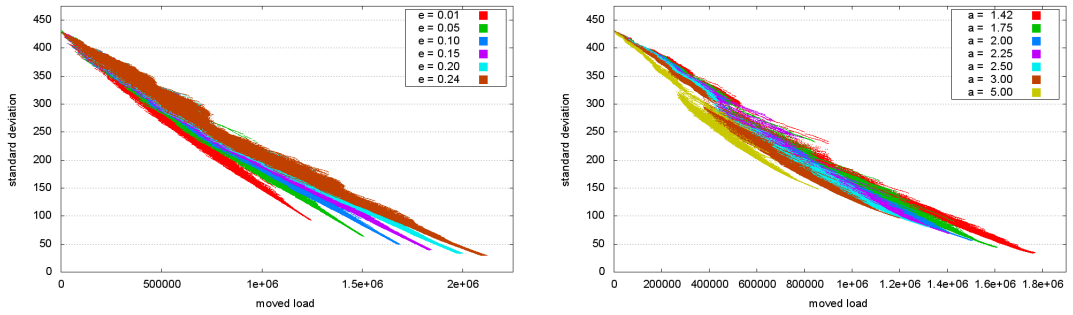
Several of the ideas pictured above can be combined to form new variants of an algorithm. This new variant is then expected to show even better results since two different measures of improving the imbalance and/or reducing the number of moved items are applied together. One of those combinations has already been suggested above: applying `avg1` and `avg2` to `avg3j`. Both resemble the ideal item movement more than a single method alone while `avg3j_avg2` resembles it the most. Less item movements can thus be expected and the simulation will show the influence on the imbalance reached at the end. Additionally to this combination, `stddev2` can be applied. It is expected to improve every aforementioned variant by setting an appropriate s and thus only executing the most useful balance operations.

The following combinations are possible and will be evaluated:

`avg1_stddev2`	`avg3j_avg1`	`avg3j_avg1_stddev2`	`avg3j_stddev2`
`avg2_stddev2`	`avg3j_avg2`	`avg3j_avg2_stddev2`	

3.3.4. Self-tuning algorithms

Most algorithms can be fine-tuned by adjusting certain parameters, i.e. ϵ in `karger` and α in `mercury`. Different values for those parameters will result in different performances in each scenario as can be seen in Figure 3.2 showing plots of moved items vs. imbalance data points of the `karger` and `mercury` algorithms with different parameters. This allows the analysis of the progression of the algorithms during the whole simulation and shows which imbalance can be reached by moving a certain amount of items. They have been taken from Section 4.4 and will be analysed in more detail there.



(a) `karger` (b) `mercury`

Figure 3.2.: *Balance results for* ***karger*** *and* ***mercury*** *with different parameters, scenario: Wikipedia page titles (en), error rate:* 25%.

It can be seen that algorithms with parameters tolerating a bigger skew in the load distribution, i.e. larger epsilon and smaller alpha respectively, start off better than the others. Every imbalance they reach, they reach by moving less items. This might be exploited for self-tuning algorithms in a way that the algorithm starts off tolerating a bigger skew and sets its parameter(s) for better imbalance results during its execution and according to the current distribution of load among the nodes. Estimates of global information will be used to gain knowledge about this distribution. This change will hopefully result in the algorithm starting off with the good results of parameters tolerating a bigger skew and continue the way the others do, finally arriving at the best imbalance the original algorithm can achieve but by moving less items.

This idea has been applied to the `karger` and `mercury` algorithms and results in the following calculations that set the algorithm's parameters for each node during their execution[1]:

```
1  epsilon = bound(0.01, l_avg / max(l_avg + σ_l, l_max - σ_l), 0.24);
2  alpha   = bound(1.42, (l_avg + σ_l) / l_avg, 10.00);
```

[1] **bound** sets the value to the first argument if it is smaller and to the last if it is larger

The idea behind both is that if the standard deviation is high only those nodes should be balanced that are responsible for this high value, i.e. those nodes that have the highest loads in the system. The trick is not to set this bound too restrictively since then too few nodes could be matched with each other. This is why for `karger` a node should only be balanced with an other if their load differs by at least a factor of $f_1 = l_{avg}/(l_{max} - \sigma_l)$, i.e. the average load divided by the maximal load minus the standard deviation. The first idea was to use a factor of $f_2 = l_{avg}/(l_{avg} + \sigma_l)$ but this did not achieve the anticipated results so a more restrictive approach was taken by using f_1 which is usually smaller than f_2. Using the maximum of the two in the calculation above is purely technical and covers the case if $f_1 > f_2$.

The same idea could unfortunately not be applied to `mercury` since it operates differently. At first, balancing is coupled to the average load and occurs only between heavy and light nodes, i.e. nodes with their local load being greater than α or lower than $1/\alpha$ respectively. Thus setting *alpha* influences both bounds instead of a (more flexible) factor as in `karger`. It therefore needs to be set with more care. Additionally the meaning is different which is why it has been set to $(l_{avg} + \sigma_l)/l_{avg}$.

In contrast to the other variants above, a self-tuning variant needs to be adjusted to the way its algorithm operates and the parameters it uses. It can thus not be formalised independently of the algorithm which is why the different implementations will probably show different behaviours. The achievements of one self-tuning algorithm can therefore not necessarily be transferred to another. However it may be evaluated whether the idea behind is good. Self-tuning algorithms will have `_self` added to their names.

4. Evaluation

The following sections will evaluate the performance of the proposed algorithms from the previous chapter. The evaluation has been carried out by implementing a simulator and running these algorithms on different scenarios which will be introduced below. The metrics used for this are explained alongside with a brief overview of the simulator program itself. Finally the collected data will be analysed and further simulations will show how robust the algorithms are in regard to several aspects of the simulations.

4.1. Simulation scenarios

In order to evaluate the proposed algorithms, several scenarios were set up for the algorithms to balance. Real-life applications often store data with keys made of words and numbers, e.g. titles of articles or names of files. Those alphanumeric keys therefore usually follow the distribution of the words of a certain language. Scenarios resembling real-life applications have thus been set up by taking the list of page titles of the English, German and French Wikipedia (page title dumps from 16/08/2009 (en), 10/08/2009 (de) and 19/08/2009 (fr) [4]). Note though that the English Wikipedia (as well as the German and French one) does not only have English page titles but instead describes topics of all languages using English. Nonetheless all three exhibit different distributions as the scenarios' plots in Figure 4.1 show. Additional scenarios include key-distributions following a normal distribution with different parameters as well as an exponential distribution.

Each of the mentioned key distributions was included into a scenario with 5, 10, 20 or 40 thousand nodes which was set up with an initial load of 0.5, 1, 2 or 4 million items. Every node was given an identifier in the circular ID space $I = [0, 2^{64} - 1)$ uniformly at random. In case of alphabetical distributions, page titles were then hashed to an identifier using an order-preserving hash function and from that list of unique keys the requested number of items was drawn uniformly at random. Scenarios following a normal or exponential probability distribution create keys by drawing them randomly according to their distribution and create items accordingly. Those items are finally inserted at the nodes responsible for them. It might happen that a key is drawn multiple times, in which case a neighbouring key is tried or a new key is re-drawn until no conflicts occur. Figure 4.1 shows the resulting load distributions among the nodes in key order

for scenarios with 10 000 nodes and a total load of 1 000 000.

4.2. Metrics

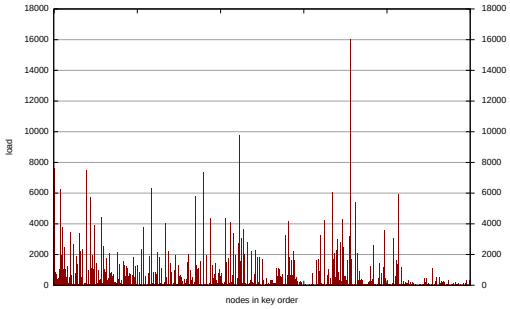
The *performance* of the different algorithms was measured using three different metrics. The first metric is the *standard deviation*, σ_l, of the nodes' loads which measures the degree of imbalance among them, as defined by the system model. The higher its value, the more imbalanced the nodes are in regard to their loads (ref. Section 3.1). An optimal state ($\sigma_l = 0$) is reached when each node stores the system's average number of items.

The second metric is the amount of *moved load*, i.e. moved items. Assuming the cost of transferring a set of items from one arbitrary node to another is proportional to the number of items, the moved load determines the overall transfer cost of the algorithm. Improving an algorithm's performance could therefore either mean reaching the same balance, i.e. standard deviation, by moving less load or reaching a better (lower) standard deviation while moving the same amount of load.

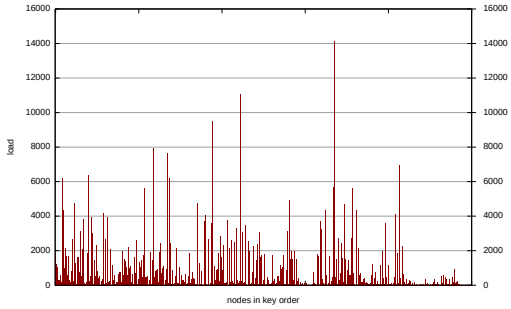
For the sake of comparability the ratio δ_{mal} between the system's maximal and average load is included as well since this, along with the ratio between the system's maximal and minimal load, has been used by several algorithms introduced in Section 2.4. The latter though has been omitted here because it is not well-defined if the minimal load is 0. Compared to the standard deviation as a metric for *imbalance* though, the ratio δ_{mal} has certain disadvantages. For example, a small number of highly loaded nodes could create a very high ratio. Continuing to balance the system and further improving its balance might not decrease this ratio though if at least one of those nodes keeps its load. In a system where this single overloaded node would have a severe impact on the overall availability it makes sense to say that its balance has not really improved. However, this should typically not be the case for DHTs storing items and so this metric is disregarded in further discussions.

4.3. Simulator program

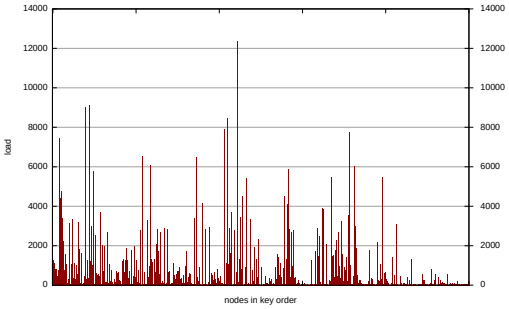
In order to evaluate the algorithms' activities on a DHT with the different scenarios, a simulator program was implemented that emulates such a DHT. It is based on Qt [40] and consists of a common library, a command line client running simulations specified by JavaScript files and a graphical user interface that supports immediate evaluation of a simulation's results with several integrated plots of the metrics described above and their relations. A screenshot of the GUI can be found in Figure 4.2, a sample simulation script in Figure 4.1. Both interfaces offer means of exporting collected simulation results to gnuplot data files and creating appropriate gnuplot scripts that generate such plots.



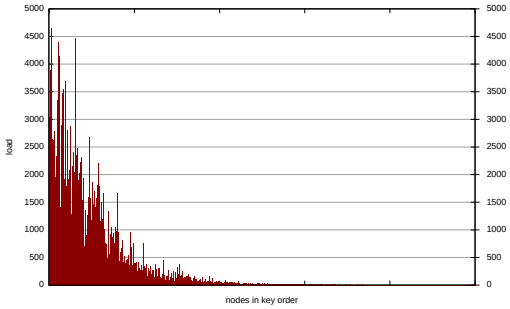
(a) Wikipedia page titles (en)



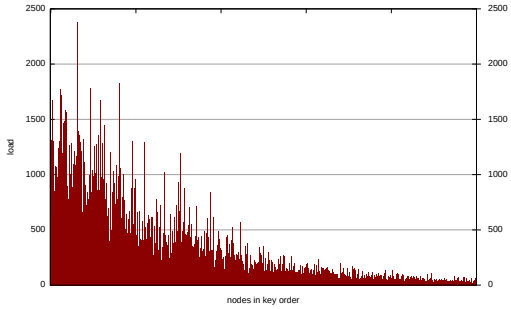
(b) Wikipedia page titles (de)



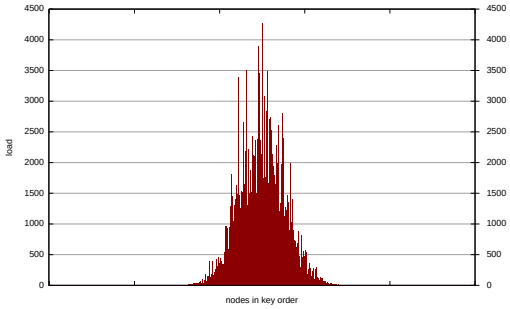
(c) Wikipedia page titles (fr)



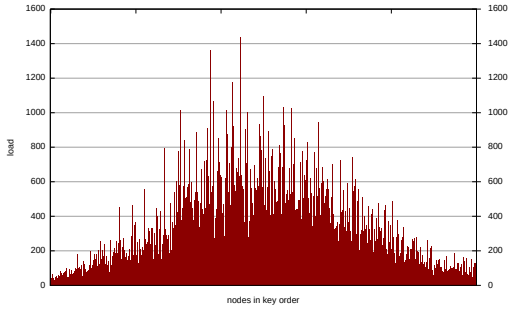
(d) Exponential - $Exp(\lambda = 6 \cdot 10^{-19}))$



(e) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$



(f) Normal - $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$



(g) Normal - $N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$

Figure 4.1.: *Simulation scenarios based on alphabetical, exponential and normal item distributions showing the load (number of items) of each of the* 10 000 *nodes in key order (*1 000 000 *items in total).*

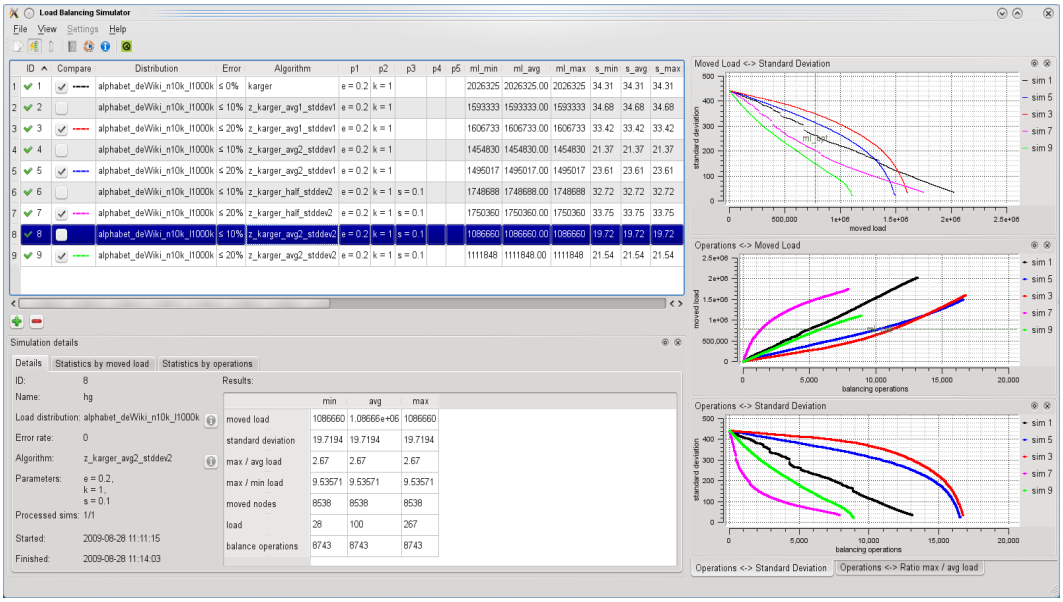


Figure 4.2.: *Snapshot of the Simulator GUI.*

Algorithms and simulation scenarios ("Load Distributions") are implemented as plug-ins and thus new variations can be easily deployed. Load distributions have one parameter: the *error rate* that influences the exactness of the estimates of global information as described above. Algorithms can have several parameters that can all be set in either application interface and are included in the results' data files.

Each simulation has a name and a description and consists of a load distribution and an algorithm. Due to the random nature of most algorithms, a simulation can be specified to run several times (each such test run starts with the same initial parameters). The results of each such simulation will be aggregated to averages over all test runs also storing the minimal and maximal values reached. When an algorithm is invoked by the simulation, it will iterate over all the nodes in the system and will perform its operations for each encountered node. The number of algorithm executions can be specified at the start of a simulation.

Three containers store simulation results: The first container stores the different values of each test run's state at the end of its life-time including the aggregates of the amount of moved load, standard deviation, moved nodes, load among the nodes, number of balance operations and the ratios between the maximal and minimal as well as maximal and average load. Another container stores such values each time the amount of moved load changes and can thus for example be used to plot the moved load against the standard deviation to show which balance state was reached at which costs. Plots showing the standard deviation and the number of moved items for each balance operation, i.e. each operation in which the algorithm signals nodes to perform a jump or slide, can be

```
var ldists = simulation.getLoadDistributions("n10k_l1000k");
var algName = "karger";
var simDesc = "Find best parameters for each scenario.";
var testRuns = 100; // number of test runs of each simulation
var maxTime = 200;// number of algorithm executions in each test run
var collectMI = true; // data by moved items
var collectOp = true; // data by balance operations
var errorRate = 0;
var e_test = new Array(0.01, 0.10, 0.20); // epsilons to test
var k = 1; // number of samples

for (var i = 0; i < ldists.length; ++i) {
 var ldistName = ldists[i];
 var gnuplotPath = "data/" + algName + "--" + ldistName;
 if (simulation.resultsExist(gnuplotPath)) {
  print("skipping " + gnuplotPath);
 } else {
  print("starting " + gnuplotPath);
  simulation.startAutoExportToGnuplot(gnuplotPath, 100);
  for (var i_e = 0; i_e < e_test.length; ++i_e) {
   var algPars = { e: e_test[i_e], k: k };
   var simName = gnuplotPath + "-k" + k + "-e" + algPars.e;
   simulation.addSim(simName, simDesc, ldistName, testRuns,
    errorRate, algName, algPars, collectMI, collectOp);
  }
 simulation.runSims();
 }
}
```

Listing 4.1: *Example simulation script for the Simulator CLI.*

created from the data stored in the third container. It creates such snapshots each time the number of balance operations changes. Since the latter two generate quite much data, they can optionally be turned off.

Much effort has been put into parallelising the core components of the Simulator. As such it uses multiple threads to run the simulations, process the results and export them. At first, each simulation creates two worker threads for the latter two containers. Each test run, which is executed in a separate thread, has a local cache of such snapshots for itself which is piped to a job-queue of the appropriate worker at the end of its life-time. This worker combines the different snapshots one after another. The integration of the snapshot of a simulation's final state into the other container is however done by itself. The number of threads used for concurrent test runs can be limited by providing the command line parameter "`-j <number>`" to the program. By default, the number of

available CPU cores is used. If the simulation scripts enable results to be automatically exported, there will be one more thread which performs those exports similarly to the worker threads mentioned above.

The simulator is available under the GPL version 3 or later [20] and can found on the enclosed DVD as binary packages for Windows and Linux. Source code, license information and documentation is included as well. The latter could be generated at any time by running the Doxygen tool [42] with the supplied `Doxyfile`.

4.4. Simulation results

The following sections will evaluate all simulations that have been run in order to analyse the algorithms themselves and under different aspects of the simulations. As only summarised information may be provided here, all simulations' detailed results are available on the enclosed DVD and can also be re-generated using the provided scripts or the simulator GUI. At first, every introduced variant will be analysed on the `karger` algorithm. This evaluation will go into great detail and try to cover most aspects of the simulation in order to assess the robustness of the algorithms as well. Afterwards the same variations will be applied to the `mercury` algorithm that will show whether the same effects can be observed with another algorithm, too. The basic algorithms themselves will always serve as a reference for the algorithm variations. Also note that this should not be a comparison between `karger` and `mercury` and only the effects of the variants compared to their basic algorithm are evaluated.

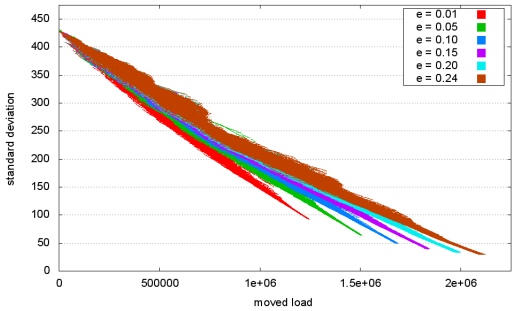
4.4.1. Karger item balancing

Without added global information

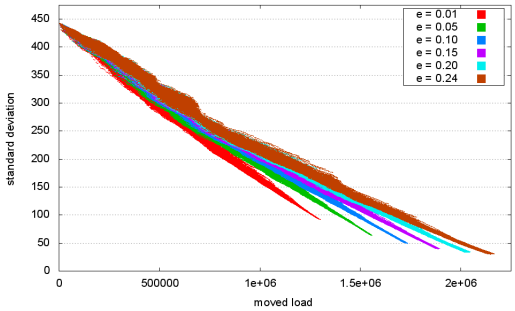
Figure 4.3 shows the standard deviation that the original `karger` algorithm reached after each load movement in each test scenario. It presents the algorithm's performance during its execution and shows which balance can be reached at which cost. Plotted are the collected data points of all simulations with different values of the ϵ parameter, 100 test runs, 200 algorithm executions and one sampled node. Since Karger and Ruhl state that ϵ should be greater than 0 and less than $1/4$, the following values have been chosen: 0.01, 0.05, 0.10, 0.15, 0.20 and 0.24. The number of algorithm executions was originally set to 100 which turned out to be too low for some of the karger variants that in turn exhibited results varying too much from the average of all test runs. Those that still show these variations will be discussed in their respective sections below. Also the effect of setting this number even higher will be evaluated for all algorithm variations.

As can be seen from figures 4.3 (a)-(c), for each single value of ϵ, the alphabetical scenarios show similar behaviour. Evaluating an algorithm's variation will therefore at first concentrate on the scenario created from the English Wikipedia's page titles. Similarly the two exponential distributions will be assessed in favour of the normal distributions which, when compared, show only slightly different results. The variant that performs best on these three scenarios ("Wikipedia (en)", "$Exp(\lambda = 6 \cdot 10^{-19})$" and "$Exp(\lambda = 2 \cdot 10^{-19})$") will finally be simulated on all of them in order to make sure it does not show any different behaviour on the other three scenarios.

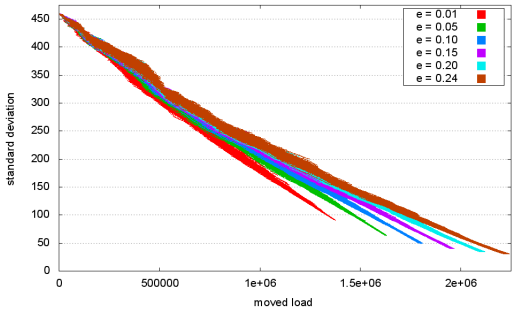
The plots of Figure 4.3 also show that no matter which scenario, smaller ϵ values



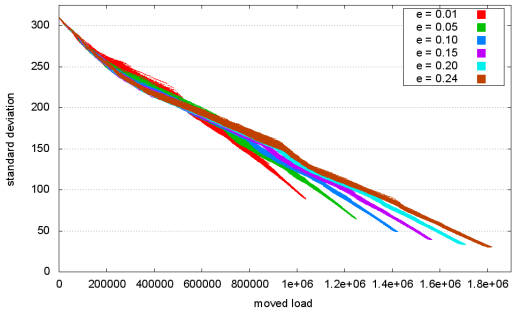
(a) Wikipedia page titles (en)



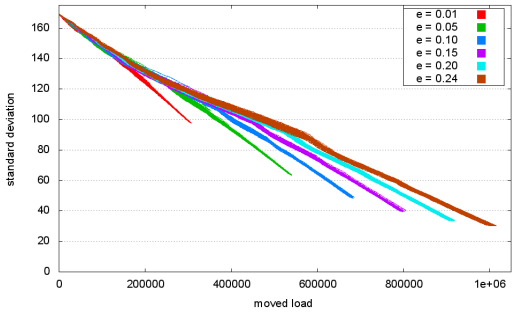
(b) Wikipedia page titles (de)



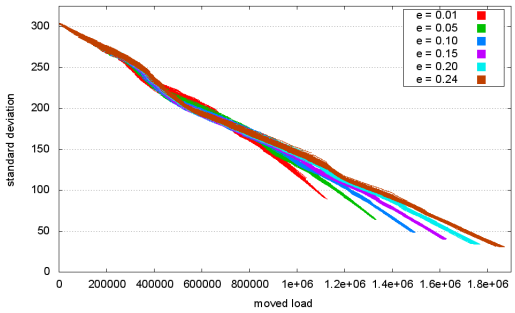
(c) Wikipedia page titles (fr)



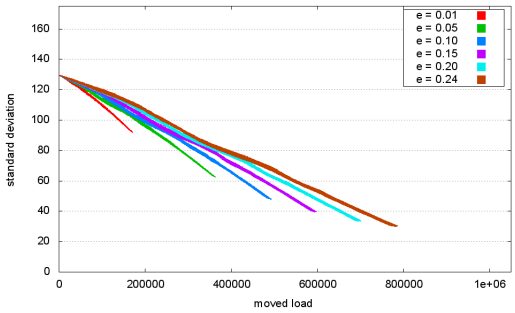
(d) Exponential - $Exp(\lambda = 6 \cdot 10^{-19}))$



(e) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$



(f) Normal - $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$



(g) Normal - $N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$

Figure 4.3.: *Balance results for* `karger` *with different* ϵ *and for each of the scenarios.*

will always reach a given standard deviation by moving fewer load. To understand that effect, recall that `karger` only balances two nodes with each other if their load differs by at least a factor ϵ. Thus small values will make the algorithm only balance nodes with big load differences and since balancing the most loaded nodes affects the imbalance the most, this will lead to a better imbalance with the same number of moved items compared to greater ϵ. However, higher values will reach a better imbalance at the end, which can be seen, too. For the following simulations, a fixed ϵ of 0.24 is used which resulted in the best imbalance at the end.

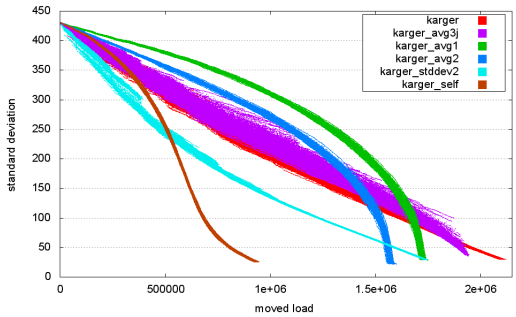
With added global information

In order to assess the improvements of a `karger` variant, it has been simulated with the "Wikipedia (en)", "$Exp(\lambda = 2 \cdot 10^{-19})$" and "$Exp(\lambda = 6 \cdot 10^{-19})$" scenarios using $\epsilon = 0.24$ as mentioned above. For those simulations a 25% inaccurateness of global information was set in order to resemble the estimate of gossiping. This percentage is supported by experiments conducted by Jelasity et al. [28] evaluating the quality of a gossip algorithm that estimates the system size in a dynamic system. It is thus also a reasonable value for the static simulations presented here. Further discussions about the influence of this *error rate* on the simulation results will however be held at the end of this section.

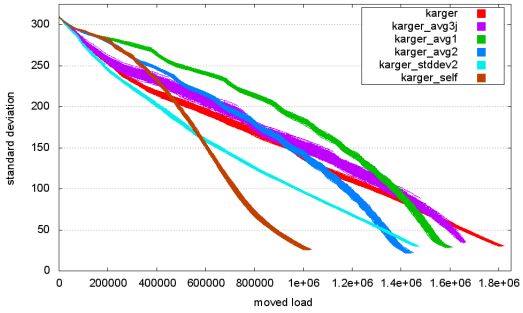
It follows a detailed performance analysis of the different `karger` variants previously introduced. The evaluations will concentrate on showing the imbalance that can be reached by moving a certain amount of load, i.e. items, and the imbalance as well as the number of moved items at the end of each simulation.

The results of the algorithm variants pictured in Figure 4.4 show that **`avg3j`** needs to move more items to reach the same imbalance most of the time during its execution[1]. Only at the end it comes around and falls below the data points of the original `karger` without however achieving the same imbalance. The algorithm is probably refusing some jumps that would have been useful nonetheless, i.e. the potential improvements by the jump would be greater than the prevented light node getting heavy. A possible reason for this might be the error of the estimate being too high but simulations with the exact value of the average load show no better results. In fact, simulations with an error of 25% produce even lower imbalances. Also as Table 4.1 shows, these jumps are traded for slide operations: `karger_avg3j` performs up to 12% fewer jumps than `karger` and increases the number of slides by up to 26%, depending on the scenario. An additional effect that can especially be seen in Figure 4.4a is that the variance of the data points is quite high among different test runs. This also affects the final imbalance at the end

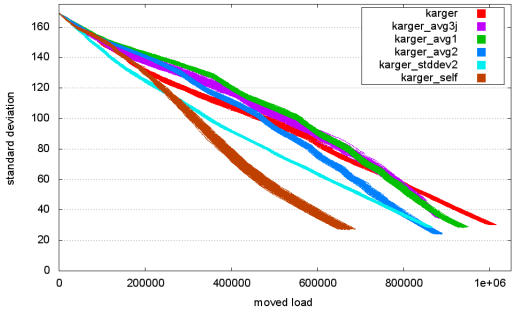
[1] time and number of moved items correlate



(a) Wikipedia page titles (en)



(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)



(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.4.: *Balance results for the* **karger** *variants with one variation, error rate* 25%.

of each test run that varies by up to 35% (ref. Table 4.2 on page 66). Since the final δ_{mal} with `karger_avg3j` is also higher than with `karger`, heavier nodes in the system still exist. It seems that those nodes have not been balanced because they have been matched with nodes for which a jump was not possible. In that case `avg3j` suggests no alternative operation so the heavy node's load is left unchanged and more attempts - and thus time - are needed in order to either find a node that is able to jump or to balance with one of its neighbours. It turns out that by further increasing the number of algorithm executions, the variance of the final imbalance of `karger_avg3j` can be greatly reduced (see the respective section below).

The **`avg1`** and **`avg2`** variants perform better, although a significant time of the simulation they expose worse results than the original algorithm, which is understandable because the item movements they carry out do not have such a great impact on the balance as the original ones. However towards the simulation's end, they significantly increase their performance in both imbalance and number of moved items and outperform `karger`. This is because towards the end fewer and fewer nodes are responsible for the bad imbalance (most of the previously heavy nodes have already been balanced) and thus balance operations on the few heavy nodes quickly improve the standard deviation without many item movements. Additionally these two variants achieve a better overall

Algorithm	**Wikipedia (en)**		$\boldsymbol{Exp(\lambda = 2 \cdot 10^{-19})}$		$\boldsymbol{Exp(\lambda = 6 \cdot 10^{-19})}$	
name	*slides*	*jumps*	*slides*	*jumps*	*slides*	*jumps*
karger ($err = 0\%$)	1101.49	12709.44	598.17	6435.19	444.10	12681.83
..._avg1	1132.43	16517.17	605.74	7600.33	529.43	16079.09
..._avg2	1625.72	15140.70	767.75	7835.22	1178.19	14316.18
..._avg3j	1158.86	12018.87	752.07	5661.78	484.59	12199.47
..._stddev2	432.06	7451.13	334.98	4936.37	124.93	7548.05
..._self	577.39	9384.95	219.64	5756.06	133.95	11304.61
..._avg1_stddev2	403.59	8317.22	312.57	5414.44	222.31	8494.17
..._avg2_stddev2	229.39	8532.15	255.44	5592.58	96.49	8861.53
..._avg3j_avg1	1245.92	13521.21	792.10	6040.37	681.64	13694.82
..._avg3j_avg2	2589.08	10609.35	1226.50	5539.62	2106.98	10522.54
..._avg3j_stddev2	528.08	6958.42	472.35	4192.76	184.06	7064.11
..._avg3j_avg1_stddev2	396.56	6633.99	412.44	4273.73	297.52	6986.91
..._avg3j_avg2_stddev2	396.56	6633.99	412.44	4273.73	297.52	6986.91
..._self_avg2_stddev2	122.36	7635.50	128.22	5043.02	42.91	7617.31

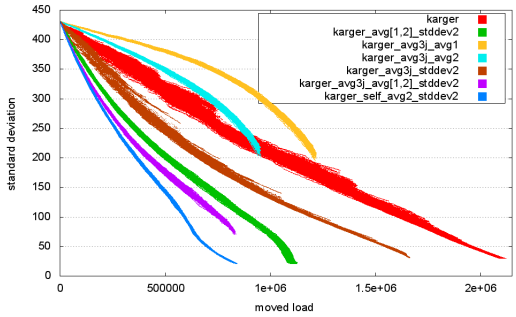
Table 4.1.: *Average number of slide and jump operations of the different* **`karger`** *variants (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 1.5$ *where appropriate).*

balance at the end of the simulation which is owed to the fact that balancing two nodes in the original `karger` could result in both of them being in the bounds of ϵ and thus not being considered for balancing anymore. Here on the other side the receiving node's load would at least be nearer to the average load which results in a better imbalance. In `avg2` this node would be even better balanced than in `avg1` so the standard deviation is expected to be lower at the end with probably fewer item movements. This effect has been confirmed by the simulations as the plots and the results at the end of the simulations in Table 4.2 show.

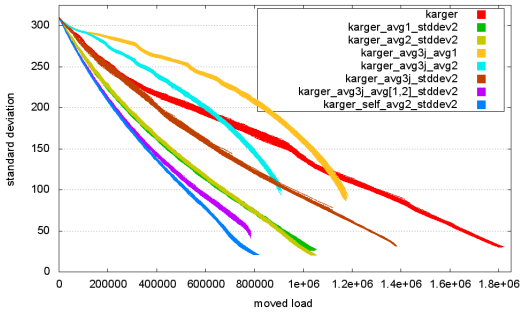
Algorithms using the **`stddev2`** variation need a further parameter, s, that influences the decision on whether to perform a balance operation or not. Initial simulations using `karger_stddev2` have shown that $s = 1.5$ is a good value for all scenarios which will thus be used for every variant of Karger and Ruhl's algorithm that incorporates `stddev2`. Figure 4.4 clearly supports that limiting balance operations to those that are really worth it is a good choice. In all three scenarios the resulting imbalance is comparable to the one of the `avg1` variant and better than the result of the original algorithm but in contrast to these two, throughout the whole simulation any given standard deviation is reached by moving far less items.

The **self-tuning** algorithm starts off similar to the original `karger` in the first and third scenario but then quickly gets to the same imbalance of the other algorithms by moving far less items. In the second scenario it first operates similarly to `avg2` but then outperforms it as well. Surprisingly, in the first two scenarios most of the time `karger_self` reaches the same imbalance by even moving far less items than `karger`

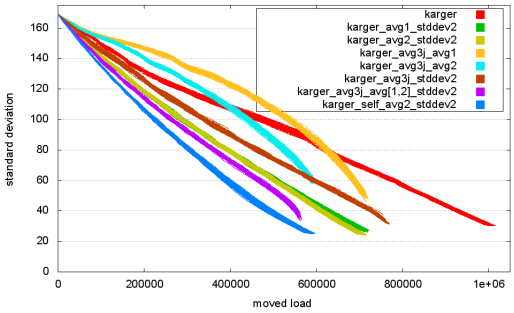
with $\epsilon = 0.01$, e.g. in the alphabetically distributed scenario an imbalance of 100 is reached by moving nearly half the number of items (680 000 versus 1 200 000). The ϵ set by the self-tuning variant is however never below 0.01. The only possible reason for this would be the (slightly) fluctuating value of ϵ due to the error rate. This might allow some balance operations which quickly propagate items from heavily loaded nodes to lightly loaded ones and are otherwise not performed. It can not be an effect due to the randomness of the algorithm since all 100 simulations exhibit similar behaviour which can be seen by the thinness of the algorithm's scatter plot, especially in the first two scenarios. The resulting imbalance at the end of the simulations is also around $10 - 15\%$ better than the original algorithm. Overall, it can be said that the self-tuning algorithm did perform even better than anticipated and is the best among the algorithm variants using a single variation.



(a) Wikipedia page titles (en)



(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)



(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.5.: *Balance results of the combined* `karger` *variants, error rate* 25%. *Indistinguishable curves have been merged, i.e.* `avg[1,2]` *is short hand for "*`avg1` *or* `avg2`*".*

Further simulations, shown in Figure 4.5, have been run in order to analyse the effect of the combination of several of the above evaluated variants incorporated into the `karger` algorithm. The plots show that although the **`avg3j_avg1`** and **`avg3j_avg2`** variants resemble the ideal item movement the most, they don't seem to perform well in any of the three simulations and especially in the alphabetically distributed scenario. This

is probably due to the restricted jump capabilities of the avg3j component as argued above. However, if this variant is combined to **avg3j_stddev2**, it performs similar to stddev2 alone and does not seem to be affected by the limited jumps as much as the original algorithm. Table 4.1 shows an increased number of slide operations of up to 47% and jumps being reduced by less than 15% when adding avg3j to stddev2. The restrictions added by stddev2 however seem to compensate for the restricted jumps and only allow such operations that are useful. Since there is no real improvement to this variant alone though, avg3j_stddev2 could be dropped.

Figure 4.5 also shows that in general variants using the **stddev2** component perform quite well. In particular combinations with **avg1** and **avg2** seem to profit from that. Both exhibit very similar behaviour and can sometimes be indistinguishable in the plots which is why their data points have been merged to a single scatter plot. In those algorithms, restricting balance operations to the ones that significantly reduce the imbalance greatly improves their performance so that any given imbalance is achieved by moving less items than the original algorithm and less items than those variants alone. The omitted operations have thus prevented nodes from taking part in future balance decisions with greater impact due to the restrictions of ϵ (recall that stddev2 only refuses balance operations and does not influence them in any other way). Additionally, the good imbalance reached at the end of the simulations is maintained (ref. Table 4.2). The latter could have been expected since all participating variants alone exhibit quite similar results. The small impact of choosing either avg1 or avg2 though was surprising but seems to be owed to the greater influence of stddev2. Table 4.1 shows another interesting effect: adding stddev2 to any other algorithm reduces its number of slide operations more than the number of jumps in terms of percentages. Thus the majority of the slides in algorithms without stddev2 does not significantly change the overall imbalance and hence most balance improvements can be achieved by moving nodes.

Further combining **stddev2** with **avg3j** and one of **avg1** or **avg2** looks quite promising at the start, regarding the number of items moved in order to achieve a certain imbalance. It is also surprising that avg3j further improves the algorithms that way although previous combinations with it have not shown this behaviour. At the end though they do not reach the imbalance that can be reached by avg2_stddev2 which is probably owed to the restrictions avg3j imposes on jumps. Additionally a large variance of the resulting imbalance can be observed as before with avg3j variants.

Substitutional for variants combined with the self-tuning facilities implemented for karger, the **karger_avg2_stddev2** variant which showed the best results among the previously evaluated algorithms has been equipped with a **self-tuning** ϵ. The plots in Figure 4.5 show the superiority of this variant over all others since it reaches any imbalance by moving far less items. Additionally Table 4.2 shows that the resulting

Scenario	Algorithm	Simulation results (avg)		
name	*name*	*moved load*	*stddev*	δ_{mal}
Wikipedia (en)	karger ($err = 0\%$)	2101882.19 $^{\pm 1.00\%}$	30.17 $^{\pm 2.45\%}$	2.07
	..._avg1	1734141.30 $^{\pm 0.99\%}$	29.07 $^{\pm 2.26\%}$	2.13
	..._avg2	1582001.44 $^{\pm 1.17\%}$	22.40 $^{\pm 2.98\%}$	2.15
	..._avg3j	1934212.76 $^{\pm 0.65\%}$	39.22 $^{\pm 35.27\%}$	10.81
	..._stddev2	1738616.51 $^{\pm 0.64\%}$	29.28 $^{\pm 2.16\%}$	2.12
	..._self	935499.20 $^{\pm 1.33\%}$	25.82 $^{\pm 2.59\%}$	2.20
	..._avg1_stddev2	1113145.43 $^{\pm 1.61\%}$	22.02 $^{\pm 3.58\%}$	2.15
	..._avg2_stddev2	1109117.52 $^{\pm 1.56\%}$	20.92 $^{\pm 1.91\%}$	1.88
	..._avg3j_avg1	1209741.98 $^{\pm 0.86\%}$	206.97 $^{\pm 6.45\%}$	133.82
	..._avg3j_avg2	945844.46 $^{\pm 1.51\%}$	214.48 $^{\pm 5.60\%}$	136.38
	..._avg3j_stddev2	1661461.88 $^{\pm 0.36\%}$	31.27 $^{\pm 3.05\%}$	3.62
	..._avg3j_avg1_stddev2	816453.40 $^{\pm 3.17\%}$	81.73 $^{\pm 15.97\%}$	75.59
	..._avg3j_avg2_stddev2	816453.40 $^{\pm 3.17\%}$	81.73 $^{\pm 15.97\%}$	75.59
	..._self_avg2_stddev2	835771.74 $^{\pm 1.15\%}$	21.43 $^{\pm 2.08\%}$	1.99
$Exp(\lambda = 6 \cdot 10^{-19})$	karger ($err = 0\%$)	1807032.79 $^{\pm 0.95\%}$	30.70 $^{\pm 2.27\%}$	2.02
	..._avg1	1591271.05 $^{\pm 1.17\%}$	28.98 $^{\pm 2.82\%}$	2.10
	..._avg2	1430089.78 $^{\pm 1.53\%}$	21.82 $^{\pm 2.31\%}$	2.09
	..._avg3j	1654011.51 $^{\pm 0.66\%}$	35.96 $^{\pm 18.33\%}$	7.40
	..._stddev2	1462922.39 $^{\pm 0.86\%}$	30.30 $^{\pm 1.64\%}$	2.12
	..._self	1018445.93 $^{\pm 1.58\%}$	25.98 $^{\pm 3.40\%}$	2.09
	..._avg1_stddev2	1036543.22 $^{\pm 1.65\%}$	26.82 $^{\pm 3.29\%}$	2.18
	..._avg2_stddev2	1041755.81 $^{\pm 1.22\%}$	20.72 $^{\pm 1.96\%}$	1.93
	..._avg3j_avg1	1171385.61 $^{\pm 1.06\%}$	93.58 $^{\pm 8.39\%}$	30.55
	..._avg3j_avg2	900437.51 $^{\pm 1.14\%}$	103.84 $^{\pm 10.00\%}$	32.76
	..._avg3j_stddev2	1380104.63 $^{\pm 0.28\%}$	31.78 $^{\pm 2.22\%}$	3.33
	..._avg3j_avg1_stddev2	781607.82 $^{\pm 0.66\%}$	47.33 $^{\pm 13.49\%}$	21.07
	..._avg3j_avg2_stddev2	781607.82 $^{\pm 0.66\%}$	47.33 $^{\pm 13.49\%}$	21.07
	..._self_avg2_stddev2	804922.24 $^{\pm 1.62\%}$	21.06 $^{\pm 2.05\%}$	1.90
$Exp(\lambda = 2 \cdot 10^{-19})$	karger ($err = 0\%$)	1003902.63 $^{\pm 1.14\%}$	30.59 $^{\pm 1.74\%}$	2.03
	..._avg1	937143.51 $^{\pm 1.43\%}$	28.99 $^{\pm 2.36\%}$	2.10
	..._avg2	880638.66 $^{\pm 1.41\%}$	24.45 $^{\pm 1.92\%}$	1.99
	..._avg3j	880305.73 $^{\pm 0.89\%}$	35.55 $^{\pm 11.68\%}$	5.81
	..._stddev2	860193.57 $^{\pm 1.06\%}$	28.76 $^{\pm 1.84\%}$	2.06
	..._self	667771.88 $^{\pm 3.05\%}$	27.53 $^{\pm 2.90\%}$	2.09
	..._avg1_stddev2	713955.21 $^{\pm 1.45\%}$	26.86 $^{\pm 2.44\%}$	2.13
	..._avg2_stddev2	706704.75 $^{\pm 1.55\%}$	24.65 $^{\pm 2.05\%}$	1.96
	..._avg3j_avg1	711236.50 $^{\pm 1.37\%}$	50.87 $^{\pm 9.25\%}$	15.09
	..._avg3j_avg2	590921.06 $^{\pm 0.98\%}$	60.80 $^{\pm 5.26\%}$	16.66
	..._avg3j_stddev2	765143.97 $^{\pm 0.56\%}$	32.17 $^{\pm 3.49\%}$	3.75
	..._avg3j_avg1_stddev2	560023.34 $^{\pm 0.95\%}$	37.13 $^{\pm 10.10\%}$	12.05
	..._avg3j_avg2_stddev2	560023.34 $^{\pm 0.95\%}$	37.13 $^{\pm 10.10\%}$	12.05
	..._self_avg2_stddev2	585624.70 $^{\pm 2.15\%}$	25.40 $^{\pm 2.02\%}$	1.93

Table 4.2.: *Results of the different* `karger` *variants, best variants for each scenario marked in yellow (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$, $s = 1.5$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

imbalance at the simulations' end is only slightly worse than `karger_avg2_stddev2` without self-tuning. It looks like the idea of an ϵ that varies depending on the system's state is also applicable to the introduced variants and can thus achieve even better results by combining the advantages of all of them.

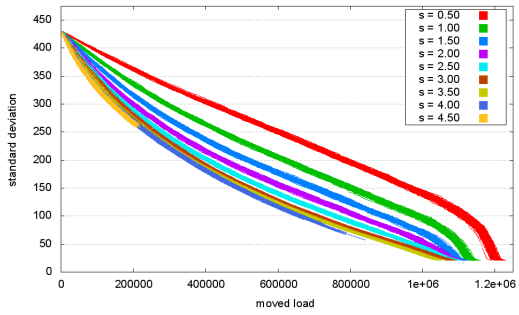
Best `karger` variants

Thus `karger_avg2_stddev2` with and without self-tuning are considered the best variations of the original `karger` algorithm. Both however can still be influenced by their s parameter that defines the minimal imbalance reduction a balance operation should have in order to be considered for execution. They have thus been simulated with different values for s to find the one with the best performance. The plots in Figure 4.6 show the results of these simulations with the three main scenarios. They show that the `karger_avg2_stddev2` variant without self-tuning relies quite much on the correct value being set. Higher s result in a slightly better imbalance at the simulations' end up to a certain bound. Higher values also reach a given imbalance by moving less items which can be clearly seen. If s is increased too much though, the algorithm will not reach a good imbalance at all which can even happen with small increments as the first two plots show (scenarios "Wikipedia (en)" and $Exp(\lambda = 6 \cdot 10^{-19})$). In both scenarios, $s = 3.5$ still shows a good result but $s = 4.0$ already is quite bad and the result of $s = 4.5$ is even unacceptable. The $Exp(\lambda = 2 \cdot 10^{-19})$ scenario however does not show this behaviour with the simulated values although a slightly worse final imbalance can already be seen with $s = 4.5$. The effect will probably start with higher values.

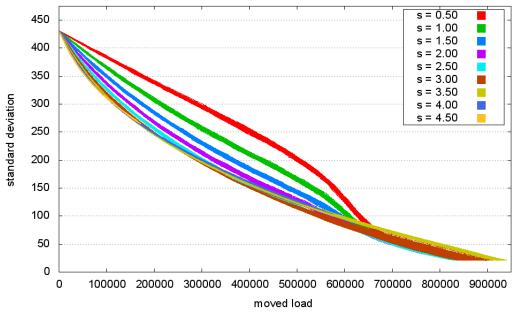
When the `stddev2` variant was developed, its new parameter was integrated in such a way that suggested that its best performance can be reached by a single s for any scenario, i.e. the value of s does not influence (much) the algorithm's performance on any scenario. s thus needs to be set depending on the number of items an algorithm moves in a single balance operation and the resulting change the operation has on the overall balance. If an algorithm only ever moves very few items those moves will probably not have a great impact on the overall imbalance and thus a too high value for s will omit too many balance operations for the algorithm to work. In the case of `karger_avg2_stddev2` however, the number of items the algorithm moves depends on the system's average load, so s needs to be set with care in order not to block too many operations.

The effect of the different performances of the `karger_avg2_stddev2` variant with $s \geq 4.0$ in the three scenarios can however not be explained with a different average load because all scenarios share a common average load of 100. It might instead result from the different imbalances of the scenarios: A single balance operation may not have a big enough affect on the system's imbalance in terms of percentages in scenarios with a greater imbalance. Especially since the number of moved items is limited by the average

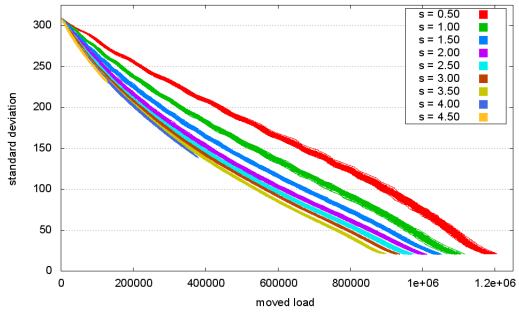
load. This suggests that a weighted bound based on the current imbalance is more useful than statically dropping all operations that do not increase the imbalance by a factor of at least s/n as suggested by `stddev2`. It will probably also be beneficial for this algorithm variant when applied to arbitrary scenarios.



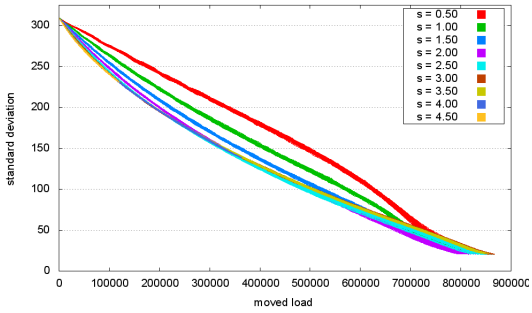
(a) Wikipedia (en), `karger_avg2_stddev2`



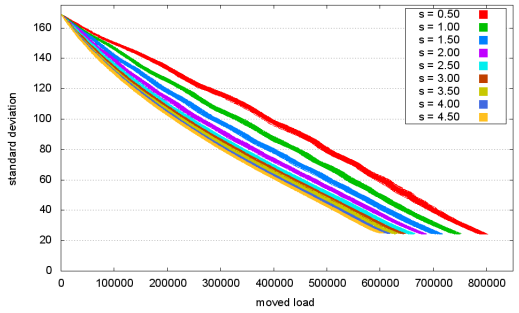
(b) Wikipedia (en), `karger_self_avg2_stddev2`



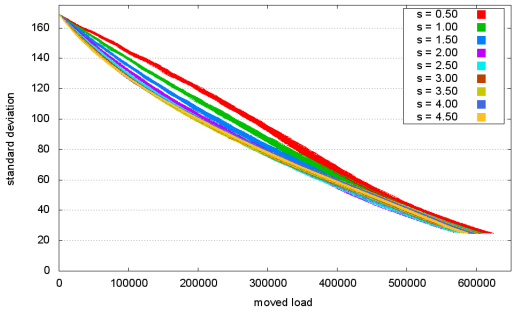
(c) $Exp(\lambda = 6 \cdot 10^{-19})$, `karger_avg2_stddev2`



(d) $Exp(\lambda = 6 \cdot 10^{-19})$, `karger_self_avg2_stddev2`



(e) $Exp(\lambda = 2 \cdot 10^{-19})$, `karger_avg2_stddev2`



(f) $Exp(\lambda = 2 \cdot 10^{-19})$, `karger_self_avg2_stddev2`

Figure 4.6.: *Balance results of the best* **`karger`** *variants with different error rates. (scenario: Wikipedia (en),* $\epsilon = 0.24$ *where appropriate,* 100 *test runs,* 200 *algorithm executions)*

From those simulations alone, the *optimal* s might be chosen as 3.0 but simulations later carried out to evaluate the influence of the error rate revealed that this value was not performing good with low error rates. This is why a safer value of 2.0 was chosen instead (ref. Figure 4.7c on page 73).

Different values of s seem to affect the self-tuning variant `karger_self_avg2_stddev2` in another way than the same algorithm without self-tuning. Results of the self-tuning algorithm shown in figures 4.6 b, d and f show the same behaviour as before with $s \geq 4.0$ in the first two scenarios. Also in the third, it can not be observed. In contrast to the variant without self-tuning though their final imbalances only vary insignificantly and do not necessarily improve with higher s.

Scenario	**Algorithm**	**Simulation results (avg)**		
name	*name*	*moved load*	*stddev*	δ_{mal}
Wikipedia (en)	karger ($err = 0\%$)	2101882.19 $^{\pm 1.00\%}$	30.17 $^{\pm 2.45\%}$	2.07
	karger_avg2_stddev2	1090815.91 $^{\pm 2.45\%}$	20.97 $^{\pm 1.53\%}$	1.87
	karger_self_avg2_stddev2	831745.79 $^{\pm 0.92\%}$	21.38 $^{\pm 1.93\%}$	1.95
Wikipedia (de)	karger ($err = 0\%$)	2150263.47 $^{\pm 0.94\%}$	30.73 $^{\pm 2.72\%}$	2.03
	karger_avg2_stddev2	1121778.42 $^{\pm 1.65\%}$	20.65 $^{\pm 1.99\%}$	1.88
	karger_self_avg2_stddev2	842240.22 $^{\pm 1.01\%}$	20.95 $^{\pm 1.93\%}$	1.91
Wikipedia (fr)	karger ($err = 0\%$)	2223748.31 $^{\pm 0.82\%}$	31.17 $^{\pm 1.85\%}$	2.06
	karger_avg2_stddev2	1166642.63 $^{\pm 1.91\%}$	20.44 $^{\pm 2.13\%}$	1.91
	karger_self_avg2_stddev2	858124.86 $^{\pm 1.36\%}$	20.54 $^{\pm 2.06\%}$	1.92
$Exp(\lambda = 6 \cdot 10^{-19})$	karger ($err = 0\%$)	1807032.79 $^{\pm 0.95\%}$	30.70 $^{\pm 2.27\%}$	2.02
	karger_avg2_stddev2	1000030.18 $^{\pm 1.28\%}$	20.70 $^{\pm 1.62\%}$	1.89
	karger_self_avg2_stddev2	809677.18 $^{\pm 2.46\%}$	20.87 $^{\pm 1.93\%}$	1.90
$Exp(\lambda = 2 \cdot 10^{-19})$	karger ($err = 0\%$)	1003902.63 $^{\pm 1.14\%}$	30.59 $^{\pm 1.74\%}$	2.03
	karger_avg2_stddev2	679464.20 $^{\pm 1.14\%}$	24.64 $^{\pm 1.77\%}$	1.95
	karger_self_avg2_stddev2	583767.38 $^{\pm 2.76\%}$	25.36 $^{\pm 1.56\%}$	1.93
$N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$	karger ($err = 0\%$)	1858444.98 $^{\pm 1.09\%}$	30.98 $^{\pm 1.79\%}$	2.03
	karger_avg2_stddev2	998929.11 $^{\pm 1.02\%}$	20.03 $^{\pm 1.57\%}$	1.86
	karger_self_avg2_stddev2	846733.68 $^{\pm 3.10\%}$	19.85 $^{\pm 2.14\%}$	1.87
$N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$	karger ($err = 0\%$)	778226.73 $^{\pm 1.25\%}$	30.49 $^{\pm 1.89\%}$	2.04
	karger_avg2_stddev2	553003.90 $^{\pm 1.22\%}$	25.86 $^{\pm 2.09\%}$	1.94
	karger_self_avg2_stddev2	501005.91 $^{\pm 3.41\%}$	26.39 $^{\pm 2.02\%}$	1.93

Table 4.3.: *Results of the best* `karger` *variants for all scenarios (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 2.0$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

Starting from a certain value of s, the number of moved items needed to get the final imbalance is increasing instead of decreasing monotonously as in the variant without self-tuning. This *barrier* at which this change is starting depends on the scenario but all three scenarios show a decreasing number of moved items up to $s = 2.0$ which is thus considered *optimal* for the self-tuning algorithm, too. The insignificant effect of different s is probably due to the algorithm's ϵ being coupled to (an estimate of) the system's standard deviation and thus probably a higher value for that is being used earlier than with lower s. This dampens the success of higher values and thus results in

the performances shown.

Since the *optimal* values for parameters of the best karger variants have been set, they can now be simulated on all scenarios previously introduced. Table 4.3 shows the final results of such simulations using `karger` and the `karger_avg2_stddev2` variant with and without a self-tuning ϵ parameter. Also given is the maximum deviation of this average value to the minimal and maximal value among all 100 test runs. It shows that in the alphabetical scenarios, the algorithm variants both achieve an imbalance that is around 30% better than the final imbalance using the original algorithm. Additionally, both move far less items: `karger_avg2_stddev2` without self-tuning moves only about 50% of the amount `karger` moves which can be further reduced to 40% if self-tuning is used. Results of the $Exp(\lambda = 6 \cdot 10^{-19})$ and $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$ scenarios show similar results (45% and 55% less item movements respectively with the same improvements of the imbalance). In the other two scenarios, imbalance improvements are only at about 15% with $30 - 35\%$ and $35 - 40\%$ fewer item movements respectively. Variances from those average results are negligible which indicates that the given results can be expected for any simulation despite the algorithms having a random component. The results of the latter two scenarios not being as good as the others can probably be explained by the scenarios already having a better imbalance at the start of the simulations and ϵ limiting any further improvements.

Number of algorithm executions

As mentioned above, some algorithms, especially those with the `avg3j` variant, have been limited by the number of algorithm executions the simulations were set up with. Simulating them with a too low value will result in a large variance of their results at the simulations' final state. This can be observed in the results presented in Table 4.2 on page 66 with most variations of `avg3j`. In order to further analyse the effect of the number of algorithm executions and see whether this is the limiting factor or inherent in the algorithm, further simulations have been set up using 200, 400 and 800 algorithm executions.

Results of those simulations on the "Wikipedia (en)" scenario are shown in Table 4.4 and indicate that the majority of the algorithms are not affected by the increased number of executions. They mostly achieve insignificantly better imbalances by moving slightly more items. This is good for real-world scenarios where the algorithms are not stopped after an arbitrary number of executions but continue to operate.

All `avg3j` variants but `avg3j_stddev2` however show significant improvements when executed more often. `karger_avg3j` without any more changes for example achieves a 15% better imbalance when executed 800 times instead of 200. Additionally the results of the different test runs then vary only by up to 6.60% instead of 35.27 which is a major

improvement. These results are achieved by moving less than 1% more items. The fact that the variance of the number of moved items with 200 executions is very little supports the assumption that this variant is omitting too many of the potential balance operations and is only waiting for a *perfect match* to occur which is left to chance. In order to achieve comparable results, it would thus have to be called at least 4 times as often as `karger`, but since the improvements compared to that are quite small, another variant is a better choice.

Algorithm	Number of algorithm executions					
	200		**400**		**800**	
name	*moved load*	*stddev*	*moved load*	*stddev*	*moved load*	*stddev*
karger ($err = 0\%$)	2101882.19 ±1.00%	30.17 ±2.45%	2103140.72 ±0.99%	30.03 ±2.45%	2103796.72 ±0.99%	29.96 ±2.44%
..._avg1	1734141.30 ±0.99%	29.07 ±2.26%	1735693.54 ±0.95%	28.87 ±2.21%	1736382.82 ±0.93%	28.79 ±2.25%
..._avg2	1582001.44 ±1.17%	22.40 ±2.98%	1583620.65 ±1.15%	22.11 ±2.76%	1584258.24 ±1.16%	22.00 ±2.92%
..._avg3j	1934212.76 ±0.65%	39.22 ±35.27%	1945354.89 ±0.55%	35.11 ±23.09%	1951379.16 ±0.54%	33.17 ±6.60%
..._stddev2	1738616.51 ±0.64%	29.28 ±2.16%	1739724.63 ±0.64%	29.14 ±2.36%	1740337.26 ±0.65%	29.06 ±2.35%
..._self	935499.20 ±1.33%	25.82 ±2.59%	938410.78 ±1.28%	25.37 ±3.44%	939270.53 ±1.30%	25.24 ±3.14%
..._avg1_stddev2	1113145.43 ±1.61%	22.02 ±3.58%	1114870.02 ±1.60%	21.71 ±3.41%	1115548.02 ±1.61%	21.59 ±3.54%
..._avg2_stddev2	1109117.52 ±1.56%	20.92 ±1.91%	1110666.33 ±1.58%	20.71 ±1.89%	1111222.90 ±1.60%	20.63 ±2.15%
..._avg3j_avg1	1209741.98 ±0.86%	206.97 ±6.45%	1244770.19 ±0.72%	186.00 ±6.54%	1271884.97 ±1.11%	166.41 ±9.26%
..._avg3j_avg2	945844.46 ±1.51%	214.48 ±5.60%	980313.28 ±1.33%	193.18 ±5.60%	1006870.52 ±1.06%	173.92 ±8.54%
..._avg3j_stddev2	1661461.88 ±0.36%	31.27 ±3.05%	1664858.96 ±0.32%	30.55 ±2.95%	1666627.45 ±0.26%	30.21 ±3.13%
..._avg3j_avg1_stddev2	816453.40 ±3.17%	81.73 ±15.97%	843764.96 ±1.84%	68.15 ±18.39%	856083.76 ±1.17%	57.42 ±19.66%
..._avg3j_avg2_stddev2	816453.40 ±3.17%	81.73 ±15.97%	843764.96 ±1.84%	68.15 ±18.39%	856083.76 ±1.17%	57.42 ±19.66%
..._self_avg2_stddev2	835771.74 ±1.15%	21.43 ±2.08%	839216.44 ±0.94%	20.93 ±1.54%	839990.51 ±0.90%	20.83 ±1.61%

Table 4.4.: *Average results of the* `karger` *variants with different number of algorithm executions, most affected algorithms marked in yellow (Wikipedia (en), error rate* 25% *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 1.5$ *where appropriate,* 100 *test runs).*

The imbalance results of the `avg3j_avg1` and `avg3j_avg2` variants can be improved by more executions, too. But unlike the pristine `avg3j` variant, the variance of these results is even higher in terms of percentages. This leaves room for improvements and it is unclear whether an even higher number of executions would solve this. At least

the number of moved items with 800 executions increases by only $5 - 7\%$ compared to 200. `avg3j_stddev2` behaves similarly to `stddev2` alone as already observed above. This continues with an increased number of executions.

The `avg3j_avg1_stddev2` and `avg3j_avg1_stddev2` variants - that looked quite promising at the start of the simulations (ref. Figure 4.5 on page 64) - continue to improve their final imbalance with an increased number of executions. Simulating it four times as often as originally though only achieves an imbalance of 57.42 at the end. Although this is achieved by moving only about 5% more items, the results vary even more than with 200 executions in terms of percentages. If the improvements can be continued by increasing the number of executions even more, it might get near the `avg2_stddev2` variant but it would then probably need much more tries to find nodes that can be balanced with each other.

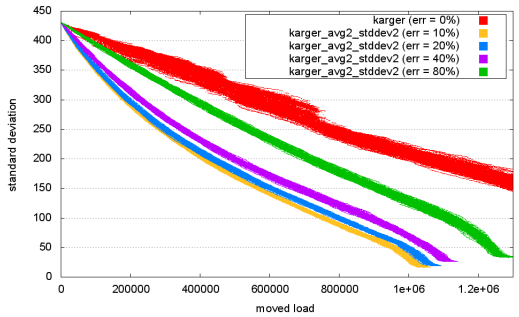
Error rate

One of the major aspects of all algorithm variations introduced above is that they work with *estimated* global information. Gossiping algorithms are used to retrieve any of such values, e.g. average load, standard deviation, and will approximate them to a certain degree which will be more exact the more often the gossip algorithm is executed. In highly dynamic systems however they need some time to incorporate any changes and thus provide worse approximations for a while. The quality of this information in the given simulations can be influenced by setting an error rate (previously at 25%). Further simulations with the best `karger` variants identified above should clarify how dependent their performance is on the accurateness of the global information.

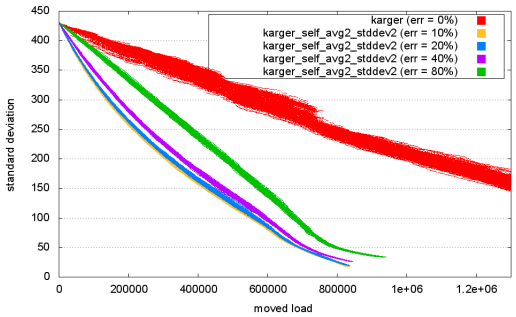
As such, simulations with error rates from $10 - 80\%$ have been run whose results are shown in Figure 4.7. It also includes results of simulations with `karger_avg2_stddev2` using $s = 3$ that has been mentioned above as a candidate for the *best* value of s. As the result table shows, this parameter probably restricts too many balance operations (moving up to $l_{avg} \pm 10\%$ items in a single balance operation is more likely to hit this barrier than $l_{avg} \pm 20\%$).

The plots in Figure 4.7a,b show the results of the `karger_avg2_stddev2` variant without and with self-tuning and also plot the ordinary `karger` algorithm for comparison. Recall that the latter is independent from the error rate since it does not use any global information. Furthermore, the variant without self-tuning seems to be influenced by the error rate a bit more than the other. Its performance however does degrade gracefully so that even with an error of up to 80% an imbalance that is comparable to `karger` can be reached by moving about 25% more items than the same algorithm with a 10% error and still about 40% less than `karger`. According to this the algorithm seems pretty robust to erroneous global information. This is probably due to the fact that the

bound that ϵ imposes has a greater affect than an error rate that only determines how many items are moved.



(a) `karger_avg2_stddev2` ($e = 0.24,\ s = 2$)



(b) `karger_self_avg2_stddev2` ($s = 2$)

Algorithm		**Error**	**Simulation results (avg)**		
name	*parameters*		*moved load*	*stddev*	δ_{mal}
karger	$e = 0.24$	$\leq$ 0%	2101882.19 $\pm$ 1.00%	30.17 $\pm$ 2.45%	2.07
..._avg2_stddev2	$e = 0.24,\ s = 2$	$\leq$ 10%	1041627.75 $\pm$ 2.08%	16.79 $\pm$ 3.02%	2.05
		$\leq$ 20%	1078691.18 $\pm$ 2.28%	19.19 $\pm$ 2.00%	1.93
		$\leq$ 40%	1123017.37 $\pm$ 2.10%	26.36 $\pm$ 1.37%	1.88
		$\leq$ 80%	1291849.74 $\pm$ 1.91%	33.38 $\pm$ 1.46%	1.97
	$e = 0.24,\ s = 3$	$\leq$ 10%	416431.18 $\pm$30.26%	168.79 $\pm$26.22%	9.98
		$\leq$ 20%	1058711.34 $\pm$ 1.39%	19.27 $\pm$ 2.09%	1.97
		$\leq$ 40%	1102975.35 $\pm$ 1.76%	26.57 $\pm$ 1.45%	1.87
		$\leq$ 80%	1268950.19 $\pm$ 2.21%	33.56 $\pm$ 1.67%	1.97
..._self_avg2_stddev2	$s = 2$	$\leq$ 10%	830177.11 $\pm$ 0.80%	17.79 $\pm$ 3.77%	2.11
		$\leq$ 20%	830203.73 $\pm$ 0.80%	19.85 $\pm$ 2.47%	2.01
		$\leq$ 40%	836896.21 $\pm$ 0.92%	26.79 $\pm$ 1.71%	1.86
		$\leq$ 80%	927569.14 $\pm$ 1.37%	34.20 $\pm$ 1.57%	1.97

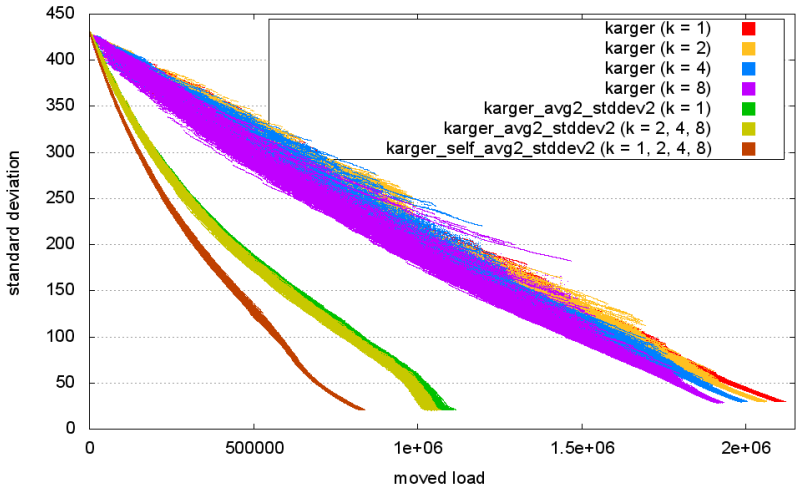
(c) Results at the end of the simulations

Figure 4.7.: *Results of the best* **`karger`** *variants with different error rates. (Wikipedia (en), 100 test runs, 200 algorithm executions each, plots cut off at* $1.3m$ *moved items)*

Even more robust is the `karger_self_avg2_stddev2` variant with self-tuning. Error rates of 10 or 20% nearly differ at all from each other and increasing the error to 40% not even moves 1% more items by still achieving a fair imbalance. A bigger difference can be seen with 80% which offers almost the same imbalance of `karger_avg2_stddev2` at 80% but still moving less than $1m$ items. The better result is probably owed to ϵ being adapted to the system's state every time at each node independently from any previous state. After all with an average load of 100 and several nodes with loads over 1 000 (ref. Figure 4.1 on page 55), an 80% difference does not matter that much. This might be different for simulations with higher loads though.

Number of sampled nodes

Another aspect of the algorithms is their ability to sample multiple random nodes instead of just one. In that case a node can decide with which of the sampled nodes it balances. This should at least allow the algorithm to get to lower imbalances earlier than with only one sample but introduces additional traffic on the network. The algorithms of the following simulations were set up to sample $k \in \{1, 2, 4, 8\}$ different nodes uniformly at random, dry-run the balance operations with each of them and choose the best among them as described in Section 3.2. The results of such simulations using `karger` and the two best variants from above are shown in Figure 4.8.



(a) Results during the simulations (indistinguishable curves have been merged)

Algorithm		**Simulation results (avg)**		
name	*parameters*	*moved load*	*stddev*	δ_{mal}
karger	$e = 0.24, k = 1$	2101882.19 $^{\pm 1.00\%}$	30.17 $^{\pm 2.45\%}$	2.07
	$e = 0.24, k = 2$	2050068.45 $^{\pm 0.68\%}$	29.98 $^{\pm 2.37\%}$	2.01
	$e = 0.24, k = 4$	1988928.43 $^{\pm 0.81\%}$	30.01 $^{\pm 2.12\%}$	1.99
	$e = 0.24, k = 8$	1917664.25 $^{\pm 0.87\%}$	28.83 $^{\pm 2.21\%}$	1.95
..._avg2_stddev2	$e = 0.24, s = 2, k = 1$	1090815.91 $^{\pm 2.45\%}$	20.97 $^{\pm 1.53\%}$	1.87
	$e = 0.24, s = 2, k = 2$	1047354.14 $^{\pm 2.09\%}$	20.75 $^{\pm 1.74\%}$	1.75
	$e = 0.24, s = 2, k = 4$	1040849.12 $^{\pm 2.21\%}$	20.67 $^{\pm 2.29\%}$	1.69
	$e = 0.24, s = 2, k = 8$	1051528.20 $^{\pm 1.91\%}$	20.65 $^{\pm 1.72\%}$	1.67
..._self_avg2_stddev2	$s = 2, k = 1$	831745.79 $^{\pm 0.92\%}$	21.38 $^{\pm 1.93\%}$	1.95
	$s = 2, k = 2$	830804.58 $^{\pm 1.12\%}$	21.03 $^{\pm 1.90\%}$	1.76
	$s = 2, k = 4$	829238.02 $^{\pm 1.18\%}$	20.96 $^{\pm 1.90\%}$	1.69
	$s = 2, k = 8$	827633.51 $^{\pm 0.93\%}$	20.95 $^{\pm 2.18\%}$	1.67

(b) Results at the end of the simulations

Figure 4.8.: *Results of the best* `karger` *variants with different numbers of sampled nodes (k). (Wikipedia (en),* 100 *test runs,* 200 *algorithm executions each, error rate* 25%*)*

The plots clearly show the expected effect with the `karger` algorithm: The more nodes are sampled the more the scatter plot of the algorithm turns bottom left, i.e. the same imbalance is reached by moving less items. Also the final results are slightly better with higher k but as seen with $k = 4$ this is not always the case in contrast to the number of moved items that decreases monotonously.

This effect however can not be observed with the `avg2_stddev2` variants with or without self-tuning. In the latter, a difference can only be found between either sampling one node or more (algorithms with $k \in \{2, 4, 8\}$ perform nearly the same). The insignificant differences of the standard deviation and the number of moved items at the end of the simulations may be owed to the ϵ preventing any further improvements. In contrast to the standard deviation though, the ratio δ_{mal} of the maximum load to the average load always decreases with higher values of k. So higher values do at least show some effect: the maximum load in the system decreases (recall that the average load is constant throughout the whole simulation).

Finally, the self-tuning algorithm does not show any significant differences between the simulations with a different number of sampled nodes, except for a decreasing δ_{mal}. This is probably due to the strong coupling of ϵ to the current system's state and the fact that this variant without self-tuning was already not influenced much by different numbers of sampled nodes.

Scalability

Up until now all simulations have always been carried out with 10 000 nodes and a total load of 1 000 000 items and parameters have been set according to such scenarios. In this section the algorithms will show whether they also perform as expected in scenarios with more nodes or greater loads.

As such `karger` and its best two variants have been run on the "Wikipedia (en)" scenario with 10 000 nodes and different total loads as shown in Table 4.5. This will, substitutionally for all scenarios, clarify whether the algorithms work with different average loads as well. At first it can be seen that the ordinary `karger` algorithm nearly doubles both the number of moved items and the standard deviation at the end of the scenarios when the total load is doubled. This effect is inherent in the increased total load and can also be observed with the two karger variants. The variance among the two values for all 100 test runs also stays within reasonable bounds and does not increase with the increased load. These results indicate good scalability in terms of system load for all three algorithms.

Furthermore scenarios with a different system size, i.e. number of nodes, can be set up to evaluate whether the algorithm's performance depends on the number of nodes in the system. The `stddev2` variant for example has already been developed with this in mind:

Algorithm	Total load							
	500 000		1 000 000		2 000 000		4 000 000	
	moved l.	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*
karger	1048218 ±0.82%	15.14 ±2.15%	2101882 ±1.00%	30.17 ±2.45%	4205981 ±1.01%	60.49 ±2.84%	8414797 ±0.83%	121.09 ±2.58%
..._av2_st2	545453 ±1.89%	10.46 ±1.97%	1090815 ±2.45%	20.97 ±1.53%	2183785 ±1.83%	41.93 ±1.58%	4362201 ±1.86%	83.90 ±1.52%
..._se_av2_st2	414751 ±0.75%	10.66 ±2.37%	831745 ±0.92%	21.38 ±1.93%	1666331 ±0.83%	42.73 ±2.75%	3336612 ±1.14%	85.71 ±1.87%

Table 4.5.: *Average results of the* **karger** *variants with different total loads (Wikipedia (en), error rate* 0% *with* **karger**, *otherwise* 25%, $\epsilon = 0.24$, $s = 2.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each,* 10 000 *nodes, moved load rounded down to the nearest integral, abbreviated algorithm names).*

it considers the fact that if more nodes share the same amount of load, a single balance operation will probably affect the overall imbalance less than with fewer nodes. This is why the system size has been integrated there as well. The same three algorithms as above also had to complete the "Wikipedia (en)" scenario with a fixed amount of 1 000 000 items but different system sizes.

The results of these simulations, presented in Table 4.6, show that `karger` scales linearly with an increasing number of nodes. Inversely to the simulations above, the standard deviation decreases if more nodes share the same total load. It halves with `karger` compared to scenarios with half as many nodes while the number of moved items constantly increases by about 10%. The latter should ideally not change much but such a small increase when doubling the system size is acceptable.

Algorithm	Number of nodes							
	5 000		10 000		20 000		40 000	
	moved l.	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*
karger	1926994 ±1.14%	60.64 ±4.15%	2101882 ±1.00%	30.17 ±2.45%	2306485 ± 0.58%	15.32 ± 1.41%	2506472 ±0.36%	7.57 ± 0.83%
..._av2_st2	1063600 ±2.48%	42.53 ±2.66%	1090815 ±2.45%	20.97 ±1.53%	1122572 ± 1.57%	10.58 ± 1.58%	902782 ±7.76%	30.24 ±21.17%
..._se_av2_st2	823234 ±1.79%	43.38 ±3.57%	831745 ±0.92%	21.38 ±1.93%	756367 ±12.36%	21.73 ±69.05%	613656 ±6.75%	35.88 ±18.32%

Table 4.6.: *Average results of the* **karger** *variants with different system sizes (Wikipedia (en), error rate* 0% *with* **karger**, *otherwise* 25%, $\epsilon = 0.24$, $s = 2.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each, total load* 1 000 000, *moved load rounded down to the nearest integral, abbreviated algorithm names).*

The `avg2_stddev2` variant without self-tuning however only halves the reached standard deviation up to a system size of 20 000 nodes. With 40 000 nodes it gets much

worse and also shows a great variance among the 100 test runs. This indicates that the number of algorithm executions is not high enough for the given scenario which might especially hit `stddev2` variants since they block balance operations and depend on a possibility to find alternatives. The same restriction applies to the self-tuning algorithm which already performs worse than `karger` with 20 000 nodes and continues to do so with more nodes. Further simulations with an increased number of algorithm executions to 400 support the previous assumption of this being the limiting factor. In those, `karger_self_avg2_stddev2` achieves an imbalance of about 5.22 with 40 000 nodes by moving about 882453 items. These values are the ones that could have been expected by this algorithm and since in real-world scenarios the algorithm would operate indefinitely and would not stop after an arbitrary number of executions, this variant is still a good choice that does scale linearly with the system size as well.

Summary of Results

Most of the introduced variants that incorporate estimated global information into the `karger` algorithm use it to their advantage. They provide much better final results in terms of both number of moved items and the imbalance of the system. Especially good performances are achieved by the `avg2`, `stddev2` and self-tuning variants which combined with each other provide even better results. An improvement of up to 60% less item movements with a 30% better standard deviation can be achieved using an optimal value for `stddev2`'s s parameter. These effects can be observed with the scenarios that have a greater imbalance at the simulations' start. Scenarios which already start with smaller imbalances only show improvements of up to 40% less item movements and a 15% better imbalance at the simulations' end.

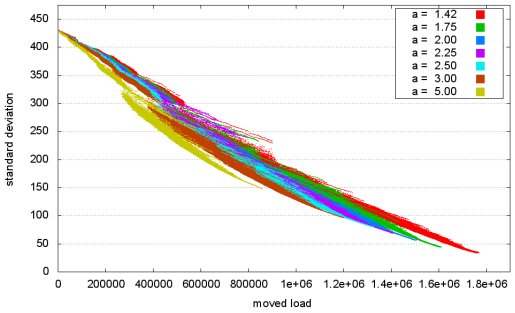
Simulations have also shown that the best two algorithms, i.e. `karger_avg2_stddev2` and `karger_self_avg2_stddev2`, are pretty robust against fluctuations in the quality of the estimated global information they use. They can however not provide the same improvements on the final imbalance with too erroneous data but at least still show major improvements in the number of moved items. With an error of 80% and in an alphabetical scenario (high starting imbalance) still about 40% and 55% less items are moved. Compared to the ordinary `karger` algorithm though, the final imbalance is around 10% worse.

Better results by sampling multiple random nodes can only be observed with the original `karger` algorithm. Sampling more nodes in `karger_avg2_stddev2` produces measurable but insignificant improvements over one sampled node. Differences of sampling either 2, 4 or 8 nodes are negligible though. If self-tuning is also applied to this algorithm variant changes in the number of sampled nodes can nearly be observed and can be buried in the variance of the results among the different test runs.

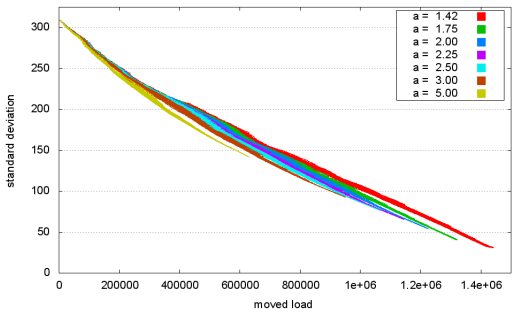
Further simulations of an alphabetical scenario with different total loads or different system sizes have also shown that these two variants scale linearly with those changes. They also offer the same improvements over the original algorithm in terms of both, moved items and imbalance at the end of the simulations. The only thing that can be noticed though is that if the system size, i.e. number of nodes, increases more algorithm executions are needed than with the ordinary `karger` algorithm. This is due to the fact that these variants omit several balance operations and wait for better node matches. It should be considered if these algorithms are applied to real-world scenarios since in order to achieve the same imbalance in the same time they would need to be called more often, e.g. 4 times as much (`karger` already achieves its results with 100 algorithm executions). Further investigations on the aspect of *time*, that was previously ignored, are needed in order to draw any further conclusions to applications on real systems.

4.4.2. Mercury

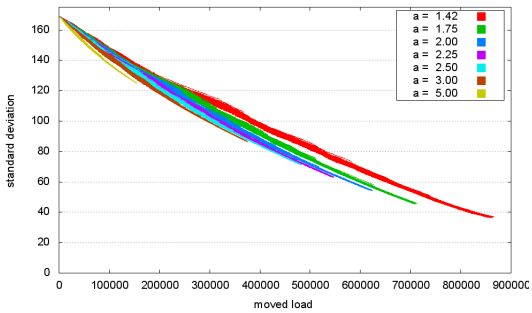
All of the introduced algorithm variants have also been applied to a second algorithm, `mercury`, in order to evaluate whether the effects observed with `karger` also apply to other balance algorithms. The following sections will analyse those variations the same way as the analysis has been done with `karger` above. In contrast to this though it will not be as thorough as before and will concentrate only on proving the effectiveness of the variants and will neither evaluate the influence of the number of algorithm executions nor the effect of multiple sampled nodes. The evaluation will thus also only use the three main scenarios that have been used with the `karger` variants. The other scenarios exhibited very similar results, so the simulated scenarios can be restricted to those three without loss of generality.



(a) Wikipedia page titles (en)



(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)



(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.9.: *Balance results for* `mercury` *with different* α *for the three main scenarios (error rate* $= 25\%$*).*

Without added global information

At first the original `mercury` algorithm has been simulated with different values of the α parameter in order to evaluate its influence. Since $\alpha \geq \sqrt{2}$, values starting from 1.42 have been chosen up to a value that still reaches a fair imbalance. Those values include 1.42, 1.75, 2.00, 2.25, 2.50, 3.00 and 5.00. The simulation results of `mercury` with those

values are presented in Figure 4.9. The plot for each scenario shows the imbalance reached after moving the given number of items and thus indicates the relation between the two invariants *quality*, i.e. imbalance, and *cost*. As with `karger` it can be seen that parameters that tolerate bigger skews, i.e. larger α's, achieve the same standard deviation by moving less items than those tolerating smaller skews. This behaviour is important for the idea of self-tuning algorithms that will be analysed in the following sections. In contrast to `karger` though, results of the 100 different test runs of the simulation on the alphabetical scenario with a given α vary much more which is probably due to the error rate being at 25% and the definition of *local load* in `mercury`.

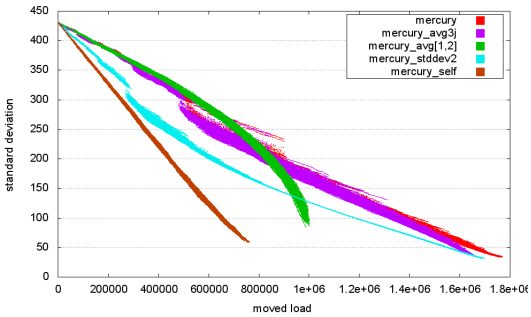
A fixed α of 1.42 is used for the following simulations because it results in the best imbalance at the end. The imbalance reached by this value and the number of moved items at the end of each simulation has been included in Table 4.7 on page 84.

With added global information

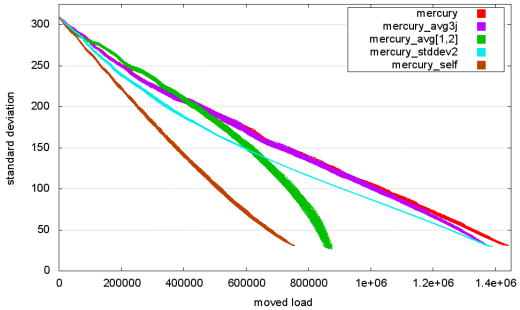
The results of the different algorithm variations that implement one of the variants introduced above are shown in Figure 4.10. In contrast to `karger` though the **`avg3j`** variant alone does perform a bit better than the original algorithm by moving less items and achieving only slightly worse imbalances at the end of the simulations in all three scenarios. This is probably due to *light nodes* (according to the definition in `mercury`) having lightly loaded neighbours with high probability and thus the additional restriction of `avg3j` not being hit that often as in `karger` (recall that only light nodes jump).

The performances of the two **`avg1`** and **`avg2`** variants are so similar when applied to `mercury`, that they could not be distinguished in the plots which is why they have been merged to a single scatter plot. Their results however are very similar to the ones they exhibited with `karger` although they do not show such bad results at the start of a simulation. The imbalance reached by these two variants at the end of the alphabetical simulations however is disappointing (ref. Table 4.7). This can only be explained by too many nodes getting a load that neither makes them *light* nor *heavy* and thus does not allow further balancing. Maybe not enough light nodes exist in order to balance the remaining heavy nodes. This however does not occur in the other two scenarios which is probably because there neighbouring nodes have a similar load with higher probability than in the alphabetical scenario. Averaging the load of three neighbouring nodes to a *local load* may thus lead to the node being neither heavy nor light although it is very heavily or lightly loaded.

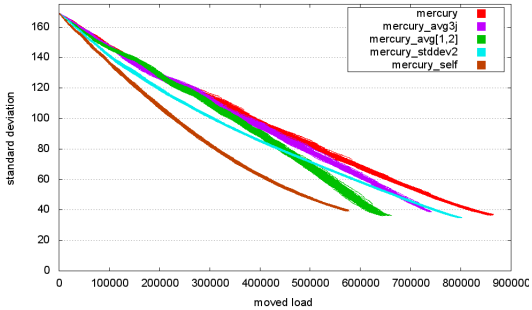
Results of the **`stddev2`** variant are as expected: a slightly better imbalance at the end of the simulations with less moved items than the original algorithm. In order to achieve that though, its s parameter was set to 3.0 which seemed to be the best for `mercury_stddev2` during initial simulations. This will allow only such balance opera-



(a) Wikipedia page titles (en)



(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)



(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

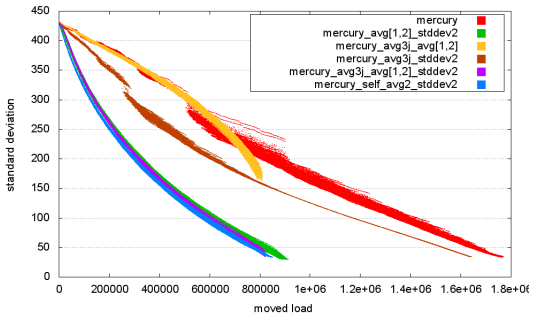
Figure 4.10.: *Balance results for the* **`mercury`** *variants with one variation, error rate* 25%.

tions, that improve the standard deviation by at least a factor of $^{3.0}/_{n}$ which is more selective than the $^{1.5}/_{n}$ used by the `karger` variants. In contrast to those however, the difference in the number of moved items compared to the original algorithm is not that high. Possibly most balance operations performed by `mercury` are already worth it and are thus not prevented by this variation.

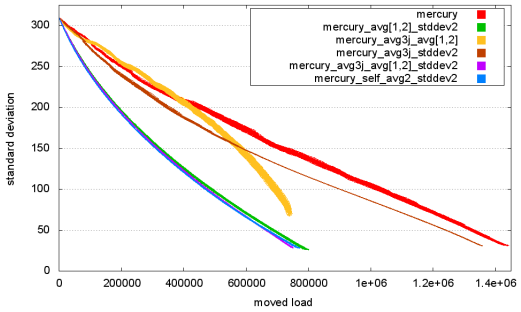
The **self-tuning** algorithm implemented for `mercury` looks quite superior to the other variants in the given plots - similarly to the results of the self-tuning variant of the `karger` algorithm. It moves a lot less items in all three scenarios and in the exponential scenarios it achieves an imbalance that is comparable to the one of the ordinary algorithm. In the alphabetical scenario however its final imbalance is disappointing. The huge variance of this value among the 100 test runs however indicates that the number of algorithm executions is not high enough for this variant which has been confirmed by additional simulations that reach better imbalances with an increased number of executions, e.g. an imbalance around 39 with a lower variance and around 830 000 moved items can be reached by doubling this number.

Results of algorithms with several variants combined are shown in Figure 4.11. Again **`avg1`** and **`avg2`** variants have been too similar to distinguish from each other and have been merged. Apparently the balance operations of mercury and its definition of local

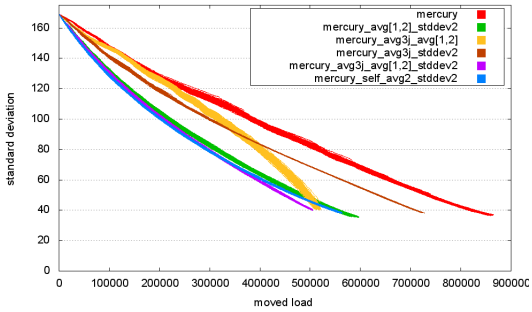
load have a greater influence on the imbalance than the number of items moved during such an operation which does not differ substantially in most cases with those two variants. Combining either of them with **avg3j** though is - as with karger - not a good idea since the resulting algorithm's performance is not as good as the other combinations and does not get close to the imbalance the original algorithm reaches at the end of the simulations. Only the last scenario exhibits good results with that combination which is probably due to its better imbalance at the simulation start. Adding **avg3j** to **stddev2** does not show much difference compared to the results of stddev2 alone which has been observed with the karger algorithm, too.



(a) Wikipedia page titles (en)



(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)



(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

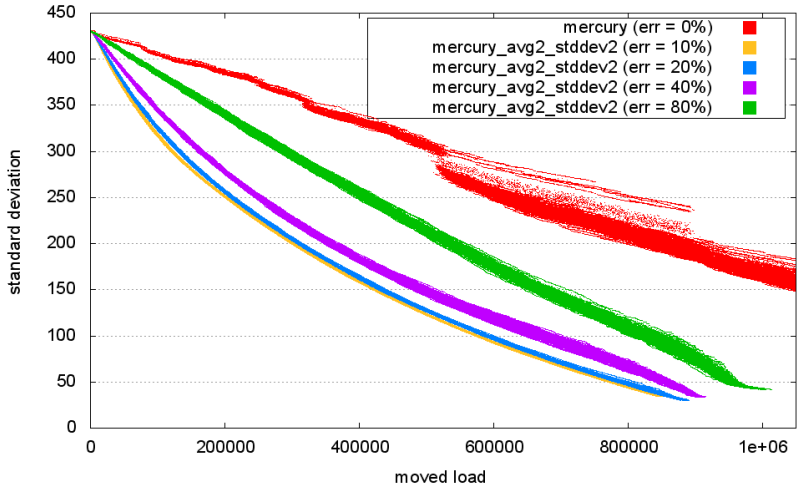
Figure 4.11.: *Balance results for the combined* **mercury** *variants, error rate* 25%. *Indistinguishable curves have been merged.*

The remaining algorithms are running shoulder to shoulder. Best final imbalances are achieved by the **avg1_stddev2** and **avg2_stddev2** variants which can be traded for a little worse imbalance by the advantage of moving slightly less items with the **avg3j_avg1_stddev2** and **avg3j_avg2_stddev2** variants. The **self-tuning** algorithm's performance is somewhere in between these two groups. In contrast to karger it does not outperform the other variants and the good results of self-tuning alone compared to other variants alone could unfortunately not be combined with the good results of another variant as this was the case with karger. Maybe $\alpha \geq \sqrt{2}$ is the limiting factor

here or the *local load* paradigm and its node classification in general. The fact that so many algorithms perform almost identically in contrast to the observations with `karger` would support that statement.

Error rate

Simulations carried out with different error rates as above show that the `avg2_stddev2` variant applied to `mercury` performs similarly than the same variant on `karger`. The only difference that should be noted here is that the performance of the algorithm with an error of 10% is closer to the one with a 20% error than before. Also, the 10% scenario shows a worse imbalance result at the end of the simulations which is probably due to less items being transferred to another node than with 20% and heavy nodes thus not getting "normal" and stealing light nodes that otherwise would have been matched with more unbalanced nodes.



(a) `karger_avg2_stddev2` ($e = 0.24,\ s = 2$)

Algorithm		Error	Simulation results (avg)		
name	*parameters*		*moved load*	*stddev*	δ_{mal}
mercury	$a = 1.42$	$\leq$ 0%	1594504.07 $^{\pm 0.43\%}$	50.13 $^{\pm 0.91\%}$	3.23
..avg2_stddev2	$a = 1.42, s = 3$	$\leq$ 10%	844367.93 $^{\pm 1.01\%}$	35.19 $^{\pm 2.92\%}$	3.18
	$a = 1.42, s = 3$	$\leq$ 20%	883390.55 $^{\pm 1.36\%}$	30.48 $^{\pm 2.93\%}$	3.00
	$a = 1.42, s = 3$	$\leq$ 40%	907052.92 $^{\pm 1.07\%}$	34.01 $^{\pm 2.42\%}$	2.81
	$a = 1.42, s = 3$	$\leq$ 80%	998260.86 $^{\pm 1.60\%}$	42.36 $^{\pm 1.82\%}$	3.05

(b) Results at the end of the simulations

Figure 4.12.: *Results of the best* `mercury` *variant with different error rates. (Wikipedia (en), 100 test runs, 200 algorithm executions each, plot cut off at 1.05m moved items)*

Scenario	Algorithm	Simulation results (avg)		
name	*name*	*moved load*	*stddev*	δ_{mal}
Wikipedia (en)	mercury	1761353.09 ±0.49%	34.43 ± 2.36%	2.87
	..._avg1	991167.62 ±1.21%	105.80 ±17.72%	95.34
	..._avg2	981093.10 ±1.57%	105.99 ±20.78%	95.04
	..._avg3j	1658430.86 ±0.43%	35.45 ± 2.31%	3.13
	..._stddev2	1692875.04 ±0.46%	31.81 ± 1.67%	2.52
	..._self	720993.42 ±5.89%	72.92 ±25.88%	49.45
	..._avg1_stddev2	904086.45 ±1.06%	30.37 ± 7.82%	3.58
	..._avg2_stddev2	887620.31 ±1.02%	30.84 ± 4.72%	3.22
	..._avg3j_avg1	804079.98 ±0.93%	174.60 ± 8.97%	126.44
	..._avg3j_avg2	791579.36 ±0.61%	174.31 ± 6.65%	126.61
	..._avg3j_stddev2	1638856.48 ±0.29%	34.51 ± 1.33%	2.86
	..._avg3j_avg1_stddev2	819585.53 ±0.83%	38.27 ±21.87%	16.85
	..._avg3j_avg2_stddev2	819585.53 ±0.83%	38.27 ±21.87%	16.85
	..._self_avg2_stddev2	841609.76 ±1.69%	34.63 ± 9.47%	3.80
$Exp(\lambda = 6 \cdot 10^{-19})$	mercury	1436953.99 ±0.55%	31.51 ± 2.24%	2.48
	..._avg1	868365.51 ±0.98%	29.12 ±16.86%	9.22
	..._avg2	863661.82 ±1.27%	29.49 ±17.17%	9.02
	..._avg3j	1367239.74 ±0.29%	32.45 ± 1.54%	2.76
	..._stddev2	1387009.40 ±0.31%	29.54 ± 1.55%	2.36
	..._self	751612.57 ±0.70%	31.32 ± 1.87%	2.89
	..._avg1_stddev2	797138.00 ±0.67%	26.47 ± 2.46%	2.55
	..._avg2_stddev2	790562.53 ±0.79%	26.73 ± 2.52%	2.53
	..._avg3j_avg1	742164.69 ±0.74%	74.06 ±10.63%	27.45
	..._avg3j_avg2	736134.58 ±0.73%	73.33 ±11.68%	27.50
	..._avg3j_stddev2	1355825.40 ±0.24%	31.07 ± 1.23%	2.44
	..._avg3j_avg1_stddev2	749505.40 ±0.47%	28.48 ± 2.03%	2.83
	..._avg3j_avg2_stddev2	749505.40 ±0.47%	28.48 ± 2.03%	2.83
	..._self_avg2_stddev2	768556.88 ±0.62%	28.14 ± 1.50%	2.70
$Exp(\lambda = 2 \cdot 10^{-19})$	mercury	858594.53 ±1.03%	36.98 ± 1.50%	2.69
	..._avg1	649470.11 ±2.05%	36.76 ± 1.64%	2.71
	..._avg2	644264.44 ±1.68%	36.98 ± 2.42%	2.72
	..._avg3j	737164.36 ±1.03%	39.15 ± 1.43%	3.02
	..._stddev2	795476.38 ±0.88%	35.11 ± 1.52%	2.58
	..._self	570593.90 ±1.26%	39.80 ± 1.86%	2.95
	..._avg1_stddev2	590802.90 ±1.62%	35.88 ± 2.29%	2.64
	..._avg2_stddev2	583538.73 ±1.18%	36.10 ± 1.50%	2.65
	..._avg3j_avg1	516676.43 ±1.12%	41.81 ± 8.50%	9.72
	..._avg3j_avg2	510947.23 ±1.11%	41.87 ± 7.15%	9.21
	..._avg3j_stddev2	720274.42 ±1.02%	38.80 ± 1.69%	2.76
	..._avg3j_avg1_stddev2	501536.74 ±1.54%	40.71 ± 1.99%	3.13
	..._avg3j_avg2_stddev2	501536.74 ±1.54%	40.71 ± 1.99%	3.13
	..._self_avg2_stddev2	558778.87 ±1.07%	38.53 ± 1.51%	2.84

Table 4.7.: *Results of the different* `mercury` *variants, best variants for each scenario marked in yellow (error rate* = 25% *unless otherwise stated,* $\alpha = 1.42$, $s = 3.0$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

Scalability

As can be seen from the results in Table 4.8, `mercury` does scale linearly with the overall load as does its `avg2_stddev2` variant. It also continues to show the same improvements in terms of percentages compared to the original algorithm.

Algorithm	**Total load**							
	500 000		**1 000 000**		**2 000 000**		**4 000 000**	
	moved l.	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*
mercury	878334 ±0.58%	17.28 ±1.79%	1761353 ±0.49%	34.43 ±2.36%	3527486 ±0.57%	68.83 ±2.30%	7056172 ±0.63%	137.71 ±1.76%
..._av2_st2	443357 ±1.36%	15.56 ±5.05%	887620 ±1.02%	30.84 ±4.72%	1776176 ±1.02%	61.52 ±5.09%	3549476 ±1.34%	123.04 ±4.42%

Table 4.8.: *Average results of the* `mercury` *variants with different total loads (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\alpha = 1.42$*,* $s = 3.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each,* 10 000 *nodes, moved load rounded down to the nearest integral, abbreviated algorithm names).*

Also these algorithm's performances in scenarios with more number of nodes, shown in Table 4.9, show similar results than what has been observed with `karger`. `mercury` moves about 10% more items in scenarios with twice as many nodes just like `karger` did and achieves nearly halved imbalances as well. The `avg2_stddev2` variant however moves only slightly more items comparing the simulations with 5 000 and 10 000 nodes but already starts to show the effect of not being executed often enough with 20 000 nodes which is supported by the variance that is shown by the results. This is similar to the observations with `karger` although there this effect with the non-self-tuning variant did start to occur with 40 000 nodes.

Algorithm	**Number of nodes**							
	5 000		**10 000**		**20 000**		**40 000**	
	moved l.	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*	*moved l.*	*stddev*
mercury	1615172 ±0.69%	69.68 ±2.99%	1761353 ±0.49%	34.43 ±2.36%	1941249 ±0.40%	17.05 ±1.57%	2113715 ±0.47%	8.31 ±1.33%
..._av2_st2	869918 ±1.27%	62.97 ±2.87%	887620.31 ±1.02%	30.84 ±4.72%	873597 ±3.39%	24.80 ±24.67%	724608 ±17.11%	29.05 ±38.04%

Table 4.9.: *Average results of the* `mercury` *variants with different system sizes (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\alpha = 1.42$*,* $s = 3.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each, total load* 1 000 000*, moved load rounded down to the nearest integral, abbreviated algorithm names).*

Summary of Results

Based on the simulations on the three main scenarios carried out and analysed above, the best `mercury` variant is `mercury_avg2_stddev2` without self-tuning. It achieves an up to 15% lower imbalance at the end of the simulations by moving up to 50% fewer items than the original algorithm (depending on the scenario). This confirms the results that have been observed with this variant on the `karger` algorithm above and indicates that the possible improvements are not limited to the presented algorithms. Only the superiority of this variant being equipped with a self-tuning parameter could not be confirmed. However the implementation of this on `mercury` is different to the implementation on `karger` which is why the results can not necessarily be transferred. There are however indications that `mercury` itself is limiting any more improvements since several algorithms that have previously showed different results now almost perform identically.

The robustness of the algorithm variants that has been observed with `karger` though still exist here. `mercury`'s best variant `mercury_avg2_stddev2` shows the same minor influence to changed error rates as `karger_avg2_stddev2` does. The same can be said about the scalability of this variant in regard to increased overall loads and increased number of nodes. It seems that those effects can be transferred to another algorithm if the algorithm itself already shows them (which `mercury` does).

5. Conclusion

5.1. Achievements

At the beginning in Chapter 2 an overview of the field of research was given. It introduced Distributed Hash Tables (DHTs) and some of their representatives, e.g. CAN, Pastry and Chord. This general concept has then been extended to such DHTs that support range queries among their stored data, i.e. they do not only allow the retrieval of the value of a set of single keys but also support queries for ranges of them. Among several implementations of such DHTs, Mercury and Scalaris have been introduced that show only few or no overhead to ordinary DHTs without range queries. Additionally, gossiping techniques have been presented that are able to aggregate global information of a DHT with high confidence and low overhead. Among those estimated values is the system's minimum, average and maximum load, the standard deviation of the load among the nodes as well as the number of nodes in the system.

The problem that arises by the way range-queriable DHTs store their data is the increased variance of *load* among different nodes. They thus apply some sort of load balancing scheme. Chapter 2 also introduced several novel load balancing algorithms that have been developed in recent research. These algorithms have been arranged into 4 different categories since most of them make use of a common set of techniques and only differ in details.

Two of the 18 presented algorithms have then been chosen and equipped with (additional) estimates of global information with the objective of improving their performances. Chapter 3 introduced the system model that is used for the evaluation and presents the two algorithms in more detail. It concludes by introducing five different techniques of using information such as the average and maximum load, the standard deviation of the load among the nodes and system size and describes the ideas behind.

These techniques have then been integrated into the algorithms and evaluated by simulation. Chapter 4 described the simulator that has been set up for this evaluation and defined the metrics that have been used in order to rate the different algorithms. It further presented the simulation scenarios the algorithms should master and finally evaluated their performances. During these evaluations an algorithm variant combining several of the ideas introduced above has been found that significantly increases the

performance of both original algorithms. When applied to any of the given scenarios, the imbalance reached at the end of the simulations is about $15-30\%$ lower and the number of moved items has even been decreased by about $30-50\%$. For this achievement three global estimates are used: the average load, the standard deviation of the load among the nodes and the number of nodes. Further simulations also verified that this variant is quite robust regarding the accurateness of the estimates and also scales linearly to scenarios with higher overall loads or increased number of nodes. With the algorithm described by Karger and Ruhl, the number of moved items can even be further reduced by using a so-called self-tuning variant that sets the algorithm's ϵ parameter according to the system's state. This variation then moved only $40-65\%$ of the amount the original algorithm moves by achieving a $15-30\%$ lower imbalance. Unfortunately this success was not observed with the self-tuning variant developed for the second algorithm.

This evaluation supports the thesis from the beginning that load balancing algorithms can profit from added global information such that they show better performances. As can also be seen, major improvements can be expected by such variations of an algorithm.

5.2. Future Work

As already mentioned in the evaluation above, some of the algorithm's aspects need further investigation. There is at first the concept of *time* which has been omitted in the current system model but is needed for real-world applications. It needs to be evaluated how much more often the new algorithm variants need to be executed in order to show the improved results they exhibited here. It will then need to be evaluated whether the additional operations performed by the omitted balance operations, e.g. getting random nodes, are still negligible in terms of impact on the network.

Another aspect that still needs further attention is a different definition of *load*. The system model used here assumes that the load of a node is proportional to the number of items it stores and so is the transfer cost. This equals a real-world scenario where every item in the system has the same size and the storage on the nodes is to be balanced. More often, another aspect of the stored items is crucial for the system's performance/availability: the popularity of the stored items and the resulting number of item accesses. Other definitions of load may take into account the nodes' (potentially different) capacities of network bandwidth and latency towards other nodes. Systems with heterogeneous nodes in general would also need to be further observed.

Finally different (additional) global estimates may further improve the algorithms and deploying the introduced values in other way might potentially be useful as well. Especially the concept of self-tuning parameters seems promising and may be further extended to different algorithms, too, or be used to create a new algorithm from scratch.

Bibliography

[1] Gnutella protocol specification. http://wiki.limewire.org/index.php?title=GDF. version from: 04/05/2009.

[2] Marble desktop globe. http://edu.kde.org/marble. Version 0.7.1.

[3] Scalaris. http://code.google.com/p/scalaris/.

[4] Wikipedia database dumps. http://download.wikimedia.org/backup-index.html.

[5] Wikipedia: Gnutella. http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=315766949. version from: 23/09/2009, 19:36.

[6] Wikipedia: Space-filling curve. http://en.wikipedia.org/w/index.php?title=File:Hilbert_curve.svg&oldid=313758608. version from: 14/09/2009, 06:03.

[7] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[8] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Manfred Hauswirth, and Seif Haridi. The essence of p2p: A reference architecture for overlay networks. In *Proceedings of 5th IEEE International Conference on Peer-to-Peer Computing*, pages 11–20, Konstanz, Germany, 2005.

[9] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. *Peer-to-Peer Computing, IEEE International Conference on*, 0:33–40, 2002.

[10] James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load balancing and locality in range-queriable data structures. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 115–124, New York, NY, USA, July 2004. ACM.

[11] Kevin Atkinson. Scowl (spell checker oriented word lists) rev 6. http://wordlist.sourceforge.net/. last access: 23/09/2009.

[12] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[13] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, pages 217–225. Springer Berlin / Heidelberg, 2005.

[14] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 80–87. Springer Berlin / Heidelberg, 2003.

[15] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony I. T. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 103–107. Springer Berlin / Heidelberg, 2003.

[16] Michel Charpentier, Gérard Padiou, and Philippe Quéinnec. Cooperative mobile agents to gather global information. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 271–274, July 2005.

[17] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. *SIGCOMM Comput. Commun. Rev.*, 35(4):97–108, 2005.

[18] Chyouhwa Chen and Kun-Cheng Tsai. The server reassignment problem for load balancing in structured p2p systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):234–246, 2008.

[19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001.

[20] Free Software Foundation. GPL. http://www.gnu.org/licenses/gpl.html. last access: 20/08/2009.

[21] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 444–455. VLDB Endowment, 2004.

[22] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24, New York, NY, USA, 2004. ACM.

[23] Ali Ghodsi, Seif Haridi, and Hakim Weatherspoon. Exploiting the synergy between gossiping and structured overlays. *SIGOPS Oper. Syst. Rev.*, 41(5):61–66, 2007.

[24] George Giakkoupis and Vassos Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 302–311, New York, NY, USA, 2005. ACM.

[25] P. Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[26] P. Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 596–606, 2005.

[27] Mikael Högqvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schütt. Using global information for load balancing in dhts. In *Workshop on Decentralized Self Management for Grids, P2P, and User Communities*, volume 0, pages 236–241, Los Alamitos, CA, USA, October 2008. IEEE Computer Society.

[28] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[29] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Mark Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[30] David R. Karger and Matthias Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory of Computing Systems*, 39(6):787–804, November 2006.

[31] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144, New York, NY, USA, 2005. ACM.

[32] Jonathan Ledlie and Margo Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1419–1430, March 2005.

[33] Daniel Mark Lewin. Consistent hashing and random trees : algorithms for caching in distributed networks. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, May 1998.

[34] Gurmeet Singh Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 197–205, New York, NY, USA, July 2004. ACM.

[35] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 131–148. Springer Berlin / Heidelberg, March 2006.

[36] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured P2P systems. In *Peer-to-Peer Systems II*,

volume 2735/2003 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2003.

[37] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[38] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350, 2001.

[39] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Range queries on structured overlay networks. *Computer Communications*, 31(2):280–291, February 2008. Special Issue: Foundation of Peer-to-Peer Computing.

[40] Qt Software. Qt 4.5.2. http://qt.nokia.com/. last access: 20/08/2009.

[41] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[42] Dimitri van Heesch. Doxygen. http://www.stack.nl/~dimitri/doxygen/. last access: 20/09/2009.

[43] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[44] Zhiyong Xu and Laxmi N. Bhuyan. Effective load balancing in p2p systems. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, pages 81–88. IEEE Computer Society, May 2006.

[45] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *SIGCOMM Comput. Commun. Rev.*, 32(1):81, 2002.

A. Implemented algorithms in Pseudo-Code

A.1. Generic helper functions

These are some of the generic helper functions that are used by every algorithm. They make use of a `calcBalancedLoad` method which must be defined for each algorithm.

```
1  // slide between n_i and its successor n_j
2  // return the actual load changes this operation will have
3  slideHelper(DHT d, Node n_i, Node n_j, bool simulate) {
4   // calculate the amount of load that should be moved between the nodes:
5   LoadMove loadMove = calcBalancedLoad(d, n_i, n_j);
6   if (simulate) {
7    return d.simulateSlide(n_i, n_j, loadMove);
8   } else {
9    return d.slide(n_i, n_j, loadMove);
10  }
11 }
12
13 // move n_j to support n_i
14 // return the actual load changes this operation will have
15 jumpHelper(DHT d, Node n_j, Node n_i, bool simulate) {
16  // calculate the amount of load that should be moved from n_i
17  // to the empty n_j after it has been moved:
18  LoadMove loadMove = calcBalancedLoad(d, n_i, emptyNode);
19  if (simulate) {
20   return d.simulateJump(n_j, n_i, loadMove);
21  } else {
22   return d.jump(n_j, n_i, loadMove);
23  }
24 }
```

A.2. Variations of `calcBalancedLoad`

The following method can be used by algorithms which want to even out the load of two nodes they decided to balance with each other. It will try give both nodes half of the sum of their loads. The exact number of transferred items is dependent on their load and will be determined by the `slide` and `simulateSlide` operations.

```
1  calcBalancedLoad_half(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   loadToMove.load = (load(n_fat) - load(n_slim)) / 2;
15   return loadToMove;
16 }
```

Another implementation might want to try not to move more than the average load for which an estimate is retrieved from the DHT. It will otherwise do the same as `calcBalancedLoad_half`.

```
1  calcBalancedLoad_avg1(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   double avg = d.getAvgLoad();
15   loadToMove.load = min(avg, (load(n_fat) - load(n_slim)) / 2);
16   return loadToMove;
17 }
```

The third implementation only balances heavy nodes (those with a load higher than the average) with light nodes (less load than the average) and never make a light node heavy. It will also move no more items than are required to make the heavy node balanced (load equal to the average).

```
1  calcBalancedLoad_avg1(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   double avg = d.getAvgLoad();
15   if (load(n_fat) > avg && load(n_slim) < avg) {
16     loadToMove.load = min(load(n_fat) - avg, avg - load(n_slim));
17   } else {
18     loadToMove.load = 0;
19   }
20   return loadToMove;
21 }
```

A.3. Variations of `getBest`

The first way of finding a best node among a list of candidates uses only local knowledge and decides for the node that improves the standard deviation the most (without knowing its exact value). As pictured in Section 3.2 only the change of the sum of the square of all loads needs to be examined which is done here.

```
1  getBest_stddev1(List<Node> candidates, Map<LoadChanges>
       results) {
2   Node bestNode = none;
3   double minSumLi2_change = 0;
4   foreach(Node n_j ∈ candidates) {
5    double currentChange = 0;
6    foreach(LoadChange lc ∈ results[n_j]) {
7     currentChange += lc.newLoad()^2 - lc.oldLoad()^2;
8    }
9    if (currentChange < minSumLi2_change) {
10    minSumLi2_change = currentChange;
11    bestNode = n_j;
12   }
13  }
14  return bestNode;
15 }
```

A second implementation will get an estimate of the old value of the standard deviation and an estimate of the size from the DHT and use them to calculate the new value. The best candidate is at first the one that improves the standard deviation the most, but additionally to the previous implementation, it is only used if by balancing this node the standard deviation would increase by at least $s/size$. Otherwise nothing is done.

```
1  getBest_stddev2(List<Node> candidates, Map<LoadChanges>
       results, double s) {
2   Node bestCandidate = getBest_stddev1(candidates, results);
3   if (exists(bestCandidate)) {
4    int size = d.getSize(); double oldStddev = d.getStddev();
5    double variance = oldStddev^2;
6    foreach(LoadChange lc ∈ results[n_j]) {
7     variance += lc.newLoad()^2 / size - lc.oldLoad()^2 / size;
8    }
9    double stddev = √variance;
10   if (stddev >= 0 && stddev < oldStddev * (1 - s / size)) {
11    return bestCandidate;
12   }
13  }
14  return none;
15 }
```

A.4. Algorithms based on the item balancing scheme by Karger and Ruhl

Basic algorithm

The following listing shows the basic algorithm as it is shared between most of the variations. Each algorithm implementation has to provide an implementation of the `calcBalancedLoad` and the `getBest` method and can override any of the given methods.

```
1  karger_item(DHT d, double e /*ϵ*/, int k /*samples*/) {
2   foreach (Node n_i ∈ d) {
3    // get k unique random nodes that are not equal to n_i:
4    List<Node> candidates = d.getUniqueRandomNodes(n_i, k);
5    Map<LoadChanges> results;
6    foreach (Node n_j ∈ candidates) {
7     results[n_j] = karger_helper(d, e, n_i, n_j, true);
8    }
9    Node n_j = getBest(candidates, results);
10   if (exists(n_j)) {
11    karger_helper(d, e, n_i, n_j, false);
12   }
13  }
14 }
15
16 karger_helper(DHT d, double e, int k, bool simulate) {
17  if (load(n_i) ≤ e * load(n_j)) { // load(n_j) > load(n_i)
18   return karger_balance(d, n_j, n_i, simulate);
19  } else if (load(n_j) ≤ e * load(n_i)) { // load(n_i) > load(n_j)
20   return karger_balance(d, n_i, n_j, simulate);
21  }
22  return []; // no changes
23 }
24
25 karger_balance(DHT d, Node n_i, Node n_j, bool simulate) {
26  if (n_i == n_{j+1}) {
27   return slideHelper(n_i, n_j, simulate);
28  } else {
29   if (load(n_{j+1}) > load(n_i)) {
30    return slideHelper(n_j, n_{j+1}, simulate);
31   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
32    return jumpHelper(n_j, n_i simulate);
33   }
34  }
35  return []; // no changes
36 }
```

Karger variations

The original `karger` algorithm uses `calcBalancedLoad_half` and `getBest_stddev1` as its implementations for `calcBalancedLoad` and `getBest` respectively. Variations of the original algorithm include the names of the used implementations in their own name, e.g. `karger_avg1_stddev2` uses `calcBalancedLoad_avg1` and `getBest_stddev2` and thus has an additional parameter `s`. It follows the pseudo-code of variants that need to be implemented inside `karger`'s main methods.

Variant `avg3j`

The implementation of `avg3j` only changes one method from the original algorithm:

```
1  karger_balance(DHT d, Node n_i, Node n_j, bool simulate) {
2   if (n_i == n_{j+1}) {
3    return slideHelper(n_i, n_j, simulate);
4   } else {
5    if (load(n_{j+1}) > load(n_i)) {
6     return slideHelper(n_j, n_{j+1}, simulate);
7    } else if (load(n_j) + load(n_{j+1}) ≤ d.getAvgLoad()) {
8     return jumpHelper(n_j, n_i simulate);
9    }
10  }
11  return []; // no changes
12 }
```

Self-tuning

The self-tuning variants of the `karger` algorithm only set a different value of the epsilon parameter for each node at each execution and then continue as the ordinary `karger`. Its main method is thus changed to:

```
karger_item(DHT d, int k /*samples*/) {
 foreach (Node n_i ∈ d) {
  double avgL = d.getAvgLoad(); double maxL = d.getMaxLoad();
  double stddev = d.getStddev();
  double e = bound(0.01, avgL / max(avgL+stddev, maxL-stddev), 0.24);
  // continue as before...
 }
}
```

A.5. Algorithms based on Mercury's load balancing scheme

Basic algorithm

As with Karger, the following listing presents Mercury's basic algorithm shared between most of its variations which need to provide implementations of the `calcBalancedLoad` and the `getBest` methods and can override any other method.

```
1  mercury(DHT d, double a /*α*/, int k /*samples*/) {
2   foreach (Node n_i ∈ d) {
3    if (isLight(n_i)) {
4     if (isHeavy(n_{i+1})) {
5      slideHelper(d, n_i, n_{i+1}, false);
6     } else if (isHeavy(n_{i-1})) {
7      slideHelper(d, n_{i-1}, n_i, false);
8     }
9    } else if (isHeavy(n_i)) {
10    // get k unique random nodes that are not equal to n_i:
11    List<Node> candidates = d.getUniqueRandomNodes(n_i, k);
12    Map<LoadChanges> results;
13    foreach (Node n_j ∈ candidates) {
14     if (isLight(n_j)) {
15      results[n_j] = mercury_helper(d, a, n_i, n_j, true);
16     } else {
17      results[n_j] = []; // no changes (do not balance!)
18     }
19    }
20    Node n_j = getBest(candidates, results);
21    if (exists(n_j)) {
22     mercury_helper(d, a, n_i, n_j, false);
23    }
24   }
25  }
26 }
27
28 mercury_helper(DHT d, double a, int k, bool simulate) {
29  // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
30  Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
31  if (n_j.isNeighbourOf(n'_i)) {
32   return slideHelper(d, n'_i, n_j, simulate);
33  } else if (n'_i ≠ n_j) {
34   return jumpHelper(d, n_j, n'_i, simulate);
35  }
36 }
```

Mercury variations

Similar to `karger`, the original `mercury` algorithm uses `calcBalancedLoad_half` and `getBest_stddev1` as its implementations for `calcBalancedLoad` and `getBest` respectively. Variations of the original algorithm include the names of the used implementations in their own name, e.g. `mercury_avg1_stddev2` uses `calcBalancedLoad_avg1` and `getBest_stddev2` and thus has an additional parameter `s`. It follows the pseudo-code of variants that need to be implemented inside `mercury`'s main methods.

Variant `avg3j`

The implementation of `avg3j` only changes one method from the original algorithm:

```
1 mercury_helper(DHT d, double a, int k, bool simulate) {
2  // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
3  Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
4  if (n_j.isNeighbourOf(n'_i)) {
5   return slideHelper(d, n'_i, n_j, simulate);
6  } else if (n'_i ≠ n_j && load(n_j) + load(n_{j+1}) ≤ d.getAvgLoad()) {
7   return jumpHelper(d, n_j, n'_i, simulate);
8  }
9 }
```

Self-tuning

The self-tuning variants of the `karger` algorithm only set a different value of the alpha parameter for each node at each execution. Its main method is thus changed to:

```
mercury(DHT d, int k /*samples*/) {
 foreach (Node n_i ∈ d) {
  double avgL = d.getAvgLoad();
  double stddev = d.getStddev();
  double alpha = bound(1.42, (avgL + stddev) / avgL, 10.00);
  // continue as before...
 }
}
```

A.10 NAT-resilient gossip peer sampling

NAT-resilient Gossip Peer Sampling

Anne-Marie Kermarrec
INRIA Rennes - Bretagne Atlantique

Alessio Pace
INRIA Grenoble - Rhône-Alpes

Vivien Quéma
CNRS

Valerio Schiavoni
INRIA Grenoble - Rhône-Alpes

Abstract

Gossip peer sampling protocols now represent a solid basis to build and maintain peer to peer (p2p) overlay networks. They typically provide peers with a random sample of the network and maintain connectivity in highly dynamic settings. They rely on the assumption that, at any time, each peer is able to establish a communication with any of the peers of the sample provided by the protocol. Yet, this ignores the fact that there is a significant proportion of peers that now sit behind NAT devices (70% is a fair ratio in the current Internet), preventing direct communication without specific mechanisms. This has been largely ignored so far in the community. Our experiments demonstrate that the presence of NATs, introducing some restrictions on the communication between peers, significantly hurts both the randomness of the provided samples and the connectivity of the p2p overlay network, in particular in the presence of high rate of peers arrivals, departures and failures (aka churn). In this paper we propose a NAT-resilient gossip peer sampling protocol, called $\mathcal{N}ylon$, that accounts for the presence of NATs. $\mathcal{N}ylon$ is fully decentralized and spreads evenly among peers the extra load caused by the presence of NATs. $\mathcal{N}ylon$ ensures that a peer can always establish a communication, and therefore initiates a gossip, with any peer in its sample. This is achieved through a simple, yet efficient mechanism, establishing a path of relays between peers. Our results show that the randomness of the generated samples is preserved, that the connectivity is not impacted even in the presence of high churn and a high ratio of peers sitting behind NAT devices.

1 Introduction

Gossip protocols have received an increasing attention in distributed computing over the past decade as they are robust, simple and highly resilient to churn. Gossip random peer sampling protocols are extensively used in that area to build and maintain unstructured networks.

In gossip peer sampling, each peer typically maintains a set of neighbors (called its view) which it periodically exchanges with another peer in the system, picked from its view. This view is expected to be a sample of peers picked uniformly at random among all peers. Such protocols rely on the implicit assumption that a peer is able to communicate with any peer of its view. Yet, it is a well known fact that today, a large number of peers sit behind NATs [1] (such peers are called *natted* in the sequel, while other peers are called *public*). NAT devices allow several peers with a private IP address to share a single public IP address. NATs implement firewall-like mechanisms that drop unsolicited incoming messages. Consequently, the presence of NATs between peers may prevent them to communicate directly.

While this issue has been addressed in the context of structured p2p networks [1, 7], it has been mostly ignored in the area of gossip protocols so far. To the best of our knowledge, the only work that deals with NATs in gossip protocol is [4]. In this solution, a peer p stores in a cache the peers with which it successfully communicated in the past. The presence of this cache is expected to ensure that at any time p has a high probability to know a peer with which it can communicate. Needless to say, such a simple mechanism cannot ensure that the network will remain connected. As we show in the sequel, the presence of natted peers significantly impacts the properties of the peer sampling protocol with respect to both the randomness of the provided samples and the connectivity. A straightforward cope out is to associate every natted peer to a public one. Provided the natted peer accepts incoming messages from its associated public peer, the latter can act as a relay between this natted peer and any other peer. Obviously, this imposes a significant overhead on public peers which is not acceptable.

In this paper we present $\mathcal{N}ylon$, a fully decentralized NAT-resilient gossip peer sampling protocol where the relay load is evenly spread among peers be they natted or public. This protocol ensures that the communication between a peer and its neighbors is always possible. As soon as a peer picks a neighbor n in its view to initiate a gossip, it

uses as relay the peer which gave it this specific entry to set up a communication with n, and becomes itself a relay to n. Note that the peer might rely on more than one relay to set up a communication with n. Typically, in our experiments, the chain of relays contains on average less that 4 peers in a system comprising 10.000 peers, 90% of which are natted. We show through a simulation study that $\mathcal{N}ylon$ *(i)* ensures that the properties of the peer sampling are preserved in the presence of NATs; *(ii)* evenly balances the relay load between peers; and *(iii)* is highly resilient to churn.

The rest of this paper is organized as follows. We provide a background on NAT in Section 2, we study the impact of the presence of NAT on existing peer sampling protocols in Section 3. Section 4 provides a description of our NAT resilient protocol. We report experimental results in Section 5. We discuss related works before concluding in Section 6.

2 Background on NATs

This section presents the various NAT devices and describes NAT traversal techniques allowing UDP message exchanges between natted peers. More details can be found in [5]. Note that in this section and in the rest of the paper, we do not consider nested NAT topologies.

2.1 NAT devices behavior

A NAT device typically orchestrates the communication between peers sitting behind it and the rest of the network (external peers). When a natted peer opens an outgoing TCP or UDP session through a NAT, the NAT assigns the session a public IP address and port number to allow subsequent messages from an external peer to be received. In addition, the NAT assigns the session a filtering rule, which specifies whether messages received from external peers on the assigned public IP address and port should be forwarded or not to the natted peer's private IP address and port. The public IP address and port mapping, as well as the filtering rule, only remain valid a limited time after the last message was sent (or received) in a session.

Existing NATs differ in the way they assign public IP addresses and ports, as well as in the filtering rules they implement. We briefly describe the four main NAT types.

Full Cone (FC). This is the most permissive type of NAT. The NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. These sessions all share the same filtering rule, which states that the NAT must forward all incoming messages.

Restricted Cone (RC). This type of NAT imposes restrictions on the IP addresses of external peers that can send messages to natted peers. As for FC NATs, the RC NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. All the sessions started from a given natted peer's IP address and port, and involving the same target IP address, share the same filtering rule: the NAT only forwards messages coming from this IP address.

Port Restricted Cone (PRC). This type of NAT imposes restrictions on the IP addresses and ports of external peers that can send messages to natted peers. As for the previous NAT types, the NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. Nevertheless, each session started from a given natted peer's IP address and port towards a target IP address and port, has its own filtering rule. This rule states that the NAT only forwards messages coming from the target IP address and port to which the session has been opened.

Symmetric (SYM). This is the most restrictive type of NAT. For every session started from a given natted peer's IP address and port, the NAT always assigns the same public IP address but a different port. Note that contrarily to other NAT types, the mapping is destination-dependent. The filtering rule is similar to the one used in PRC NATs: the NAT device only forwards messages coming from the target IP address and port to which the session has been opened.

2.2 NAT traversal techniques

The public IP address and port mapping and the filtering rules determine how peers can communicate. As long as a peer behind a FC NAT regularly sends or receives messages through the public address and port the NAT device assigned to it, it will have a valid filtering rule forcing the NAT device to forward it all incoming messages. Rather, if the target peer is behind a RC, PRC, or SYM NAT, the source peer willing to communicate with it has to apply a so-called NAT traversal technique. NAT traversal techniques rely on the use of *rendez-vous* peers (*RVP*) able to exchange messages with both the source *and* the destination peers[1]. There exist two different techniques depending on the combination of source's and target's NAT type. The two techniques are described below. The table summarizes which one should be used in various configurations. Source peer's NAT type is given in the most-left column, whereas target peer's NAT type is given in the heading row.

	public	RC	PRC	SYM
public	direct	hole punching	hole punching	relay
RC	direct	hole punching	hole punching	hole punching
PRC	direct	hole punching	hole punching	relaying
SYM	direct	mod. hole punching	relaying	relaying

[1] *RVP* is usually a public node to which the source and destination peers periodically send PING messages.

Hole punching. In the hole punching technique, the source peer sends a PING message to the destination peer. Consequently, the source peer's NAT device creates a filtering rule forcing it to forward incoming messages from the destination peer. The source peer then sends an OPEN_HOLE message to an *RVP*, indicating that it wants to communicate with the destination peer. The *RVP* forwards the OPEN_HOLE message to the destination peer. As soon as it receives the OPEN_HOLE message, the destination peer sends a PONG message to the source peer. Thereafter, the NAT device of the destination peer has a valid filtering rule allowing incoming messages from the source peer (we say that there is a *hole* in the NAT). The source peer can start sending messages to the destination peer as soon as it receives the PONG message[2]. Note that for most combinations (i.e. those not involving SYM NATs), after the hole punching technique has been applied, the destination peer can also send messages directly to the source peer.

Relaying. In some cases, the hole punching mechanism cannot be used: when the destination peer is behind a SYM NAT and the source peer is either behind a PRC NAT or a SYM NAT, or when the destination peer is behind a PRC NAT and the source peer is behind a SYM NAT. This is due to the fact that the SYM NAT device assigns a different port to every new session, and this port is not known by the source peer. The only possibility for sending messages to the destination peer is then to use the *RVP* as a relay.

3 Impact of NATs on existing protocols

Various peer sampling protocols have been proposed [10, 15, 18]. The protocols described in [10, 15] rely on random walks. These protocols assume a fairly static peer interconnection topology and are not specifically designed to sustain high levels of churn. Conversely, gossip protocols have been designed to handle peers joining and leaving the system at a high rate. We focus on such protocols in the sequel.

A generic gossip peer sampling protocol is described in Figure 1. The system is composed of a set of uniquely identified peers, each one storing references to few other peers into a view. Typically, the view size is in the order of $log(n)$, where n is the number of peers in the network.

The generic protocol works as follows: each peer periodically initiates a communication (i.e. gossips) with one target peer selected from its view. The source and/or the target peer exchange their views. When a peer receives a view, it merges it with its view, and truncates the result to a constant maximum view size. This is typically called a view *shuffling*.

A peer sampling protocol is expected to provide the following properties: (i) the graph formed by peer views remains connected, and (ii) every peer in the network has the same probability to be selected by other peers (the provided sample is *random*).

```
1  every shuffling_period units do
2    target ← select_gossip_destination(view)
3    send ⟨REQUEST, view⟩ to target
4    if push_pull then
5      receive ⟨RESPONSE, view_t⟩ from target
6      view ← merge_and_truncate(view, view_t)
7    increase_view_age()

8  on receive ⟨REQUEST, view_s⟩ from source do
9    if push_pull then
10     send ⟨RESPONSE, view⟩ to source
11   view ← merge_and_truncate(view, view_s)
12   increase_view_age()
```

Figure 1. Generic peer sampling protocol.

The generic gossip-based peer sampling protocol described in Figure 1 can be configured along the following three dimensions [9]: *(i)* Gossip target selection: can either be done randomly (rand), or by picking the oldest peer in the view (tail); *(ii)* View propagation: either only the source peer sends its view to the target peer (push), or both source and target peers exchange their view (push/pull);

(iii) View merging: when truncating a view, randomly chosen peers are kept (rand), or the youngest ones (healer), or the ones received from the other peer (swapper).

We evaluated six different configurations of the generic protocol described in the previous section. The view propagation strategy is the same in all the configurations and is set to push/pull, which is most used in the literature as a push mode consistently exhibits significantly worse performances than push/pull. The gossip target selection and view merging strategies that we evaluated are those described above.

The experiments have been obtained through simulations. The network size is 10,000 peers, and the bootstrapping procedure is such that at the beginning of the simulation all peers' views are filled with randomly chosen public peers. The initial graph is thus always connected. No churn was considered. A more detailed description of the experimental setup is done in Section 5. Moreover, for the sake of simplicity, only PRC NATs are considered in the experiments presented in this section. We evaluated the protocols along the following metrics: *(i)* the resilience of the protocol with respect to network partitioning; *(ii)* the ratio of stale entries in the views and; *(iii)* the randomness of the resulting views.

Network partitions. Figure 2 shows the size of the biggest

[2]When the source peer is behind a SYM NAT, the hole punching technique needs to be slightly modified. Indeed, as the destination peer does not know the public IP address and port that has been assigned to the source peer, it uses the *RVP* to send the PONG message to the source peer.

cluster as a function of the percentage of natted peers for two view sizes (15 and 27). The biggest cluster size is expressed as the percentage of peers that belong to it. We clearly see that the graph partitions when the percentage of natted peers reaches a certain threshold (50% and 70% for the considered view sizes). We observe that, as expected, increasing the view size has a positive impact on the biggest cluster size for all protocols. This result is not surprising as it is well known that a graph remains connected with only a few neighbors. One can legitimately consider that increasing the view sizes is enough to prevent partitions in the presence of NATs. We show in the reminder of this section that increasing the view size is not a satisfactory solution with respect to the two other metrics, the randomness and ratio of stale entries.

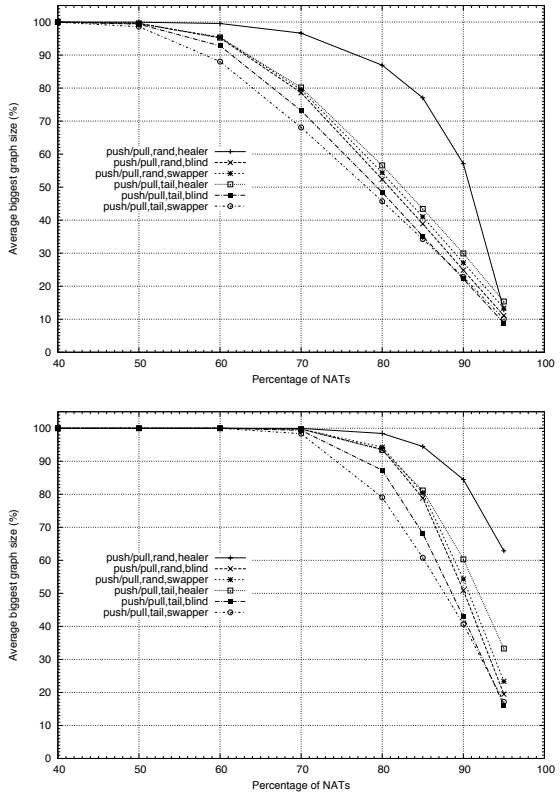


Figure 2. Size of the biggest cluster for view sizes equal to 15 (top), 27 (bottom).

Stale references. Figure 3 shows the average percentage of stale references in peer views for two different view sizes (15 and 27). A reference to a peer is said to be stale when it is not possible to communicate with this peer (due to the presence of NATs). We observe that a small proportion of natted peers suffices to cause peers to maintain stale references in their view. This percentage of stall references almost linearly grows with the percentage of natted peers. Moreover, we observe that the percentage of stale references increases when the view size increases, and that the percentage of stale references decreases for view size 15 when the percentage of NATs reaches a certain threshold (85%). These two observations can be easily explained by two facts. First, increasing the view size decreases the probability that two peers shuffle with each other twice during the lifetime of a NAT filtering rule. Second, with a large percentage of NATs and view size 15, the network starts to significantly partition in many small clusters. Consequently, two peers within a cluster have a very high probability to shuffle with each other twice during the lifetime of a NAT filtering rule.

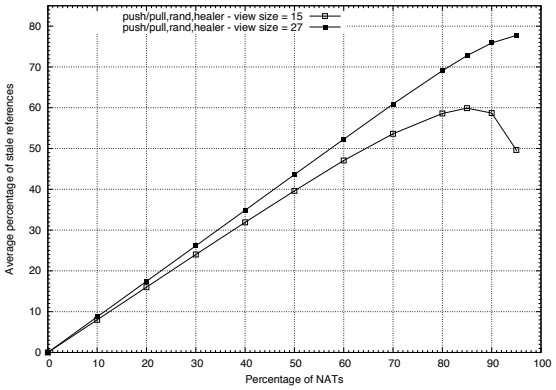


Figure 3. Percentage of stale references.

Randomness. Figure 4 shows the average ratio of non-stale references that correspond to natted peers. Again, we consider two different view sizes (15 and 27). For instance, the plot shows that with 40% of natted peers and a view of size 15, peers have on average only 10% of their non-stale references that correspond to natted peers. This typically means that 40% of the peers are sampled only 10% which is obviously a non uniform random sampling. As in Figure 3, we observe that increasing the view size negatively impacts the protocol. We also observe that when the percentage of NATs reaches a certain threshold (70%), the average ratio of non-stale references increases. The explanation is similar to the one given for Figure 3.

4 The $\mathcal{N}ylon$ protocol

In this section, we present $\mathcal{N}ylon$, a NAT-resilient gossip peer sampling protocol. A commonly used technique for traversing NATs is to use public RVPs [11, 19]. This technique could be used to build a NAT-resilient peer sampling protocol as follows: a source peer needing to communicate with a natted peer, would contact first the natted

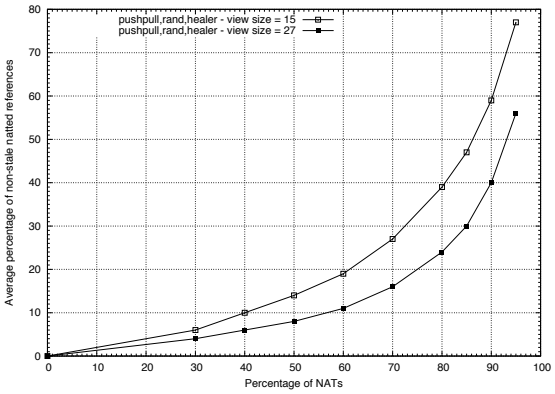


Figure 4. Ratio of non-stale references towards natted peers.

peer's public RVP to forward an OPEN_HOLE message to the target peer. This simple scheme suffers however from several drawbacks. First, the extra load induced by the presence of NATs is supported by the public peers. This creates an uneven distribution of the load where public peers contribute much more to the protocol than natted peers.

Another issue is the non uniform impact of failures of natted and public peers. A public peer's failure invalidates all references to natted peers bound to it. A possible solution is to use several RVPs for each natted peer. Nevertheless, this solution increases the bandwidth consumption.

In order to overcome the limitations imposed by using only public RVPs, we design a fully decentralized protocol that uses both natted and public peers as RVPs. Relying on natted peers for implementing RVPs is challenging: effectively, an RVP must be reachable by all peers willing to communicate with peers for which it acts as RVP. This is obviously impossible to ensure that a natted RVP will have valid filtering rules for every peer in the system. Instead, peers may rely on a routing infrastructure to send messages to any RVP in multiple hops. This is for instance what is applied in Distributed Hash Tables (DHT), where each peer in the DHT maintains valid filtering rules for the natted peers that are in its routing table. When a peer needs to communicate with an RVP, its message is routed using the DHT. Unfortunately, it is not possible to use such a routing infrastructure in the large-scale, highly dynamic, environments that we consider in this paper.

The design of $\mathcal{N}ylon$ relies on two observations:

1. A gossip protocol does not require all peers to be reachable at any time by all peers. Effectively, at a given time, the only peers a given peer might want to communicate with are those that are in its view.

2. In gossip protocols, although a peer *should* be able to communicate with any peer in its view at any time, it does not. Instead, a single peer of its view is picked upon each gossip operation. It may be the case that a peer p in the view of a peer q is removed from q's view without p and q effectively gossip with each other.

$\mathcal{N}ylon$ leverages these two observations to build NAT-resilient gossip-based peer sampling protocols in which all peers can act as RVPs. The first observation is used to implement a hole punching protocol for only a subset of the system. The second observation is used to implement a *reactive* hole punching protocol which consists in performing the actual hole punching protocol between two peers only when needed, namely when a gossip between the two peers is initiated. This avoids to systematically send an OPEN_HOLE message to all peers that p adds in its view.

The $\mathcal{N}ylon$ protocol. The main idea of $\mathcal{N}ylon$ is to implement *reactive* hole punching. Intuitively, this works as follows: a peer only performs hole punching towards peers it gossip with. Hole punching is implemented using a chain of RVPs that forward the OPEN_HOLE message until it reaches the gossip target.

The chain of RVPs is built as follows. Consider the case of a peer $n1$ shuffling with a peer $n2$. After having performed hole punching towards $n2$ (using a chain of RVPs), peer $n1$ and $n2$ can directly communicate with each other. Thus, they both become RVP for each other. Consider now that later, one of them, say $n2$, shuffles with a peer $n3$ and gives it a reference to $n1$. Before shuffling, peers $n2$ performs hole punching towards $n3$. Consequently, as between $n1$ and $n2$, peers $n2$ and $n3$ both become RVP for each other. Finally, consider that $n3$ shuffles with a peer $n4$ and gives it a reference to $n1$. A chain of RVPs has thus been created, as shown in Figure 5. This chain allows $n4$ to shuffle with peer $n1$. For this purpose, it performs hole punching towards peer $n1$ by sending an OPEN_HOLE message to $n3$ that will forward it to $n2$, that will forward it to $n1$.

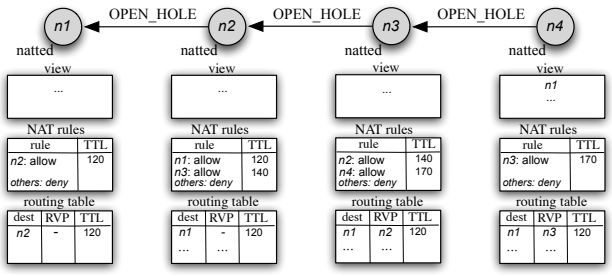


Figure 5. $\mathcal{N}ylon$ operating principle.

As illustrated in Figure 5, in addition to its view, each peer maintains a routing table. This routing table maintains the mapping between a natted peer in its view and its associated RVP.

For each peer n in the routing table, the RVP is the peer it shuffled with to obtain the reference to n. RVPs in $\mathcal{N}ylon$ are constantly changing and following the reactive flavour of $\mathcal{N}ylon$, RVPs do not proactively refresh holes. Therefore, a time to live (TTL) is associated to each RVP entry in the routing tables. TTLs are exchanged by peers together with their views and are updated every shuffling period, and every time a message from one RVP stored in the routing table is received. Note that the TTL mechanism assumes that there is a known upper bound on the latency between each pair of peers[3].

Pseudocode. The pseudocode of the $\mathcal{N}ylon$ protocol is presented in Figure 6. The basis of the protocol is the (push/pull, rand, healer) protocol presented in Section 3. The only additions to the protocol are for handling NAT traversal techniques and implementing the RVP chaining mechanism presented in the previous paragraph. The routing table code is not presented in the figure. It is abstracted in four methods. The `next_RVP()` method returns the next RVP to be used for a given destination. Note that if the destination is directly reachable (because either the destination is public or the peer acts as an RVP for the destination), the method returns the destination itself. The `update_next_RVP()` method is used to update (or create) an entry in the routing table. It is called whenever a message is received. The `update_routing_table()` method is called to update the routing table. It takes as parameter a view that has been received during a shuffle. This method adds an entry in the routing table for each entry in the view and specifies that the RVP for these entries is the peer with which the shuffle was performed. The `decrease_routing_table_ttls()` method is used to decrease the TTL of routing table entries, and purge the expired ones.

```
1  every shuffling_period units do
2    target ← select_gossip_destination(view)
3    if (target is public
       or next_RVP(target) = target) then
4      send ⟨REQUEST, view, self, target⟩ to target
5    elif ((target is SYM and self is PRC)
       or self is SYM) then
6      // Use relaying
7      send ⟨REQUEST, view, self, target⟩
         to next_RVP(target)
8    else
9      // Hole punching
10     send ⟨OPEN_HOLE, self, target⟩ to next_RVP(target)
11     if self is not public then
12       send ⟨PING⟩ to target
13   increase_view_age()
14   decrease_routing_table_ttls()

15 on receive ⟨REQUEST, view_s, src, dest⟩ from p do
16   update_next_RVP(p,p,HOLE_TIMEOUT)
17   if dest ≠ self then
18     // Forwarding
19     send ⟨REQUEST, view_s, src, dest⟩ to next_RVP(dest)
20   elif (src is SYM and self ≠ public)
       or (self is SYM and src ≠ public) then
21     // Use relaying
22     send ⟨RESPONSE, view, src⟩ to next_RVP(src)
23   else
24     send ⟨RESPONSE, view, src⟩ to src
25     view ← merge_and_truncate(view, view_s)
26     update_routing_table(view)

27 on receive ⟨RESPONSE, view_t, dest⟩ from p do
28   update_next_RVP(p,p,HOLE_TIMEOUT)
29   if dest ≠ self then
30     // Forwarding
31     send ⟨RESPONSE, view, dest⟩ to next_RVP(dest)
32   else
33     view ← merge_and_truncate(view, view_t)
34     update_routing_table(view)

35 on receive ⟨OPEN_HOLE, src, dest⟩ from p do
36   update_next_RVP(p,p,HOLE_TIMEOUT)
37   if dest = self then
38     send ⟨PONG⟩ to src
39   else
40     send ⟨OPEN_HOLE, src, dest⟩ to next_RVP(dest)

41 on receive ⟨PING⟩ from p do
42   update_next_RVP(p,p,HOLE_TIMEOUT)
43   send ⟨PONG⟩ to src

44 on receive ⟨PONG⟩ from p do
45   update_next_RVP(p,p,HOLE_TIMEOUT)
46   send ⟨REQUEST, view, self, p⟩ to p
```

Figure 6. The $\mathcal{N}ylon$ protocol.

5 Evaluation

In this section, we report the results of the evaluation of the $\mathcal{N}ylon$ protocol. We simulated a system of 10,000 peers and varied the percentage of peers sitting behind NATs. In short, we show that (i) it achieves uniform random peer sampling, (ii) it induces a reasonable overhead and homogeneously balances the load among natted and public peers, (iii) it achieves reasonable latency, and (iv) it is highly resilient to churn. Before describing these results in more detail, we first present the experimental setup.

Experimental settings. To the best of our knowledge, ex-

[3] If the upper bound is not met, this could cause an entry in the routing table to be stale. We show in Section 5 that the protocol resists to the simultaneous departure of 50% of the nodes. This shows that the protocol would resist to half of the message exchanges simultaneously exceeding the upper bound.

isting p2p simulators do not take into account NATs. We thus developed a Java-based, event-driven simulator that takes into account the four kinds of NATs described in Section 2. Message latency was set to $50ms$, the hole timeout was set to $90s$ (a typical vendor value), and the shuffling period was set to $5s$. Experiments were conducted on a 10,000 peers system. Although we experimented with all four kinds of NATs, experiments with FC NAT are not reported. In practice, as explained in Section 2, peers behind FC NATs behave similarly to public peers as long as they frequently send or receive messages. The distribution we used is the following: 50% of RC NATs, 40% of PRC NATs, and 10% of SYM NATs. Note that we evaluated other distributions and got comparable results. Peers were initialized with a view composed of a random set of public peers to ensure connectivity at the start of each experiment. Unless explicitly mentioned otherwise, the view size is set to 15. All experiments were run with 30 different seeds, the results reported are the average of those 30 runs. Finally, experiments lasted a long enough time to observe, most of the time, a negligible variance. However, any non negligible observed variance is indicated in the graphs.

Correctness. We assessed the correctness of $\mathcal{N}ylon$ with different experiments. Due to space limitation, we do not show graphs for these experiments. First, we checked that there were no network partitions and no stale references in peer views. Moreover, we assessed randomness using the *diehard* test suite for random number generators [14].

Network bandwidth usage. We made experiments to assess the bandwidth usage of $\mathcal{N}ylon$. We computed the average number of bytes per second that each peer sends and receives as a function of the percentage of NATs. Results are depicted in Figure 7. $\mathcal{N}ylon$ consumes less than 350B/s. For comparison, we plotted the average number of bytes per second consumed by the (push/pull, rand, healer) configuration (line "reference"). We also observe that the bandwidth usage does not evolve linearly with the number of NATs. This comes from the fact that the length of RVP chains do not evolve linearly with the number of NATs (see next section).

As explained in Section 4, one of the objectives of $\mathcal{N}ylon$ is to ensure that all peers contribute almost equally to the protocol[4]. This is reflected in Figure 8 which shows the average number of bytes per second sent and received by public and natted peers. We observe that public peers send and receive between 10% and 20% less bytes per second than natted peers. This comes from the fact that (i) all peers can act as RVP, and (ii) public peers do not receive OPEN_HOLE messages for themselves and do not send PONG messages.

Latency. The latency is expressed in the number of hops re-

[4]The only exception being that messages sent and received by peers sitting behind SYM NATs must be relayed by public peers.

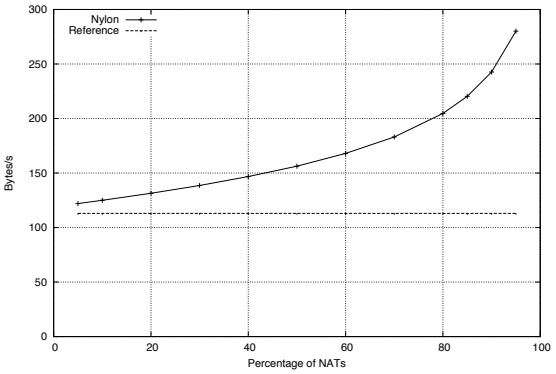


Figure 7. Average number of Bytes/s sent and received by a peer.

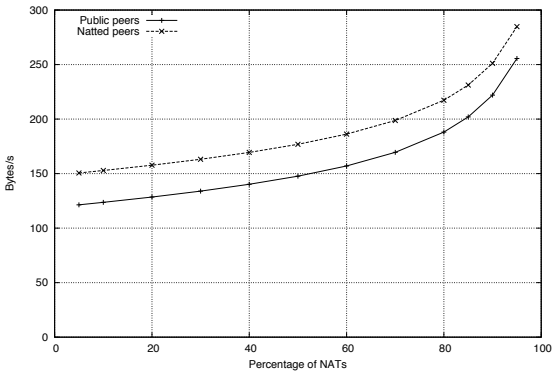


Figure 8. Average number of Bytes/s sent and received by public and natted peers.

quired for a peer to establish a message exchange with the peer it selected for shuffling. The latency towards public peers is obviously equal to one hop. Regarding natted destinations, the protocol requires sending one PING and one PONG message. The main factor impacting latency is the length of the RVP chain used to send the OPEN_HOLE message. Figure 9 shows the average length of RVP chains with two different view sizes (15 and 27). Not surprisingly, we observe that the number of RVPs increases with the percentage of NATs. Note that this increase is not linear, which explains the non-linear bandwidth usage observed for $\mathcal{N}ylon$ in Figure 7. With a view size of 15, the RVP chain length ranges from 1 (with 10% of NAT) to 3. The average relaying latency of $\mathcal{N}ylon$ is thus smaller than 4 hops, which is very reasonable. The fact that the length of RVP chain is small limits the TTL expiration. Finally, an interesting observation is that the average RVP chain length decreases when the view size increases. This result is consistent with random graph theory results on the average distance between peers in a graph as a function of their in and out degree [3].

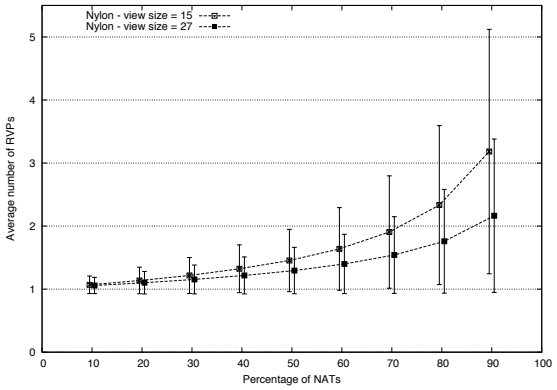


Figure 9. Average number of RVPs towards a natted destination.

Churn resiliency. We conclude this section by an analysis of the behavior of $\mathcal{N}ylon$ under massive churn. The experiments consisted in removing a varying fraction of peers after each of them had performed 500 shuffles. Public and natted peers were removed proportionally to their number in the system. We present results in Figure 10. The different bar types correspond to different percentages of NATs. On the X axis is represented the percentage of peers that are leaving the system. The Y axis represent the size of the biggest cluster 1500 shuffles after the start of the massive churn. We observe that $\mathcal{N}ylon$ is highly resilient to churn. It tolerates the departure of 50% of the peers without partitioning. Even with higher percentage, it exhibits very good performance. This result can be explained by the fact that each peer can be reached by different chains of RVPs at the same time.

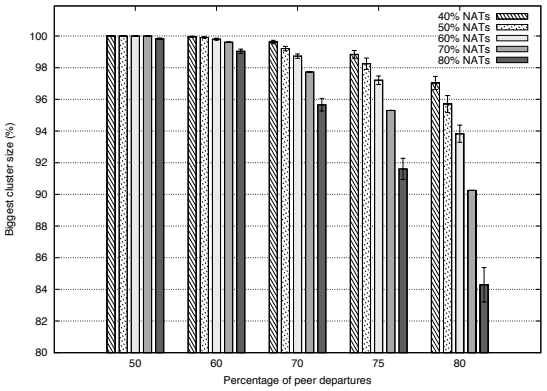


Figure 10. Impact of massive churn on the size of the biggest cluster.

6 Related works and conclusion

Several systems have tried to overcome the problem of limited connectivity [13, 16, 12]. All these systems rely on an explicit structure to route messages on top of a gossip protocol. They use proactive mechanisms to ensure that communication between natted peers is possible under the implicit assumption that the network is fairly static. Some works have also been done in the context of Distributed Hash Tables (DHTs) [11, 17]. Traversing NATs in such systems can be achieved provided that each peer has a relatively static set of neighbors. In addition the structure of DHTs can be used as a natural vector to assign public peers to natted peers. Let us also note that there exist protocols allowing the creation of permanent NAT filtering rules: NAT-PMP [2] and UPnP [8]. These protocols could be used in gossip protocols to avoid the problems caused by the presence of NAT devices. Unfortunately these protocols have limitations. First, they are not supported by all NAT devices. Second, they pose security issues since any application running on a peer can open ports on the NAT device without any approval or notification to the node's user.

Finally, some works have also been done at the network level. For instance, [6] proposes an extension to the routing process of IPv4 in order to take into account NAT devices. Nevertheless, the proposed architecture requires modifications to NAT devices and to end hosts.

While taking into account NATs can be achieved in fairly static systems, this is challenging in the context of highly dynamic systems.

In this paper, we have proposed $\mathcal{N}ylon$, a fully decentralized NAT-resilient gossip peer sampling protocol. $\mathcal{N}ylon$ leverages the fact that in a gossip protocol each peer only

needs to communicate with a subset of peers contained in its view and does actually communicate with an even smaller subset of the peers. It uses a *reactive* hole punching protocol, which creates a path of relay peers to setup communications. Experiments have shown that $\mathcal{N}ylon$ accommodates a large proportion of NATs without impacting the properties of the peer sampling. Moreover, $\mathcal{N}ylon$ evenly spreads the overhead induced by NATs between public and natted peers and is highly resilient to churn.

Acknowledgments

This work was supported in part by the European Commission via the SELFMAN project (IST-2006-34084).

References

[1] M. Casado and M. J. Freedman. Peering through the shroud: The effect of edge opacity on ip-based client identification. In *NSDI*, 2007.

[2] S. Cheshire. NAT Port Mapping Protocol (NAT-PMP). RFC draft, October 2008.

[3] F. Chung, F. Chung, F. Chung, L. Lu, and L. Lu. The average distances in random graphs with given expected degrees. *Internet Mathematics*, 99:15879–15882, 2002.

[4] N. Drost, E. Ogston, R. van Nieuwpoort, and H. E. Bal. Arrg: real-world gossiping. In C. Kesselman, J. Dongarra, and D. W. Walker, editors, *HPDC*, pages 147–158. ACM, 2007.

[5] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, pages 179–192, 2005.

[6] P. Francis and R. Gummadi. Ipnl: A nat-extended internet architecture. *SIGCOMM Comput. Commun. Rev.*, 31(4):69–80, 2001.

[7] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. 2nd Workshop on Real, Large, Distributed Systems (WORLDS 05)*, San Francisco, CA, Dec. 2005.

[8] P. Iyer and U. Warrier. Internet Gateway Device (IGD) Standardized Device Control Protocol. UPnP Forum White paper, November 2001.

[9] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[10] V. King and J. Saia. Choosing a random peer. In *PODC*, pages 125–130, 2004.

[11] M. Lee, H. Choi, and S. Park. Donet-p: A streaming overlay network protocol with private network support. *IEEE Region 10 Conference*, pages 1–4, 30 2007-Nov. 2 2007.

[12] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *Proceedings of the European Dependable Computing Conference on Dependable Computing*, pages 364–379, London, UK, 1999.

[13] J. Maassen and H. E. Bal. Smartsockets: solving the connectivity problems in grid computing. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 1–10, New York, NY, USA, 2007.

[14] G. Marsaglia and W. W. Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 1 2002.

[15] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. J. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *PODC*, pages 123–132, 2006.

[16] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[17] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of SIGCOMM*, pages 73–86, New York, USA, 2002.

[18] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[19] H. Yoshimi, N. Enomoto, Z. Cui, K. Takagi, and A. Iwata. Nat traversal technology of reducing load on relaying server for p2p connections. *Proceedings of the Consumer Communications and Networking Conference*, Jan. 2007.

A.11 Adaptive Deployment on P2P systems

Adaptive deployment in P2P systems with "Salute"

Willy Malvault, Vivien Quéma and Jean-Bernard Stefani

Abstract

This paper presents "Salute" : a peer-to-peer middleware that can manage a potentially large set of concurrent application deployments among a wide and heterogenous pool of resources (the cloud). Salutes combines new and well-known gossiping protocols to build and maintain its infrastructure, so that it inherits their high scalability and robustness properties.

1 Introduction

Salute[1] is a fully distributed framework that manages multi-scaled application deployments in large P2P networks. It is hosted by an unstructured P2P network overlay and its design relies on gossip based protocols [8], making it scalable and robust to peer churn. *Salute* allows users to book any pre-specified subset of nodes that match a *slice specification*. These nodes are then organized into a sub-overlay network called *slice*, where applications can be deployed. The global architecture of Salute is presented in the Figure 1.

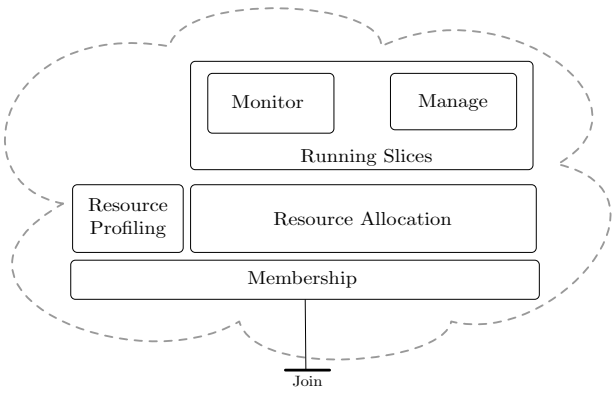


Figure 1: Salute middleware overview

The basis of the middleware infrastructure is an unstructured network overlay built and maintained by a gossip peer sampling service. A resource profiling service categorizes nodes in the overlay according to their capacities and availability. A resource allocation service called *Sarcasm* builds the slices from their specification and books participating peers. Sarcasm also provides an autonomous slice monitoring and managing service that maintains the resource allocation despite node churn in the slice hosting the application. Some scenarios using Salute could be : trading resources in an anonymous P2P community, or allocating resources to volunteer computing, or driving the deployment of several collaborating P2P services over a large collection of workstations.

[1] Slice Allocation in Large UnstrucTured Environments

The document is organized as follow : Section 2 surveys the related work and background technology, Section 3 describes the architecture of Salute, Section 4 details the *Sarcasm* reservation mechanism, Section 5 presents the resource profiling service and Section 6 discusses the evaluation of the Salute middleware.

2 Background and related work

In this section, the related work is presented the advantages and limitations of existing solutions are discussed in the context of large scale service deployment. In addition we give a brief presentation of all gossip-based services used by Sarcasm to provide resource allocation features.

2.1 Distributed Slicing

2.1.1 Attribute based slicing

The attribute based slicing protocols [17, 12, 13] are the first gossip-based solutions proposing to divide an unstructured overlay network into several slices. The slicing specification used by these protocols has the form $\mathcal{S}(Att, N_s)$ where Att refers to an attribute used to compare peers capability, and where N_s is the number of slices requested. Following this specification, peers estimate their ranking among all other members and are assigned to a slice according to that ranking. By instance, the following specification $\mathcal{S}(CPU, N)$ would produce N slices where the lower CPU capable peers belong to the first slice and the most capable ones to the last slice. The simplified idea of this gossip-based protocols is to figure out peers ranking using samples coming from an underlying peer sampling service [19]. Each peer periodically compares its attribute value to a small sample of other peers and deduce the slice it belongs to from these comparisons. As shown in [13], the slicing service can leverage the uniformity of attribute space distribution, resulting in accurate slicing performances. While being accurate, scalable and robust to system dynamism, classic slicing protocols provide a limited service to drive peer-to-peer adaptive deployments. Actually attribute based slicing protocols are not capable to efficiently reconfigure their slicing specification and their specification doesn't take into account the "absolute" size of slices they produce. An example of classic slicing is presented in Figure 2 where the slice specification is $\mathcal{S}(xxx, 4)$. In the scenario A the system is composed of 6 peers. Applying the classic slicing protocol with the specification S leads to the creation of four slices. Two of these slices contain 1 single peer and the two others contain 2 peers. The scenario B describes the evolution of the same system with two new peers. In that scenario the system is composed of 8 peers and the slicing specification remains unchanged. The classic slicing protocol would so allocate 2 peers to each slice. These two scenarios introduce the principle of the classic slicing protocols : the slicing specification is always respected so that the number of slice is independent from the system size, but the slice sizes do depend from the system size. The classic slicing protocols are so not good candidates to provide highly configurable deployment solutions as the size of the slices can't be configured.

2.1.2 Absolute slicing

In [22], Montresor and al. propose an alternative version of the slicing problem called absolute slicing. The goal is to assign a specified number of nodes to a slice and maintain such assignment in spite of churn. The slice specification is defined as follow : $\mathcal{S}(c, s)$ where c is a condition expressed through a first-order logic over the set of peers' attributes and s is the desired slice size. The protocol is a combination of existing p2p services such as peer sampling service, epidemic broadcast protocol and decentralized aggregation protocol. The Figure 3 describes an example of absolute slicing where the

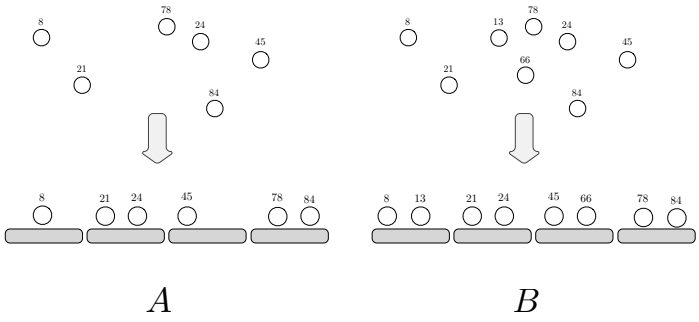


Figure 2: classic slicing

specification is $\mathcal{S}(Att > 22, 2)$ (the peers satisfying the condition $Att > 22$ are in grey on the figure). In the scenario A 6 peers are composing the system. Once the absolute slicing protocol has been deployed two slices are built : one containing two peers that satisfy the slicing specification and one slice containing all other peers. In the scenarios B the system is composed of 8 peers and the resulting slicing is the same as in scenario A : one slice contains two peers that satisfy the specification and one other slice contains all other peers. This example introduces the principle and drawbacks of the absolute slicing protocol, while the slice size is conform to the specification and is independent from the system size, only one slice can be built regardless the composition and the size of the system. The absolute slicing protocol provides good

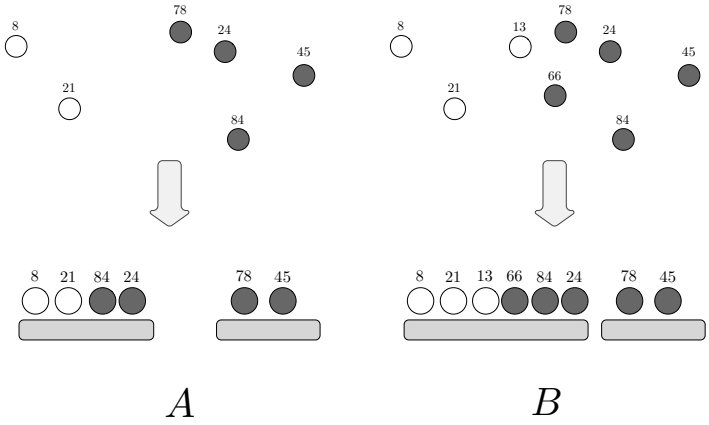


Figure 3: absolute slicing

specification possibilities regarding to the classic slicing protocols but has a major limitation : it can only build and maintain one single slice. For that reason it is not a good candidate for a large scale application deployment solution.

2.2 Synchronization

In [3] authors propose a decentralized solution capable of synchronizing all peers within an unstructured overlay network. This solution is a gossip based protocol that is inspired by mathematical models of flash synchronization in certain species of fireflies. The protocol is scalable and robust to churn. The specification of that protocol is that peers in a distributed system generate periodic local "heartbeat" events approximately at the same time. Therefore the protocol used to maintain the synchronization assumes that peers can access to local clocks that can measure the passage of real time with reasonable accuracy that is, with a small short-term drift. Evaluations demonstrated that under several "voluntarily pessimistic" scenarios and parameter settings, the nodes synchronize their heartbeats to fall in an interval of 1%-10% of the cycle length of the periodic heartbeats. We assume that such a service can easily be incorporated in the set of gossiping protocols that compose the core of the Salute infrastructure. In the next sections of this document, peers are considered to be able to synchronize any events within a given period, particularly to start or end the resource reservations.

2.3 Resource availability

2.3.1 Availability Patterns

While most of studies about resource availability in peer-to-peer systems [10, 5, 21, 24, 14] show that a large portion of peers have truly random behavior, these peers can be classified into several uptime categories, that would be very helpful to drive large scale application deployments. As far as we now, the first availability categorization was presented in [21]. In this work, authors first analyzed the availability history of peers in two largely distributed systems : Microsoft corporate network [6] and the PlanetLab testbed [4]. Mickens et. al. wanted to show the existence of availability patterns. In order to exhibit such patterns, they apply a technique inspired from [10], where the availability of a node is modeled as a binary signal, stating to 0 when the node is offline and to 1 when it is online. Initially Douceur detected diurnal behaviors in [10] by applying a Fourier transform to the availability signal and looking in for spikes in the daily and weekly frequencies. Mickens et. al. categorize peers that pass the test of Douceur into a class called "work week periodic". They found that almost 10% of nodes in the Microsoft corporate network belong to this class, while none of PlanetLab peers do. This statement seems to fit the reality as PlanetLab is mainly composed of long running dedicated servers, while stations composing the Microsoft corporate network are likely to behave in accordance with work period. While studying further the availability of hosts in PlanetLab, Mickens and al. found that 67% of nodes have low frequency availability changes, meaning that their online and offline periods are long running. This uptime class is called "long stretch" and 20% of the Microsoft corporate network also belong to this category. These two uptime classes regroup the periodic availability patterns, but Mickens and al. also defined uptime classes for peer following aperiodic patterns. These peers are called "unstable" and are categorized according to their averaged uptime. Unstable peers are classified into five classes according to their ratio $\frac{uptime}{utime+downtime}$. A peer being available more than 90% of time would belong to the "always on" uptime category, while a peer available less than 10% of the time would belong to the "always off" class. Three other intermediate classes are defined to fit all possible availability patterns : the "unstable 10-50", "unstable 50-70" and "unstable 70-90" uptime classes.
Once existence of availability patterns have been confirmed, Mickens. and al. devise a collection of algorithms providing per peer availability prediction. These algorithms implemented as "predictors" uses the history of peers to forecast their availability. The prediction algorithms used in [21] will not be detailed in this document. Therefore, Mickens and al. show that the predicatability of peers availability depends on the proportion of some uptime classes within the peer-to-peer community. The main

results was that "always on" peers are more accurately predictable and that long stretch peers only support short term predictions. Using the uptime classes defined by Mickens and al. would allow peer-to-peer applications to fit the available resources to their deployment requirements.

2.3.2 Toward an availability aware deployment

Modeling the availability in a large collection of heterogeneous resources is mandatory to drive concurrent application deployments. Actually many peer-to-peer protocols, if not all, are devise to face the dynamism of the peer-to-peer community hosting them. This dynamism is called churn and model the continuous arrival and failure (or departure) of peers. Therefore each peer-to-peer applications has a different tolerance to the churn intensity. To name an extreme case : in FlashBack [9] the peer-to-peer service can be satisfied by a collection of resources reaching an extreme churn of 90% every ten seconds. Few distributed applications are able to work well under such extreme conditions, but this example shows that peer-to-peer applications may have different requirements in term of churn profile. Studying related work about resources availability help us to choose the suitable solution to devise a "churn aware" deployment solution. In particular the Section 3.2 explains how the availability patterns defined in [21] can be used in Salute's slice specification.

3 The deployment environment

This section describes Salutes deployment environment. It first explains how concurrent applications can be deployed on a "cloud like" resource pool and introduces the need of controlled resources allocation. In a second sub-section the slice specification is detailed, and a third sub-section describes the implementation of the Salute middleware infrastructure.

3.1 Application deployment in the cloud

The main goal of Salute is to allow several peer-to-peer applications to co-exist within the same overlay and to maintain them running despite the dynamicity of the system. To reach that goal the peer-to-peer network hosting Salute is divided into several sub-networks called "slices" where client applications can be deployed and run separately. As described later in the following subsection the slicing specification in Salute, allows any client to specify the number of peers it needs to deploy its application. The dynamicity of a peer-to-peer system, better known as "churn", may lead an allocated slice to fall below the requested size after a slice has been booked and an application deployed. The main idea here is to consider that if the application is a peer-to-peer application, it should support such a churn by dynamically redeploying, if and only if the resource allocation service counter-balances the resource loss. In such cases Salute would provide the application with new peers, while others disappear because of the churn. Otherwise the resource pool allocated to that application would decrease as the slice looses peers because of churn, and the client application requirements would not be met. So as to maintain concurrent slices Salute always keeps "safety" peers in a "reserve" overlay network and stops to create new slices when this reserve overlay falls under a certain threshold. To summarize, when peers join the Salute infrastructure, they directly join the reserve overlay and when running slices lacks peers, they request additional ones to the reserve. The Figure 4 illustrates the resource flows triggered by the churn in the different network overlays that compose the system. The white circles symbolize the peers joining the system in the reserve overlay. The grey circles symbolize the peers that are leaving the system from the reserve overlay or from slices S_i, S_j and S_k. This figures helps to understand that the role of the reserve overlay is

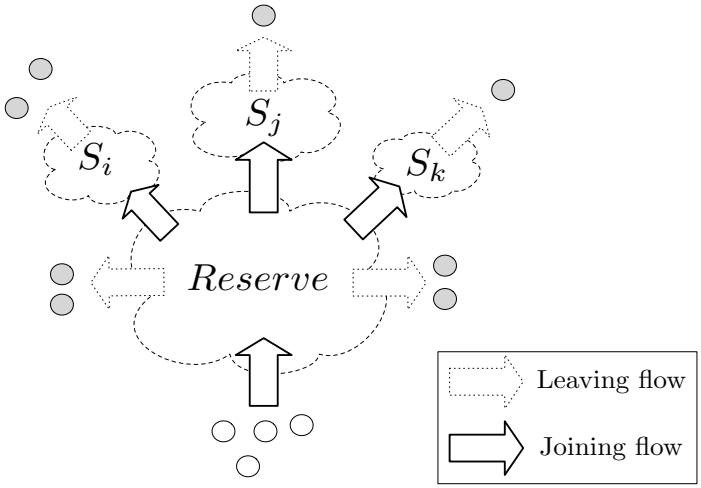


Figure 4: Salute middleware overview

to control the churn so as to distribute the joining resources to leverage the leaving ones.

3.2 Slice specification in Salute

The middleware Salute, allows users to book any set of resources by requesting a slice overlay. The slice specification describes the attribute values, the number of peers that should compose the slice and the expected duration of the slice reservation. The following grammar formalizes the description language used to write such specifications.

$$
\begin{aligned}
R_e &::= \emptyset \mid N \mid R_e + R_e \\
S &::= (F, N_p, L) \\
N_p &::= \text{val} \mid \perp \\
L &::= \text{val} \mid \perp \\
F &::= \text{true} \mid A\ C\ \text{val} \mid \text{F} \wedge \text{F} \\
A &::= \text{CPU} \mid \text{MEM} \mid \text{HD} \mid \text{BDW} \mid \text{UC} \\
C &::= > \mid < \mid \leq \mid = \mid \neq \\
val &::= i \in \mathbb{N}
\end{aligned}
$$

A slice request specification R_e is a set containing at most one sub-specification. A sub-specification S is composed by a collection of filters F, an expected number of peers N_p and a reservation length L. Filters composing F specify boolean constraints that a peer must respect to join the requested slice (by instance, it can define a minimal CPU capacity for peers composing the slice). Considering the reservation duration parameter : if the requester wants to book a set of resources for an unbound time, the parameter D is set to the particular value $\perp$. Otherwise the expected duration D is set with a real integer indicating the expected duration in hour. The number of requested peer N_p is set with a natural integer value. Therefore, there exist a particular type of request, that we call "best effort" requests, consisting in requesting as many peers as possible. Best effort requests are specified by setting the parameter N_p to the particular value $\perp$. In Salute, the slice specification filters can define constraints on the following attributes : the computing capacity of a peer CPU, its memory capacity MEM, its hard disk storage capacity HD, its network bandwidth capacity BDW and its uptime class UC. To improve the relevance of attribute value specification, theses values are classified so as to fall in a pre-specified categories, that are set according to the the global resource profile. The rough idea is to choose a collection of attribute categories, so that the peers attribute distribution in the whole

system would be uniformly distributed among these categories. The establishment of attribute categories is done by studying the system evolution. The study of automated attribute categorization is presented in Section 5. Once a collection of categories has been chosen for a given attribute, categories must be ranked according to the attribute value so that the comparison operators $>$, $\leq$, $>$, $\geq$ can be used to refers to several categories. For instance the expression $CPU > 1$ refers to the set of categories having a rank greater than 1. The algorithms used to rank the attribute categories also depend on the study of the resource profile presented in Section 5. An example of peer categorization is presented in Table 1.

Attribute	Category	ranking
CPU	$Att < 1$ GHz	1
	1 GHz $\leq Att \leq$ 3 GHz	2
	$Att >$ 3GHz	3
...	...	...
UC	$ALWAYS\_OFF$	1
	$UNSTABLE$ 10_50	2
	...	...
	$ALWAYS\_ON$	N

Table 1: Attribute categorization example.

In Table 1 peers CPU attributes are classified into three categories : the first category regroups peers having a CPU frequency lower than 1 GHz, the second category regroups peers having a CPU frequency lower or equal to 1 GHz and greater or equal to 3 GHz and the third category regroups peers having a CPU frequency greater than 3GHz. The second attribute categorization presented in this table sorts peers according to their uptime classes. These uptime classes characterize the peer availability patterns according to categories presented in Section 2.3. In order to familiarize the reader with the Salute slice specification, some examples of specification using the attribute categorization proposed in Table 1 are presented below :

- $R_e = (CPU > 1, 10, 1) + (CPU > 1 \wedge UC \neq 3, 10, 1)$: Looking in Table 1, we found that this specification means that : 20 peers are requested and they must have a CPU frequency greater or equal to 1 GHz (greater than the category having the ranking 1). Then 10 of these peers should not belong to the uptime category : $ALWAYS\_OFF$. Finally, this request has an expected lifetime of one hour.
- $R_e = (CPU > 2, -1, -1)$: the requested slice should contain as many peers with CPU frequency greater than 3 GHz as possible and the duration should be as long as possible.
- $R_e = (true, 100, -1)$: the requested slice should contain 100 peers, whatever their attribute characteristics and the duration should be as long as possible.

With such slicing specifications, Salute can provide highly configurable application deployments, combining the absolute size of a slice with constraints on peers attributes composing it. The duration parameter allows peers to define a date after which they can leave the slice.

3.3 Building a reserve overlay

The reserve overlay and slices are build and maintained by a peer sampling service that connects peers together and provides random peer samples to the gossiping protocols running in those overlays. The peer sampling service used in Salute is Cyclon [26]. Note that several overlays will co-exists, running their own instance of Cyclon and

that a peer is mandatory member of the reserve overlay, while it can be member of **at most one** slice. Salute contains three types of peer regarding to their role in client applications deployment process :

- `free peers` : are only member of the reserve overlay and are available to resource allocation.
- `booked peers` : are member of one slice in which they run a client application and can't be allocated to another task before the slice reservation ends.
- `safety peers` : are only member of the reserve overlay, are not available to resource allocation, but can rescue running slices if churn makes them to fall under the specified size.

As explained in section 3.1 the reserve overlay should stop to allocate peers to new slices when it estimates that the current churn could endanger running slices. In order to prevent client application from over-booking peers, the reserve overlay is configured to contain a minimal percentage of "safety" peers. Our evaluations have shown that a value between 5% and 15% is suitable to maintain all deployed slices, depending on the estimated availability pattern of peers composing the system.
In order to maintain the specified safety peers percentage, peers in Salute use a counting service that estimates the current number of safety peers in the system. This service is presented in Section 4.1.2. When a peer joins the system for the first time or after a downtime period it joins the reserve overlay network by contacting a bootstrapping peer. This bootstrapping peer gives the joining peer its safety percentage estimation, figured out thanks to the counting service. If the estimated safety peer percentage in the reserve is too low : the new peer joins the system as a "safety" peer, otherwise it joins the system as a "free" peer. When a slice reservation ends, the booked peers it contains change their status according to this heuristic, so that if any user detects that the churn drive the system in a configuration where the reserve overlay could no longer feed the running slices, it can stop a running slice to distribute its resources to other slices.

4 The Sarcasm protocol

Sarcasm is the scalable reservation mechanism provided by Salute to book peers, build and maintain a slice overlay from its specification. The simplified principle of Sarcasm is the following : when any peer want to book a set of resources so as to build a slice, it broadcast the requested slice specification to the reserve overlay and free peers that can satisfy this request join the building slice overlay. As soon as the slice is complete, i.e. reaches its requested size, the reservation is started. If there are not enough available resources to satisfy the request, the reservation is eventually canceled and the potentially booked peers are released. In case of concurrency for a given resource, Sarcasm uses a priority mechanism capable to break the tie between any number of concurrent requesters. The different strategies establishing request priority will be discuss in Section 6, while this section focuses on Sarcasm's presentation. The assumptions made to run Sarcasm on an unstructured network overlay are the following :

1. **No cheating** : we do consider only fail stop crashes. Every working peer follows the exact protocol specification (no cheating).
2. **Availability** : each peer runs an availability predictor as described in [21] and can deduce its uptime classes from the output of this predictor. Such availability categorization allow Salute to consider the uptime class of a peer as an attribute, that can compose a slice specification (see section 3.2).
3. **Identification** : peers have a unique identifier $id =< IPAddress, portnumber >$.

The first subsection presents sub-services used by Sarcasm. These sub-services can have several implementation depending on the slicing specification. A second sub-section details Sarcasm algorithms and protocols.

4.1 Devising Sarcasm as a meta-protocol

Initially Sarcasm has been devised so as to book multiple sized sets of resources and build slices overlays with them. Also Sarcasm is presented as a meta-protocol using three sub-services : 1) a request propagation service, 2) a counting service and 3) a membership management service. The implementation of such services depend on the slice specification, by instance the requested slice size, and can provide different qualities of service. This sub-section introduces the role of the three sub-services and discuss the relevance of some possible implementations.

4.1.1 Request propagation service

The request propagation service is a decentralized protocol that broadcasts the reservation request to potential candidates (free peers willing to satisfy the request) so as to book enough resources to build the requested slice overlay network. While potential candidates receive the reservation request, if they can satisfy it, they join the slice by participating to its membership service. As described later in Section 4.2 it is coupled to the counting service so as to stop propagating the request when the requested slice is complete. The "meta-methods" provided by the reservation propagation service are the following :

- **start(***RSpec***)** : is a primitive that starts the request propagation service, the parameter *RSpec* is the specification of the requested slice. This methods is called by the reservation requester.
- **propagate()** : This methods is used to propagate the request to a sub-set of the reserve overlay, so that it can be invoked periodically until the requested slice become complete or abort. This method can be invoked by any peer in the reserve overlay.

Several implementations of the request propagation service can be relevant regarding to the requested slice size. This paper discusses three possible implementations for this service : an epidemic broadcast protocol, a random walk-based solution and a distributed resource discovery service. These implementations are introduced in the following :

Epidemic broadcast : The natural implementation for the request propagation service is an epidemic broadcast protocol like Lpbcast [11]. Such gossiping protocol has good scalability properties and is able to broadcast an information to a very large collection of peers within a relatively small period. But epidemic broadcast could waste a lot of bandwidth as it propagates the requested information to every peer within the peer-to-peer unstructured overlay. In some scenarios, our request propagation protocol could only need to broadcast its reservation request to a small proportion of the reserve overlay, wasting the network resources of other peers. By instance if any user wants to book a dozen of peers among a thousand, the epidemic broadcast protocol would involve in average hundred time as much peers as necessary. The epidemic broadcast would so be a good implementation for large sized slice requests, but it would waste a lot of bandwidth in case of small sized slice requests.

Random walk search : Random walk resource localization has been a topic of main interest in peer-to-peer research area during the beginning of 21st century [1, 7, 25]. One can imagine to implement the request propagation service by looking for potential candidates using a random walk and request resulting peers to join the building slice. A random walk over an unstructured peer-to-peer overlay starts from any

peers and visits another randomly chosen peer periodically, so that after a pre-specified number of hop w, the walk has collected information from a sub-set of averaged size w within an unstructured overlay. Consequently, this kind of protocol can sample an unstructured overlay while saving the bandwidth consumption regarding to the gossiping protocols. Therefore, if the requested resources constitute a small sub-set of the search domain (by instance if there exists only 1% of resources satisfying a request among a whole network), the probability that the random walk visits these resources is very low. Finally, the random walked based solution would be a good implementation for the request propagation service in scenarios where the number of peer satisfying the request constitutes a large proportion of the reserve overlay.

Resource discovery service : In [2] authors present Sword : a distributed resource discovery service that could fit the request propagation service's requirements under certain scenarios. The simplified idea of Sword is to collect all peers attribute values and store them into a distributed infrastructure (DHT or smaller group of peers). Sword also handles complex queries, allowing users to describe resources as a topology and to impose some constraints on the relative networking capabilities of peers. A simple way of implementing the request propagation service using Sword would consist in translating Salute's slice specification into Sword's specification language, then query Sword with that specification and individually request resulting peers to join the building slice. This solution has consequent limitations as Sword don't scale well over several thousand of peers to monitor and needs dedicated peers to run it. Therefore Sword's capacity to handle complex queries with network topology constraints is an interesting feature that cannot be provided with other solutions. Finally one can follow the Sword deployment strategies presented in [2] to install it within the Salute middleware.

To conclude, the epidemic broadcast is a good candidate to be the default implementation of the request propagation service. Effectively the gossip based solution would succeed with every possible scenario, sometime wasting resources, but scaling up to million of peers. The random walk based solution could be carefully used to request a small slice composed of "common" peers and tough save some networking resources. The resource discovery service is harder to use in Salute as it needs to be installed as a client application over the Salute's peer-to-peer community, but it can provide better slice specification capabilities than ones provided by the other implementations.

4.1.2 Counting service

The counting service is used during all the life cycle of a slice : first it counts the number of joining peers, as the slice is building, so as to decide when the slice is complete. Then the counting service is still running within the overlay of a complete slice in order to detect if the slice size falls under a certain threshold because of the churn. Notice that a peer belonging to two network overlays would run an instance of the counting service for each overlay, in particular for the reserve overlay and for the potential slice it belongs to. The counting service is able to count several "types" of peers within a given overlay. These types refers to peers attribute categorization (cf Section3.2) so that each peer knows if it belongs to that type or not. The counting service API used by Sarcasm is the following :

- **start(PS)** : this methods starts a counting service instance on the overlay PS given in parameter. The parameter PS gives a reference allowing the counting service to use the associated peer sampling service.
- **addType(T)** : this method adds a type T of peer to be counted within the overlay associated to the counting service instance. This type should be defined according to a set of attribute filters as described in Section 3.2
- **removeType(T)** : this method removes a type T of peer to be counted.

- **getSize(T)** : this method returns the number of peers of type T within the associated overlay.
- **stop()** : this methods stops the counting service and should be invoked when a peer leaves the associated network overlay only.

There exist two main implementations of totally decentralized counting protocols [18, 20] suited for unstructured overlay networks, therefore we also discuss in the following the relevance of a centralized solution :

Gossip based aggregation protocol : the reference implementation of the gossip based aggregation protocol is [18]. In this work authors present a gossiping protocol that use global aggregation technics to figure out the number of peers within an unstructured overlay network. The particularity of this protocol is that every peers composing the overlay has is own estimation of the overlay size. Evaluations presented in [18] estimates the maximum deviation of such estimations from the real value to 10%, meaning that every peers knows its overlay size with a 10% deviation in worst cases. Like every gossiping protocol, this implementation has good scalability properties but it suffers from a high bandwidth consumption as it involves every peers composing the targeted overlay.

Random Walk based protocol : the reference implementation of random walk based counting protocols is [20]. In this paper authors present a protocol able to estimate a network overlay size using a set of random walks. The particularity of this protocol is that the network size estimation is centralized around a requester that is the source of the random walks used to evaluate the network size. This protocol as been devised so as to save bandwidth regarding to the gossip based solution, and it actually reached that goal as described in [20]. Therefore in the case of the counting service, this protocol needs to be coupled with a broadcasting protocol (by instance Lpbcast [11]) so as to broadcast the resulting size estimation to all peers within the targeted overlay. The use of another gossiping protocol in this case would cancel the bandwidth savings made by the random walk based protocol. For this reason, we choose to evict this implementation as a candidate for the counting service, but its low bandwidth consumption property remains a good reference for the evaluation of Salute's counting service.

Centralized protocol : A centralized approach can be considered to implement Salutes's counting service. If we consider small sized overlays, the centralized solution would work as follow : an elected peer (by instance the reservation requester) would know all other peers belonging to the overlay and would monitor them (using ping or hearth-beat technics) so as to keep the count updated. This solution is clearly not scalable but it has the advantage to give an exact estimation, regarding to random walk based protocol or gossiping protocol. Even if this implementation is less challenging in term of scalability, some users can need it to deploy some distributed application on a small but strict number of peer.

To summarize, the default implementation for the counting service is based on the gossip based aggregation protocol, because of its simplicity and scalability properties. The random walk based method is abandoned but would be used to compare its bandwidth consumption to gossiping solution one's. Finally the existence of centralized solution has been discussed, this solution can provide a strict counting service (instead of having a 10% standard deviation) for small slice overlays.

4.1.3 Peer Sampling service

The peer sampling service is in charge of maintaining the overlay network composing an allocated slice. As explained in Section 3.3 the peer sampling service is also deployed

so as to maintain the reserve overlay. It provides peer samples that are then used by every gossiping protocols running in Salute. Notice that if a peer is member of a slice, it runs two peers sampling services : one for the reserve overlay and one for its slice. This two peer sampling services are totally independent. The main peer sampling service implementations are [11, 15, 16, 26, 19] and will be compared in another version of this document. The API of the peer sampling service used in this document is the following :

- **bootstrap(Π)** : this method is used by a peer to join an unstructured overlay network. The parameter Π is a set containing at least one identifier that can be used as bootstrap peer to join the overlay. This set can contain several bootstrap peer identifiers, so that a peer using this set can join the requested overlay if one of the bootstrap peers is no longer available. If any peer P_i uses this method with its own identifier as parameter ($\Pi = \{P_i\}$) a new network overlay will be created. This method can be used in Salute by a slice requester to create the associated slice overlay network.
- **getPeers(N)** : this method returns N randomly chosen peers that are member of the overlay on which the method has been invoked.
- **stop()** : this method stops the peer sampling service. It should be invoked when a peer leaves the associated network overlay.

To summarize, Sarcasm uses three services : a propagation service, a counting service and a peer sampling service. Let now detail the Sarcasm protocol itself.

4.2 Detailing the protocol

This subsection details some pseudo code that illustrate the algorithms used by Sarcasm. This pseudo code is presented as an object oriented language and refers to the Salute's service presented in Section 4.1. The peer sampling services are accessible via two objects : the object *Reserve_sampler* implementing the peer sampling service of the reserve overlay and the object *Slice_sampler* implementing the peer sampling service of the local peer's slice overlay. The request propagating service is accessible via the object *Propagator*. Counting services are accessible via two objects : *Reserve_counter* or *Slice_counter*, regarding to the network overlay they are running on. Notice that these objects do implement the meta-methods described in Section 4.1.

4.2.1 Requesting a slice with Sarcasm

When a user wants to book a set of resources so as to deploy its own application on it, it uses the following Sarcasm methods : `Propagate_Reservation`($< Req_{id}, RSpec, T >$). The parameter Req_{id} refers to the requester identifier, $RSpec$ is the requested slice specification (cf. Section 3.2) and T is the time at which the slice has been requested. The reservation request and the slice are so identified by the couple $< Req_{id}, T >$. The pseudo code of this method is described in the Figure 5.

```
1: upon Propagate_Reservation(< Req_id, RSpec, T >) do
2:    Slice_sampler.bootstrap(P_i)
3:    Slice_counter.start(Slice_sampler)
4:    for S_i ∈ RSpec do
5:       Π ← {Req_id}
6:       Propagator.propagate(< REQ, Req_id, RSpec, T, S_i, Π >)
7:       Slice_counter.addType(S_i.Filters)
8:    end for
9:    MonitorSliceStatus()
10: end upon
```

Figure 5: Pseudo code of Sarcasm reservation request method

In this method, the slice requester creates a new overlay for the slice and initializes a counting service running on that overlay (Lines 2 and 3). Then, for each sub-specification S_i contained in the slice specification $RSpec$ (Lines 4 to 7), the requester propagates a message containing the following informations : $< REQ, Req_{id}, RSpec, T, S_i, \Pi >$ in the reserve overlay network. Where REQ is a flag indicating that the message is a reservation request (and not a rescue request) and where Π is a set containing a small number of peers that are already member of the slice. This set Π is used to contact the slice overlay or to choose any bootstrapping peer to join that network (cf Section 4.1.3). Initially the slice is only composed by the requester, also for the first request propagation $\Pi = \{Req_{id}\}$. Then the requester adds a new type to the slice counting service with the method $Counter.AddType()$. Once the reservation propagation has been initialized within the loop, the requester monitors the slice during its building process using the slice counting service. The method `MonitorSliceStatus`() is described in Section 4.2.3. Therefore the next section introduces the algorithms used by free peers to join a slice when they receive a reservation request, the slice monitoring heuristics comes immediately after that section.

4.2.2 Joining a slice overlay in Sarcasm

When any peer propagates a reservation request, every other peer can receive it even if it is not willing to join the requested slice. The reservation request handling process is composed by three phases : first it checks if the request has already been received, then it choses to join the requested slice or not and finally it checks if it should continue to propagate the request or not. The Figure 6 gives the pseudo code of the algorithm used by peers when they receive a reservation request.

When a peer P_j receives a reservation request $< REQ, Req_{id}, RSpec, T, S_i, \Pi >$, it checks if it already received any reservation request with this identifier (Line 2 and 3). If it does, the request is discarded, otherwise the request is treated. If P_j is free and can satisfy the request sub-specification S_i then it joins the slice (Lines 4 and 5). The procedure `JoinSlice`$(< Req_{id}, T >)$ (Lines 11 to 20) executes the following actions : it saves the slice identifier in a variable called $joinedSlice$, it modifies P_j's status from "free" to "booked" and initializes the slice's peer sampling and counting services (Lines 14 to 18). P_j would now become member of the slice overlay and would be able to monitor its status according to heuristics describe in Section 4.2.3. Whether it choses to join the slice or not, P_j should then decide if the reservation request has to be further propagated or not. In order to check if the requested slice is complete, it saves the set of members Π into a local variable *slice_to_contact* and sends a message $< SL\_CHECK, Req_{id}, T >$ to one peer randomly chosen in Π. Let call P_c the peer that receives this request. P_c would reply with appropriated informations if it effectively is a member of the slice identified by the couple $< Req_{id}, T >$. In case of mistake P_c would invalidate the request (Line 26). Therefore, if the request is valid P_c sends back its estimation of the slice completeness plus a set Π' containing different identifiers that ones in Π. Changing the "bootstrapping peer set" Π to Π' ensures that the request for slice completeness are uniformly distributed among the members of a slice. With these informations, P_j would now if it should further propagate the reservation request and would do so if needed (Line 30 and 31). In case of invalid answer from P_c, P_j would retry to contact a member the requested slice. The retry process is not detailed here, but we can assume that P_j would try to contact each peer in the saved set *slice_to_contact*. If none of this peers sends back any valid answer, P_j would finally decide not to forward the reservation request.

To summarize, when a peer request a slice in Salute, it broadcasts a reservation request. Free peers join the building slice when they receive that request. Therefore every peers, even not free forward the request until the requested slice is complete or

```
1: upon Receive_Reservation_Request(REQ, < Req_id, RSpec, T, S_i, Π >) do
2:   if < Req_id, T, S_i > ∉ alreadyReceived then
3:     alreadyReceived ← < Req_id, T, S_i >
4:     if MyStatus = FREE ∧ LocalAttributes.Satisfy(S_i.Filters) then
5:       JoinSlice(< Req_id, T >)
6:     end if
7:     slice_to_contact ← Π
8:     send < SL_CHECK, Req_id, T > to P_c ∈ Π
9:   end if
10: end upon

11: procedure JoinSlice(< Req_id, T >)
12:   joinedSlice ← < Req_id, T >
13:   MyStatus ← BOOKED
14:   Slice_sampler.bootstrap(Π)
15:   Slice_counter.start(Slice_sampler)
16:   for S_i ∈ RSpec do
17:     Slice_counter.addType(S_i.Filters)
18:   end for
19:   MonitorSliceStatus()
20: end

21: procedure ReceiveSliceCheckRequest(< SL_CHECK, Req_id, T >) from P_r
22:   if joinedSlice =< Req_id, T > then
23:     Π ← Slice_sampler.getPeers()
24:     send < True, Req_id, T, hasStarted, Π > to P_r
25:   else
26:     send < False, Req_id, T, ⊥, ⊥ > to P_r
27:   end if
28: end

29: upon ReceiveSliceCheckAnswer(< is_valid, Req_id, T, is_complete, Π' >) do
30:   if is_valid ∧ ¬is_complete then
31:     Propagator.propagate(< Req_id, RSpec, T, S_i, Π' >)
32:   else if ¬check_valid then
33:     retry(slice_to_contact)
34:   end if
35: end upon
```

Figure 6: Receiving a reservation request

until it aborts. The inner slice protocol that decides when a slice is complete, when it ends or even when it should abort is presented in the following section.

4.2.3 Maintaining a slice with Sarcasm

Once slices are built thanks to the reservation propagation service, an inner slice protocol monitors and maintains the slice reservation. As explained in Section 3.1 slices are prone to suffer from the system dynamism and would probably need to ask for additional resources during their reservation time. The inner slice protocol uses the counting service describes in Section 4.1.2 to decide when a slice is complete and when it needs to ask for additional resources by sending rescue request to the reserve overlay following the protocol described in Section 3.3. When a peer joins a slice overlay it invokes the routine `MonitorSliceStatus()` that is in charge to monitor and maintain the slice overlay network. The pseudo code of this method is described in the Figure 7.

The monitoring routine periodically checks the number of peers composing the slice overlay to whom the local peer belongs to. The frequency of the routine is fixed according to the period Δ that could be modulated according to the churn estimated by the slice. Actually the period Δ should be relatively short when the slice is filling so as to detect the slice completeness and stop the request propagation. However, this period could be extended once the slice is complete, as the slice size would not longer be expected to vary rapidly. The modulation of such period is not discussed further

```
1: procedure MonitorSliceStatus()
2:   for each Δ seconds do
3:     complete ← True
4:     for each S_i ∈ JoinedSlice.RSpec do
5:       if Slice_counter.getSize(S_i.Filters) < S_i.Nb_Peer then
6:         complete ← False
7:         toRescue ← S_i
8:       end if
9:     end for
10:    if complete then
11:      if ¬ hasStarted then
12:        hasStarted ← True
13:      end if
14:    else
15:      if hasStarted then
16:        Π ← Slice_sampler.getPeers()
17:        Propagate_Rescue_Request(< RESCUE, Req_id, RSpec, T, toRescue, Π >)
18:      else if CurrentDate > joinedSlice.CancelationDate then
19:        LeaveMySlice()
20:      end if
21:    end if
22:    if CurrentDate > joinedSlice.EndReservationDate then
23:      LeaveMySlice()
24:    end if
25:  end for
26: end

27: procedure LeaveMySlice()
28:   Slice_counter.Stop()
29:   Slice_sampler.Stop()
30:   if Reserve_counter.getSize(SafePeer) < SafetyThreshold then
31:     MyStatus ← SAFE
32:   else
33:     MyStatus ← FREE
34:   end if
35: end
```

Figure 7: Monitoring a slice overlay.

in this paper. In the routine, each peer checks if the estimated network size is greater than the size requested by the slice specification. If the counting service estimates that the slice network size is big enough regarding to its specification, the variable *hasStarted* is set to the boolean value $True$ (Line 12). This variable is initially set to $False$ and indicates if the slice reached at most once its requested size. This variable is also used by the slice members to indicate to other peers that the slice is complete, so as to stop the reservation propagation as described in the Figure 6 (Line 23).
When the estimated slice size is lower than the requested one, the slice is either not yet complete or some of its members has left the overlay because of the churn. If the slice is not yet complete, the routine checks that the cancelation deadline has not been reached (Line 18). If it does, the local peer leaves the slice and goes back to the reserve overlay either as a free peer or as a safe peer. Such cancelation mechanism prevent reservation requests to run indefinitely when there is not enough resources to complete the requested slice. If the slice needs extra peers because of the churn, the local peer broadcasts a rescue request to the reserve overlay using the method `Propagate_Rescue_Request` so as to get additional resources. This methods simply chooses a sub-set of size $\log N$ (where N is the size of the system) of peers within the reserve overlay and send them the following message : $< RESCUE, Req_{id}, RSpec, T, S_i, Slice\_sampler.getPeers() >$. The algorithm used by peers to handle the rescue requests is described in the Figure 8. Finally the slice monitoring routine checks is the Slice reservation has ended (Line 14) and allows the local peer to leave the slice overlay and go back to the reserve overlay. Notice that the methods `LeaveMySlice()` used by a peer when leaving the slice it belongs to does respect the reserve overlay joining rule as described in Section 3.3 : the joining peer

chooses its status regarding to the proportion of safe peer in the reserve overlay (Lines 30 to 34).

To summarize, the slice monitoring service is distributed among all peers involved in the slice. Notice that peers are supposed to have synchronized clocks thanks to the synchronization protocol running within the reserve overlay (cf Section 2.2). they are so synchronized on epochs of length Δ and choose a random offset value to run their monitoring routine within that epochs. This offset allows the distribution of the routine "start-time" of peers to be uniformly distributed among all slice members. This mechanism distributes peers monitoring actions among a whole epoch length, so that the slice is continuously monitored by its peers. One peer could only broadcast one rescue request per epoch. Let now introduce the method `Receive_Rescue_Request`() that handles the rescue requests and manages them so as not to flood the reserve overlay. The pseudo code of this method is described in the Figure 8.

1: **upon** `Receive_Rescue_Request`($< RESCUE, Req_{id}, RSpec, T, S_i, \Pi >$) **do**
2: **if** $< Req_{id}, T, S_i > \notin alreadyReceived$ **then**
3: $alreadyReceived \leftarrow < Req_{id}, T, S_i >$
4: **if** MyStatus = FREE $\wedge$ MyStatus = SAFE $\wedge$ LocalAttributes.Satisfy($S_i.Filters$) **then**
5: `JoinSlice`($< Req_{id}, T >$)
6: **end if**
7: **end if**
8: $slice\_to\_contact \leftarrow \Pi$
9: `send` $< SL\_CHECK, Req_{id}, T >$ `to` $P_c \in \Pi$
10: **end upon**

Figure 8: Rescuing a slice with Sarcasm

Like for a reservation request, if P_j already received a rescue request with the slice identifier $< Req_{id}, T, S_i >$, it discards it. Otherwise the message $< RESCUE, Req_{id}, RSpec, T, S_i, \Pi >$ is treated. This simple filtering rule allows the reserve overlay not to be flooded by concurrent rescue requests that could be sent within the same epoch for a same slice. The variable *alreadyReceived* keeps track of the messages received during a period equal to Δ, so that a slice can renew a rescue request for each new epoch. The receiving peer P_j only decides to treat the rescue request if its status is either safe or free. Any peer treating a rescue request immediately joins the slice and then choose to forward this request according to the checking mechanism described for the reservation request propagation in Figure 6.

4.2.4 The deadlock problem

In case of concurrent requests the version of Sarcasm presented so far can suffer from deadlocks. The Figure 9 describes a scenario where two concurrent requests lead to a deadlock state and none of them can succeed.

In this scenario two peers R_1 and R_2 are propagating a slicing request, respectively SR_1 and SR_2. Both slicing specifications are requesting 5 peers of an arbitrary type $\mathcal{T}$ (peers of type $\mathcal{T}$ are green in the Figure 9). There exist 6 free peers of type $\mathcal{T}$ within the system, so that only one of the two requests can be satisfied. The request R_1 succeeds to book 4 peers of type $\mathcal{T}$ and the request R_2 succeeds to book the 2 remaining peers of type $\mathcal{T}$. None of the two requested slices are complete and both reservation requests will abort, while in theory one of them could have succeed. This scenario is realistic as no assumption on request concurrency are made in Salute. Consequently, deadlock states can't be predicted and the reservation mechanism Sarcasm must provide a solution to prevent concurrent reservation requests to lead to a deadlock state. The solution chosen in Sarcasm is to add a priority mechanism to reservation requests, so that the priority can break the tie between two concurrent reservations requesting the

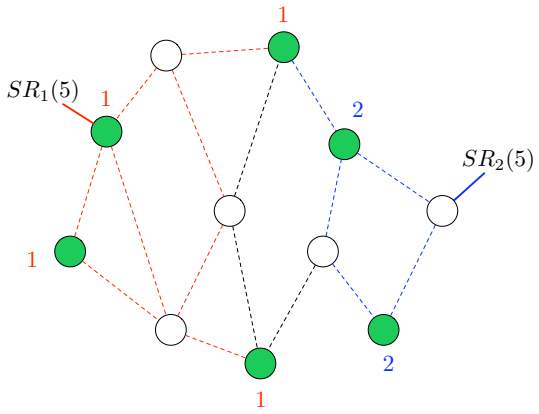


Figure 9: Deadlock state with two concurrent reservation requests

same set of resources. The Figure 10 depicts a scenario where the peer R_1 has the priority on the peer R_2.

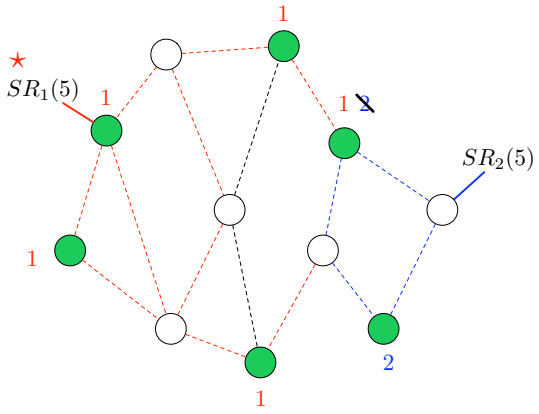


Figure 10: Using priority with concurrent request

This scenario starts from the deadlock state depicted in Figure 9. As the slice request SR_1 has priority on the request SR_2, it would steal a resource to the blocked reservation SR_2 so as to complete the slice requested by the peer R_1. Using this simple priority mechanism ensures that at least one request would succeed when there are enough resources to satisfy it. Therefore if the priority doesn't take into account the number of resources that a peer has already booked, this mechanism can lead to starvation. By instance, in the scenario studied in Figure 11 if the peer R_1 always has priority on peer R_2 : it could always steal resources to peer R_2 and that peer would suffer from starvation. A simple way to provide a fair priority mechanism is to give the priority to the peer having the smallest number of successful reservations in its history, breaking the tie with the peer identifier if needed. One can imagine a lot of policies defining the priority to be used in case of concurrent request. Such policies would have a significant effect on the system efficiency as it will be discussed in Section 6. Before discussing the evaluation of Salute, a second version of Sarcasm is presented in the

following subsection, where the algorithms implementing the reservation priority are introduced.

4.2.5 A priority aware version of Sarcasm

The simplified idea of the priority aware version of Sarcasm is to allow any peer P_j to join a slice Sl_i regarding to 3 new conditions : *i*) P_j has already joined a slice Sl_j, *ii*) Sl_j is not complete yet and *iii*) the slice Sl_i has the priority on Sl_j. Notice that the peer P_j is stolen (by the slice Sl_i) from the Sl_j resource pool, thus implementing the resource stealing mechanism described in the previous paragraph. To simplify the description of the protocol, peers that belong to a slice that is not complete are called pending peers, as they are waiting for the completeness of the slice. Any pending peer can so be stolen and would become booked only when the slice it belongs to become complete. A detailed version of the Sarcasm protocol taking into account request priority is depicted in the Figure 11. The methods that are changed regarding to the first version of the protocol are the following : `Receive_Reservation_Request`(...), `JoinSlice`(...) and `WatchForSliceStart`() . The methods not described in the Figure 11 remain the same as in the first version of the protocol.

```
1: upon Receive_Reservation_Request(< Req_id, RSpec, T, S_i, Π >) do
2:   if < Req_id, T, S_i > ∉ alreadyReceived then
3:     alreadyReceived ← < Req_id, T, S_i >
4:     if MyStatus = FREE ∧ LocalAttributes.Satisfy(S_i.Filters) then
5:       JoinSlice(< Req_id, T >)
6:     end if
7:     if MyStatus = PENDING ∧ LocalAttributes.Satisfy(S_i.Filters) then
8:       if joinedSlice.hasLowerPriority(< Req_id, T, S_i >) then
9:         LeaveMySlice()
10:        JoinSlice(< Req_id, T >)
11:      end if
12:    end if
13:    slice_to_contact ← Π
14:    send < SL_CHECK, Req_id, T > to P_c ∈ Π
15:  end if
16: end upon

17: procedure JoinSlice(< Req_id, T >)
18:   joinedSlice ← < Req_id, T >
19:   MyStatus ← PENDING
20:   Slice_sampler.Start(Π)
21:   Slice_counter.Start(Slice_sampler)
22:   for S_i ∈ RSpec do
23:     Slice_counter.addType(S_i.Filters)
24:   end for
25:   MonitorSliceStatus()
26: end

27: upon MonitorSliceStatus() do
28:   for every Δ second do
29:     .....
30:     if hasStarted then
31:       MyStatus ← BOOKED
32:     end if
33:     .....
34:   end for
35: end upon
```

Figure 11: Dead lock aware Sarcasm protocol

As in the previous version of Sarcasm, when a peer P_j receives a reservation request $< REQ, Req_{id}, RSpec, T, S_i, \Pi >$ for a slice Sl_i, it checks if it already received any reservation request with this identifier (Line 2 and 3). If it does, the request

is discarded, otherwise the request is treated. If P_j is free and can satisfy the request sub-specification S_i then it joins the slice (Lines 4 and 5). The procedure `JoinSlice`($< Req_{id}, T >$) (Lines 11 to 20) executes the same actions as in the previous version of the protocol excepting that when a peer joins a slice, it modifies its status from "free" to "pending" instead of "booked". Notice that a sub-routine is added to the routine `MonitorSliceStatus`() (Lines 31 to 33). This sub-routine is in charge of modifying the status of the local peer from "pending" to "booked" when the inner slice monitoring service detects that the slice has been complete. When the peer P_j is pending while receiving a new reservation request for a slice Sl_i, it checks if Sl_i has the priority on the slice Sl_j, that corresponds to the slice P_j already belongs to. If and only if the slice Sl_i does have the priority, then P_j leaves its former slice (Sl_j) and joins the new one (Sl_i) (Lines 7 to 12). Finally, whether P_j choses to join the slice Sl_i or not, it decides to propagates further the reservation request according to the same protocol as one depicted in the Figure 6.

To summarize, Sarcasm is a service that allows any user to specify a resources reservation and request the allocation of associated peers within a large peer-to-peer system. This service builds sub-network overlays called slices that correspond to the resources reservation specifications made by users. Sarcasm books peers composing a slice overlay, so that no peer can belong to more than one slice. This service, also monitors the composition of allocated slices so as to rescue them if their sizes fall under a certain threshold because of the churn. Finally Sarcasm is able to manage concurrent requests thanks to a priority mechanism. Client applications in Salute would use the slices build by Sarcasm to deploy their own code. The following section discusses the establishment of a distributed resource profile used by these client applications to adjust their slice specifications to the system composition. The Section 6 would then discuss the relevance of different request priority policies that can be used to optimize the system's performances.

5 Resource Profiling service

As explained before, the reservation request policy chosen in Salute is to cancel a resource reservation if the associated slice doesn't reach its requested size within a bounded period. Consequently, if a user requests more peers than ones available within the reserve overlay, then its request would never succeed. It is thus important for the middleware Salute to provide a reliable resource profiling service. This service can be used by users to evaluate the composition and number of available resources within the system and adjust their reservation request specifications according to the truly available resources. This section discusses the specification and the feasibility of such a profiling service. It first introduces the profile used in Salute to model the resource availability and then presents some ideas to automatize the configuration of this profile.

5.1 Profile of the system

In wide peer-to-peer systems the heterogeneity of peers and the churn make it difficult to know the exact composition of the system at a given time. Actually the quantity of information would be linear to the number of peers in the system and that information would rapidly evolve as the peers join and leave the system. Salute proposes to use attribute categorization as introduced in Section 3.2 in order to synthesize the attribute profile of peers into a "system attribute profile". Therefore this system attribute profile is not sufficient to drive the configuration of the middleware Salute. A second sub-profile giving the global availability of resources is then added to the system profile to complete the information given by the categorization of peers attributes. The following paragraph explains how the attribute profile is built, while a second paragraph present

the global availability profile.

5.1.1 Categorization of peers attributes

The system attribute profile can be seen as a map associating a given attribute category to the number of peers in the system that fall into this category. The categories present in this map should be the same as ones used to specify the set of filters composing a slice specification (cf Section 3.2). This categories are : the computing capacity of a peer CPU, its memory capacity MEM, its hard disk storage capacity HD, its network bandwidth capacity BDW and its uptime class UC. The Table 2 illustrates an example of attribute profile.

Attribute	Category	Size
CPU	$Att < 1$ GHz	1520
	1 GHz $\leq Att \leq$ 3 GHz	2600
	$Att >$ 3GHz	880
...	...	...
UC	$ALWAYS\_OFF$	500
	...	...

Table 2: Example of system profile.

As shown in this table, categories are sorted according to the peer attribute they rely on. The number of categories for a given attribute can be different from an attribute to another but should remain small (i.e. at least a dozen of categories per attribute), so as to keep the system profile size reasonable. Then for each category the profile indicates the estimated number of peers falling into it (called category size). By instance, the Table 2 describes a system composed by approximately 5000 peers, where 1520 of these peers have a CPU capacity less than 1 GHz, 2600 peers have a CPU capacity less or equal to 3 GHz and greater or equal than 1 GHz and finally, 880 peers have a CPU capacity greater than 3 GHz [2]. The set of categories composing the system attribute profile is called the attribute profile configuration and is the same for every peers. The global profile is build using Salute's counting service (cf Section 4.1.2) on the reserve overlay. A peer type is added to the counting service for each category available in the attribute profile configuration, so that peers can build their own system profile overview by asking to the counting service the estimated size of each attribute category.

The categorization process, i.e. setting the attribute profile configuration, should be organized such that the profile of peers attributes is uniformly distributed among all categories. If the distribution of peers attributes is such that one category among a dozen contains 80% of peers, the specification capacity for Salute users would be reduced by the inaccuracy of the profile. This distribution means that users have a dozen of choices, that is great, but these choices concern only 20% of the peers composing the system. Actually, if all attribute categories have approximately the same size, the more categories there are in the attribute profile, the more accurate is the resource profiling service. The attribute profile configuration should so deal with a tradeoff between the accuracy of the profiling service and the cost of the counting service building the attribute profile. Effectively the cost, in term of bandwidth used, of the counting service would grow in function of the number of categories to count. Another problem relative to the profile configuration is to choose the appropriate category to add to the profile, while maintaining the "approximatively uniform" distribution of peers

[2] Notice that the values given in the Table 2 are arbitrary set to illustrate the principle of attribute categories. These values are so not relevant to describe a really deployed system.

attributes within the different categories. Some clue allowing the implementation of an automated profile configuration are given in Section **??**.

The resource profiling service of Salute gives a global resource profile to the users, including the distribution of peers into the different uptime classes defined in [21]. Therefore one can need some extra informations that could not be extracted from the availability profile of peers, such that the minimal number of peers in the system within a given period or the average number of peers available at the same time. The following paragraph discusses the relevance of a global availability profile that could provide these extra informations.

5.1.2 Global availability profile

In addition to the informations given by the attributes categorization, the resource profiling service of Salute also provides a global availability profile. This second "sub-profile" models the global availability of peers involved in the Salute peer-to-peer community. Some properties on the peers availability such that the minimum number of peers encountered during the past history of the system are provided by this new profile. These informations cannot be extracted from the resource attribute profile and is therefore very useful to manage the middleware as explained in Section 6.

In order to synthesize the global availability profile, some informations worked out by the counting service running on the reserve overlay has to be periodically collected and stored so as to build a global availability history of the system. For simplicity reasons the correlation of peers availability with other attributes are not included in this history. This simplification implies by instance that the availability history contains the number of peers available within the system at a given time, but not the number of that peers having a greater CPU capacity than 3GHz. The informations collected by the counting service for a given time are : *i)* the total number of peers running in the system, *ii)* the percentage of free peers in the whole system, *iii)* the percentage of safety peers in the whole system and *iv)* the number of running slices in the system. The period at which the informations are collected could be on the order of a quarter of hour. This value of 15 minutes is proposed because it corresponds to the finest public peer-to-peer traces of our knowledge [23]. These traces could so be used as input to evaluate the resource profiling service. Moreover, that period is small enough to allow an accurate profiling of availability regarding to the expected duration of reservations that can spread from several minutes to several days or even weeks. The Table 3 gives a sample of a global availability profile.

Date	System Size	% free	% safe	nb slices
23/09/2009 15:00:00	50000	50%	10%	15
...	...	...	...	...
23/09/2009 17:30:00	55000	20%	7%	80
...	...	...	...	...
23/09/2009 18:15:00	65000	3%	3%	110
...	...	...	...	...
23/09/2009 19:00:00	55000	15%	9%	75
...	...	...	...	...
23/09/2009 02:30:00	45000	75%	10%	10

Table 3: Sample of global availability profile.

This sample describes the evolution of a system over a period of 15 hours and 30 minutes. For each date at which the informations has been collected from the counting

service, the system size, the percentage of free and safe peers and the number of slice are given. These informations indicates, by instance, that Salute resources are likely to be intensively used in the end of the afternoon, whereas few resources are likely to be used during the night [3]. Such informations could then be used to manage Salute and optimize the system performances as it would be further discussed in Section 6.

In Salute, each peer has to build its own system profile, simply using the counting service to collect the information needed to work out that profile. The following subsection discusses the configuration of the system, in particular the heuristics that adapt the attribute profile categories to the estimated available resources.

5.2 Profile configuration

As explained before the attribute profile can be configured through the categorization of peers attributes. Remember that if all categories have roughly the same size, the more categories there are, the more accurate the attribute profile is. The routine called `MonitorAttributeCategories`() is in charge of adjusting the number of categories for a given attribute so as to optimize the attribute profile regarding to the attribute distribution of the available peers. The execution of this routine (i.e. who runs the routine) is discussed in the Section **??**, while the Figure 12 describes the pseudo code used by this routine.

```
1: upon MonitorAttributeCategories() do
2:   for every Δ second do
3:     for Att ∈ [CPU, MEM, HD, BDW, UC] do
4:       Cat ← Profile.get(Att)
5:       ideal ← Reserve_counter.getSize(ALL) / |Cat|
6:       Cat' ← ∅
7:       for C ∈ Cat do
8:         Csize ← Reserve_counter.getSize(C)
9:         if Csize > 2ideal ∧ |Cat| < 12 then
10:          Profile.split(C)
11:        else if Csize > ideal/2 then
12:          Cat' ← Cat' ∪ C
13:        end if
14:      end for
15:      Profile.merge(Cat')
16:    end for
17:  end for
18: end upon
```

Figure 12: Heuristic configuring the attribute categorization.

The method `MonitorAttributeCategories`() is periodically executed following a period Δ. As this routine monitors the informations given by the counting service, it should be executed with a period such that the counting protocol has updated these informations between two executions. Thus we admit that the counting protocol is able to renew its informations accurately within a period Δ. This routine analyzes the attribute profile structure, for each attribute Att considered in the Salute slice specification (Line 3) it collects the list Cat of categories associated to this attribute (Line 4) and figures out the ideal size of each category according to the number of categories in that list and to the system size (Line 5). This size is the one that maximizes the accuracy of the attribute profile for a given number of categories, it is stored in the variable $ideal$. Then the routine checks the size of each category $\mathcal{C}$ in the list Cat (Lines 7 and 8). If the size of $\mathcal{C}$ is at least twice as big as the ideal size, and the number of categories associated to the attribute Att doesn't exceed a dozen, this category is split into two new categories by the method $Profile/split(C)$ (Lines 9 and

[3]Notice that the values given in the Table 3 are arbitrary set to explain the interest of the global availability profile. These values are so not relevant to describe a really deployed system.

10). This method is not described in this document but it simply splits a category into two parts equally sized, as it would use the dichotomy principle. By instance the categories design by the expression $1 < X < 3$ would be divided into the following categories : $1 < X < 2$ and $2 \leq X < 3$. If the size of $\mathcal{C}$ is at most twice as small as the ideal size, it is stored into the list Cat' to be potentially merged with other categories. Once all categories associated to the attribute Att has been checked, the method $Profile.merge(Cat')$ merges the categories stored into the list Cat' so that the resulting category doesn't exceed the ideal size $ideal$.
This simple heuristic allows to automatically configure the attribute profile so as to eliminate the useless categories (i.e. the categories containing a small number of peers) and to split the useful ones (i.e. split the categories containing a lot of peers). The effect of such periodical re-configuration would be to refine the attribute profile until the distribution of peers attributes is (roughly) uniformly distributed among the different categories.

To summarize, the Salute profiling service provides a synthesized view of the global system properties threw two sub-profiles : *i)* a global attribute profile that autonomously categorizes the peers' attributes and a global availability profile that builds an history of peers availability. This profile is then used by client applications to evaluate the properties of available resources and adjust their reservation specification in function of the profile's informations.

6 Salute performances

This section explains how the middleware Salute will be evaluated, in a first part it describes some relevant evaluation criterions and in a second part it presents our expectations about Salute's performances.

6.1 Evaluation criterions

The evaluation of Salute depends of three criterions : *i)* the healthiness of the system, *ii)* the accuracy of the resource allocation service and *iii)* the fairness of the request reservation satisfaction.

6.1.1 Healthiness

As explained before in Section 3.1 the safety peer percentage plays a role of main importance in the slice requester satisfaction. Effectively, it ensures that the reserve overlay contains enough peer to rescue the running slices if needed. In Salute, the health of slices is evaluated thanks to four status : *i)* the slice is `satisfied` if it contains at least 100% of the number of requested peers, *ii)* it is `healthy` if it contains at least 95% of these peers, *iii)* it is `endangered` if it contains a number of peers stating between 90% and 95% of the requested slice size and *iv)* it is `dead` if the number of peers it contains fall under 90% of the requested slice size. Basically, when peers belonging to a slice detects thanks to the inner slice monitoring service (cf. Section 4.2.3) that their slice is no longer satisfied, they request rescues to the free and safety peers available in the reserve overlay. Combining all running slices health status gives a global system health overview. This overview can be figured out by every peers using the global availability profile, assuming that the counting service collects the relevant informations. In the following the system is considered to be healthy if all of the running slices are healthy too. The objective of the middleware is to maintain the running slices so that they are always healthy despite the evolution of the global system availability.

6.1.2 Accuracy

The accuracy of Salute measures the ratio of global resource allocation over global resource request. The best accuracy that Salute can achieve is 1, meaning that the total number of peers allocated to the different running slices is strictly equal to the number of requested resources. Therefore the core of Sarcasm, the protocol building the slices, is based on a counting service that is not perfectly accurate (cf. Section 4.1.2). Thus the slices build by Sarcasm may not contain exactly the number of requested peers [4]. This inaccuracy would not be a problem for a single slice as we suppose that this inaccuracy is reasonable for client applications, but at the global scale of the system this inaccuracy could lead to waste resources. The previous paragraph explained that if Salute allocates less peers than requested to a slice, the health of the system would decrease and so would the satisfaction of client applications. On the contrary, if Salute allocates more peers than requested to the deployed slices, then the health of the system would be good but the resource allocation capacity would decrease. By instance, if 90% of the peers composing the system are allocated to different slices while only 60% have been initially requested by users, the accuracy of the system is $\frac{2}{3}$. Such an accuracy means that the middleware can only allocate $\frac{2}{3}$ of the available resources, that would be quite inefficient. The Figure 14 illustrates the impact of an inaccurate slice allocation service.

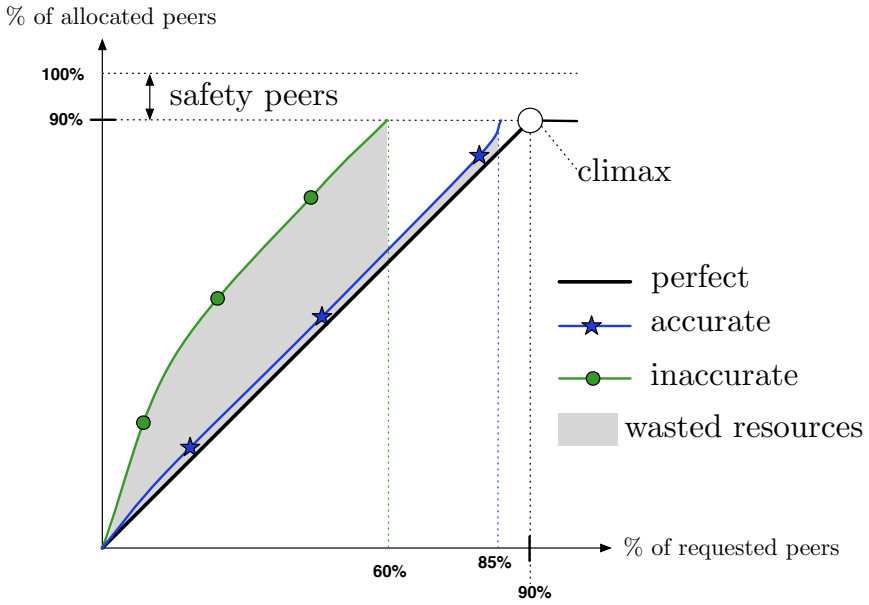


Figure 13: Accuracy of Salute

This figure gives the global percentage of allocated resources (on the ordinate) in function of the global percentage of requested resources (on the abscissa). On this picture the perfect system accuracy is represented by the function $f(x) = x$ so that the total number of peers that are allocated to different slices is strictly equal to the number of requested peers. The climax corresponds to the theoretical highest accuracy that can be reached by the systems, meaning that all peers but safety ones have been successfully allocated and every running slices contains the exact number of requested peers. It is easy to understand that the climax is a theoretical concept that is not willing to happen because of the churn of peer-to-peer systems on which Salute is willing to be hosted. The Figure 14 also describes an inaccurate system : the area

[4] we empirically estimated the slice size error to 5% on average with a small standard deviation

colored in grey represents the amount of resources that are wasted by the allocation service. The inaccurate system represented in this figure allocates in average 30% more peers than requested. As a consequence it can only allocate 60% of the available resources. A realistic estimation of an accurate system is also shown. Within this last system few resources would be wasted, but as it is discussed in Section 6.2, the expected accuracy of Salute could be not as accurate as this system.

6.1.3 Fairness

The fairness criterion would ensure that the total amount of requested resources would be equally distributed among the requesters. Intuitively, a resource allocation protocol would be fair if it distributes as many resources to all requests. Therefore Salute has to deal with concurrent requests having different specification regarding the number of requested resources. By instance a fairness problem could occur when Salute has to choose between satisfying a large number of small request instead of one large requests. If Salute always gives the priority to smaller requests, then a large request would never succeed in a highly concurrent environment. On the contrary if Salutes always gives the priority to biggest requests, users would oversize their request (i.e. request more resources than needed), so as to get the priority on concurrent requests. Salute is said to respect the fairness criterion if it never uses the requests size to set the priority used by Sarcasm to break the tie between several concurrent requests (cf. Section4.2.5).

To summarize, Salute should always respect the healthiness and fairness criterions. The Salute efficiency can be evaluated regarding its accuracy under several scenarios in which the healthiness and fairness criterions are respected.

6.2 Expected performances

This sub-section describes a synthesis of the evaluation done so far with the middleware Salute and explains the resulting expected behavior of this resource allocation service. In order to implement and validate the algorithm composing it, the reservation protocol Sarcasm (cf. Section 4) has been simulated under various scenarios. The churn given as input for these simulations has been extracted from All-pairs PlanetLab Ping Data [23]. The simulation protocol is voluntarily not presented in this document, but the main results are presented in the following. A first part discusses the configuration of the safety peer percentage and a second part describes the expected behavior of Salute in a large scaled, dynamic and heterogeneous environment inspired from Stribling's PlanetLab traces [23].

6.2.1 Efficient configuration of the safety peer percentage

The percentage of safety peers within the system has a non-negligible impact on its performances and its configuration should be chosen with care in order to optimize the allocation service. Intuitively, if 50% of the peers in the system are safety peers, then the probability that any slice could not be rescued is very low as there exist one safety peer to replace each booked resource. On the contrary the reservation possibilities are small as only 50% of the peers can be booked. A good system configuration would be a safety percentage high enough to preserve the system health and low enough to optimize the system performances. In order to bootstrap the system, the default value of the safety percentage would be 10%. Effectively some simulations not presented in this document have shown that if the global size of the system doesn't vary (i.e. there are as many peers joining the system as peers leaving it) a safety percentage of 10% is enough to respect the healthiness criterion. Therefore the churn given as input to these simulations doesn't contains unpredicted events such that the departure of 10% of the peers within the same time. As a consequence, a system running Salute with

10% of safety peers would not be able to guarantee the good health of its slices if unpredicted events occur. Notice that a system willing to maintain slices despite such an extreme churn would use a big safety percentage and would so reduce the system performances. The cost of extreme churn prevention is very high and is delegated to client applications. Effectively if Salute's clients are willing to run critical applications on a peer-to-peer community having a high unpredicted churn, they would oversize their reservation specification so as to manage themselves the cost of such prevention.

6.2.2 Expected behavior

Using the configuration discussed in the previous paragraph, the expected behavior of the middleware Salute states between the behavior of an inaccurate (i.e. inefficient) system and an unhealthy one. The Figure 14 illustrates the expected behavior of Salute.

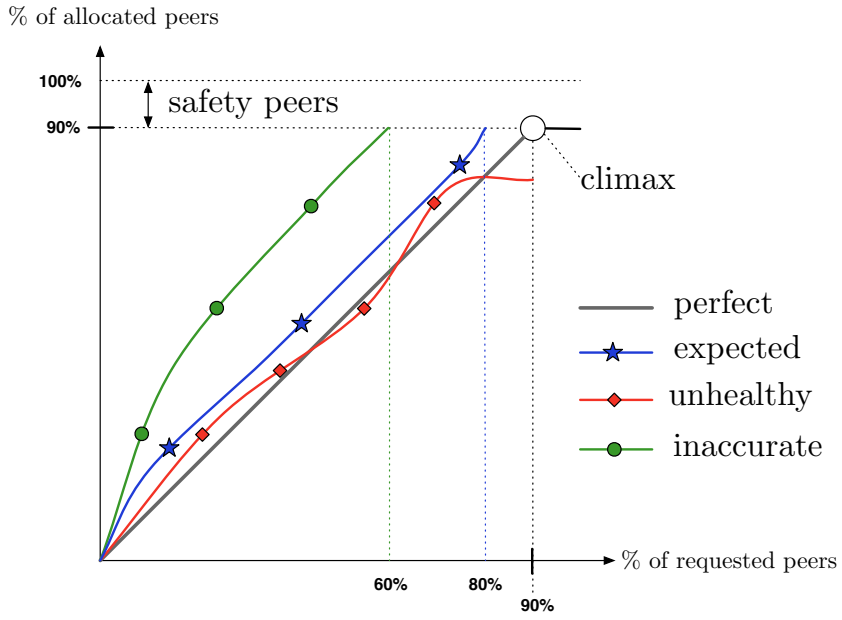


Figure 14: Accuracy of Salute

Under several scenarios differentiated by the number of concurrent requests and the size distribution of these requests the accuracy of Salute allows approximately 80% of the available resources to be allocated to slices. In addition the safety peer percentage of 10% empirically evaluated is sufficient to maintain the system health.

7 Conclusion

To summarize, this document presents Salute : a decentralized middleware providing a resource allocation service in peer-to-peer environments. Salute can be used by peers to request a well specified set of resources and book them so as to deploy their own distributed applications on it. The profiling service gives a synthetic view of the system properties and availability so that client applications can use these informations to adjust their reservation specifications. The protocol Sarcasm builds some sub-network overlays corresponding to the reservation specifications. These sub-overlays are called slices and are maintained by the Salute's infrastructure despite the churn of the environment hosting them.

The reservation mechanism Sarcasm has been successfully evaluated threw simulations and provides reasonable performances according to the evaluation criterions described in this document. Therefore the profiling service has not been evaluated as we do not succeed yet to collect sufficient informations from really deployed peer-to-peer systems. The next step in the development of Salute is to provide a realistic input to the profiling system to validate the set of services provided by the peer-to-peer middleware Salute.

Acknowledgments

This work has been supported in part by the European Commission via the SELFMAN project (IST-2006-34084).

References

[1] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *CoRR*, cs.NI/0103016, 2001.

[2] Jeannie R. Albrecht, David L. Oppenheimer, Amin Vahdat, and David A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. Internet Techn.*, 8(4), 2008.

[3] Özalp Babaoglu, Toni Binci, Márk Jelasity, and Alberto Montresor. Firefly-inspired heartbeat synchronization in overlay networks. In *SASO*, pages 77–86, 2007.

[4] Andy C. Bavier, Mic Bowman, Brent N. Chun, David E. Culler, Scott Karlin, Steve Muir, Larry L. Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating systems support for planetary-scale network services. In *NSDI*, pages 253–266, 2004.

[5] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *IPTPS*, pages 256–267, 2003.

[6] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS*, pages 34–43, 2000.

[7] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for load change detection. In *SIGMETRICS*, pages 272–273, 2002.

[8] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, 1987.

[9] Mayur Deshpande, Abhishek Amit, Mason Chang, Nalini Venkatasubramanian, and Sharad Mehrotra. Flashback: A peer-to-peerweb server for flash crowds. In *ICDCS*, page 15, 2007.

[10] John R. Douceur. Is remote host availability governed by a universal law? *SIGMETRICS Performance Evaluation Review*, 31(3):25–29, 2003.

[11] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[12] Antonio Fernández, Vincent Gramoli, Ernesto Jiménez, Anne-Marie Kermarrec, and Michel Raynal. Distributed slicing in dynamic systems. In *ICDCS*, page 66, 2007.

[13] Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, and Robbert van Renesse. A fast distributed slicing algorithm. In *PODC*, page 427, 2008.

[14] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P. Anderson. Mining for availability models in large-scale distributed systems:a case study of seti@home. 2009.

[15] M. Jelasity, W. Kowalczyk, and M. van Steen. Lightweight probabilistic broadcast. *Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science*, 2003.

[16] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware*, pages 79–98, 2004.

[17] Márk Jelasity and Anne-Marie Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing*, pages 117–124, 2006.

[18] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *ICDCS*, pages 102–109, 2004.

[19] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[20] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *PODC*, pages 123–132, 2006.

[21] James W. Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *NSDI*, 2006.

[22] Alberto Montresor and Roberto Zandonati. Absolute slicing in peer-to-peer systems. In *IPDPS*, pages 1–8, 2008.

[23] J. Stribling. All-pairs planetlab ping data. http://www.pdos.lcs.mit.edu/~strib/pl_app/.

[24] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Internet Measurement Conference*, pages 189–202, 2006.

[25] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *Peer-to-Peer Computing*, pages 102–109, 2003.

[26] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. Network Syst. Manage.*, 13(2), 2005.

A.12 Decentralized Transactional Collaborative Drawing

Decentralized transactional collaborative drawing

Jérémie Melchior[1], Boris Mejías[2], Yves Jaradin[2], Peter Van Roy[2], Jean Vanderdonckt[1]
[1]Information Systems Unit
[2]Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
{$firstname.lastname$}@uclouvain.be

Abstract

When multiple users collaboratively edit a vector image, avoiding conflicts requires synchronizing exclusive access to the objects of the image. This synchronization needs a true concurrency control algorithm. One of the most common strategy to achieve this synchronization is to use a centralized architecture where a single server becomes the transactional manager. Unfortunately, a central point of control is also a single point of failure. This paper proposes a decentralized architecture based on a peer-to-peer network providing decentralized transactional support with replicated storage. As a consequence, there is a gain in fault-tolerance and the transactional protocol eliminates the problem of network delay improving usability and network transparency. The same result can be applied to text edition and other collaborative editing tasks.

1. Introduction

There are many software applications supporting collaborative work, such as drawing, text editing or software development. Collaborative work can be done synchronously or asynchronously. In the later case, the participants make their modifications on their local copy without direct interaction with the other participants. Once the changes are made, they are committed to the global state. In the former case, which is the focus of this paper, all participants are concurrently working on a shared working space. Such scenario requires continuous synchronization of the participants in order to avoid conflicts. One way of achieving such synchronization is by letting the participants lock the part of the shared space they want to modify, granting exclusive access to that part. Since all participants can take any lock, having a single point of control make sense, resulting in the classical client-server architecture. Unfortunately, it is well known that having a single point of control also means having a single point of failure, because the whole application relies on the stability of the server.

Transdraw [5] is a distributed collaborative vector-based graphical editor with a shared drawing area. Each user runs the application and joins a server to get access to the shared area. When someone is drawing in this area, feedback is sent to other users reflecting the action. In addition, TransDraw uses a transactional protocol to allow users to make optimistic changes on the drawing with immediate conflict resolution. This feature eliminates the problem of performance degradation caused by network latency and it is a crucial property of TransDraw. The synchronization and storage of the global state is done on a server which centralizes the control of the work flow. When users modify an object on the drawing, they request exclusive access for it, which may succeed or fail depending on the behaviour of the other users. All this is reflected graphically in the shared drawing space.

As we have mentioned, a problem of TransDraw, due to its centralized architecture, is its dependency on the server. If the server crashes the work is lost, and the application will not run until the server is rebooted.

Peer-to-peer networks have the nice property of being self-organized, fault-tolerant and fully decentralized. We propose in this paper to redesign the transactional protocol of TransDraw to overcome the problem of the single point of failure. In order to do that, we use Beernet [2], a structured peer-to-peer overlay network providing a fault-tolerant distributed transaction layer with replicated storage. Every time a user attempts to modify a graphical object, this modification will be done inside a transaction with a different transaction manager, which is replicated to allow the transaction to finish in case of failure of the manager. Unfortunately, this fault-tolerance mechanism is not free. Replication requires a higher usage of network resources increasing latency of transactions, but the optimistic approach for starting the modification of an object counteract the latency. We consider this a small drawback because functionality of TransDraw is fully respected and there is an important gain in fault-tolerance.

What follows is a more detailed description of TransDraw and related works in sections 2 and 3. Beernet is described in section 4. The core of the proposal is explained in section 5, being followed by conclusions.

2. TransDraw

2.1 Description

Transdraw is a collaborative vector drawing tool created by Donatien Grolaux using transactions[5]. The toolbar provides, not only the traditional tools of vector editing (eg. lines, ellipse, rectangles), but also a pair of tools supporting collaboration. As soon as a user selects an object, a request is sent to the server for the corresponding lock. However, the user is permitted to edit the object optimistically before the server can answer the request. The optimistic nature of the operation is visually presented to the user by red selection handles. When the server grants the lock, the transaction on the object is committed and the user can continue to edit the object in exclusive mode, indicated by black selection handles until he deselect it at which time the lock will be returned. If the lock was already held by another user, the server has to refuse it to the user and the transaction is aborted. The user see the modification he did optimistically undo themselves and the object is deselected.

A user can also manage explicitly his locks by using the *"take lock"* tool, for example to make a complex reorganisation of the drawing, involving several individual objects. He then has to release the locks manually using the flashing *"release-locks"* button.

In order to prevent starvation which could happen as simply as by a user inadvertently selecting every object before taking a rest, a lock stealing mechanism is provided. The *"steal lock"* tool make a request to steal a lock to the server which forwards it to the current owner of the lock. This user then as a few seconds to accept or reject the stealing of her locks. On timeout, the stealing is considered accepted. Once accepted, the previous owner notifies the server to forward the lock to the stealer.

2.2 Example scenario

Figure 1, presents the view of two users working on the same drawing, each in his own window. Bob, on the right, had the top ellipse selected long enough for the server to grant him the lock has can be seen by the black selection handles around it. Alice, on the left has just tried to select this ellipse. After a, normally brief, period during which she was able to do optimistic changes to this ellipse, her transaction is aborted, and she is notified of it by the disappearance of her selection and the red dot on the ellipse which will

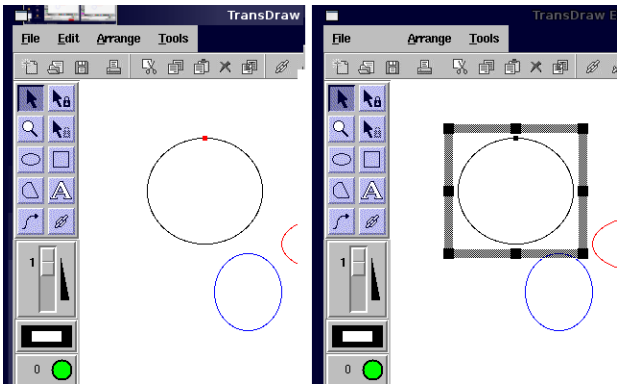


Figure 1. Alice, on the left, see a locked and non-editable ellipse while Bob has it is selected and editable.

blink a few times to explain that Bob is a currently editing this object.

The diagram in Figure 2 describes a possible continuation of the scenario in which Alice steals the lock from Bob to perform the update she wants. Alice ask to steal the lock to the server. Since Bob currently has the lock, the server ask Bob whether he allows his lock to be stolen or not. This is shown to Bob as two blinking buttons at the bottom of his edition window as we can see in Figure 3. If Bob allows his lock to be stolen, either explicitly or by ignoring the request long enough, he loose selection of the object and possession of the lock and the server transfer them to Alice.

Of course, all of this assumes that the server does not crash...

3. Related works

There are some applications that already support collaboration in different ways. We describe and comment some of them briefly.

3.1 BOUML

Some researchers have released an application to provide an easy to use and free UML tool, named BOUML[8]. It allows drawing diagrams and generating code in multiple languages. The tool has been developed as a multiuser application in a sequential way. Each user of the application must choose an identifier which allows working on some diagrams. The work may be done in parallel but there is not any feedback on other users work as there is no support for concurrent work. There are many problems with the tool. The lack of feedback prevents user to know what others are doing and to see their changes. It is also impossible to know

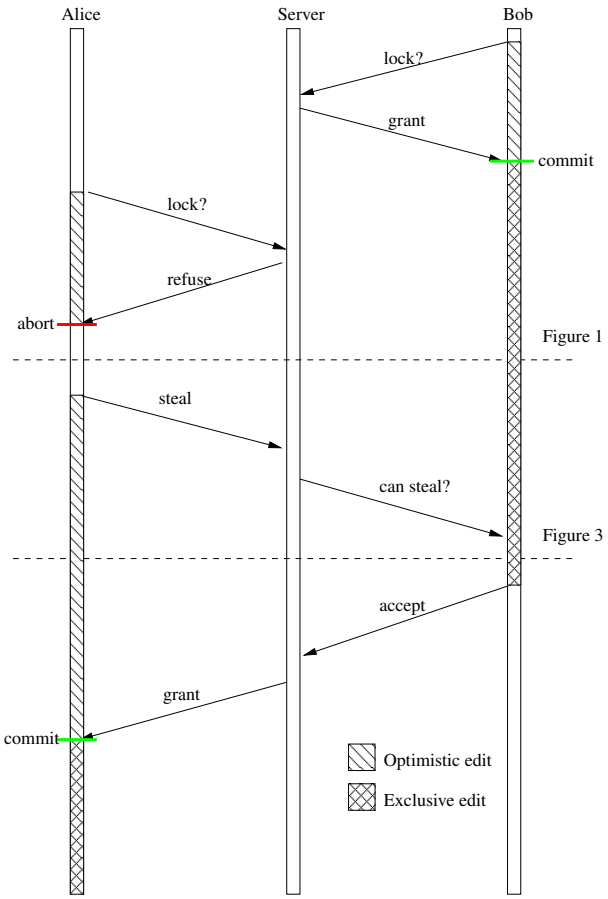


Figure 2. Scenario of complex interaction

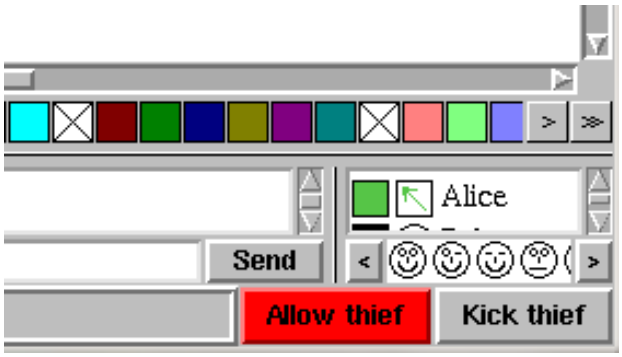


Figure 3. Bob is asked whether he allows his lock to be stolen.

which files are currently being modified or that have been modified and saved. There can be conflicts when saving the project. When users are working collaboratively, the work of a user will be saved but not all the modification of other users. This leads to irreversible lost work without any warning. Another problem is the impossibility to lock part of the work to prevent modification from another user.

3.2 Gobby

Gobby is a free text-editor that allows collaborative work [1]. It supports multiuser parallel edition on multiple documents and a multiuser chat. A user has to start a session and create the documents, he will host the server needed to centralize the information. Other users must choose a name and a color and connect to the server host. The collaboration between all the users is simple thanks to the feedback brought to users with colors. A list of users allows knowing the color of each editor. The application has the ability to recognize patterns of many different text formats and enable syntax coloration. As the BOUML application, Gobby does not support any lock of some part of the text and all the users can edit what they want. This is not a major issue since other users can observe the changes in real time and the team work may rely on trusted users. Nevertheless there is a problem when the server crashes. All the unsaved modifications can be saved by another user but the whole process of creating a server and joining the server must be restarted.

3.3 Google Docs

Google Docs [3] is an online office suite that allows multiple users to modify the same file at the same time. On particular feature, similar to TransDraw, can be seen on spreadsheets. One a user is modifying a cell, this one is coloured differently as in any single user spreadsheet application. When other users connect to Google servers to edit the same file, then, the cells they select will appear with a different colour on the view of the other users, and with a tag identifying the user. Instead of locking the cell, changes are save incrementally using versioning. Google Docs uses also a centralized architecture because everything is control at Google side. But, there is a very important difference. There is not only one server to rely on, but a set of servers with replicated information, so if a server crashes, another one takes over. Of course, these are only conjectures about Google's back-end.

4. Decentralized transactional DHT

Beernet [2] is a structured overlay network providing a distributed hash table (DHT) with symmetric replication.

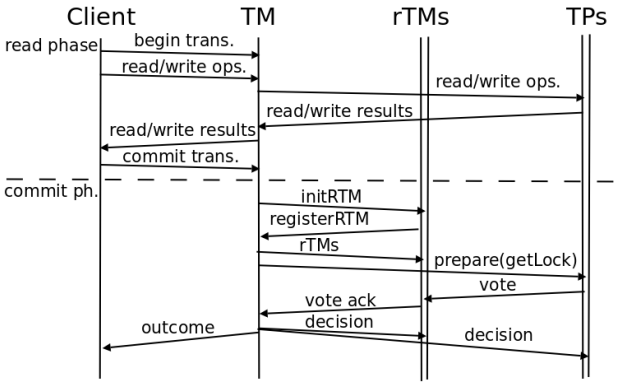


Figure 4. Paxos consensus protocol for distributed transactions.

Peers are self-organized using the relaxed-ring topology [6], which is derived from Chord [10], with cost-efficent ring maintenance and self-healing properties. Data replication is guaranteed with a decentralized transactional protocol allowing the modification of different items within a single transaction. The transactional protocol implements a Paxos-consensus algorithm [7, 4], with requires the agreement of the majority of peers holding the replicas of the items. We will focus on the transactional layer of Beernet because it will be our mean to decentralize TransDraw.

Figure 4 describes how the Paxos-consensus protocol works. The client, which is connected to a peer that is part of the network, triggers a transaction in order to read-/write some items from the global store. When the transaction begins, the peer becomes the transaction manager (TM) for that particular transaction. The whole transaction is divided in two phases: *read phase* and *commit phase*. During the *read phase*, the TM contact all transaction participants (TPs) for all the items involved in the transaction. TPs are chosen from the peers holding a replica of the items. The modification to the data is done optimistically without requesting any lock yet. Once all the read/write operations are done, and the client decides to commit the transaction, the *commit phase* is started.

In order to commit the changes on the replicas, it is necessary to get the lock of the majority of TPs for all items. But, before requesting the locks, it is necessary to register a set of replicated transaction managers (rTMs) that are able to carry on the transaction in case that the TM crashes. The idea is to avoid locking TPs forever. Once the rTMs are registered, the TM sends a $prepare$ message to all participants. This is equivalent to request the lock of the item. The TPs answer back with a $vote$ to all TMs (arrow to TM removed for legibility). The vote is acknowledged by all rTMs to the leader TM. Like that, the TM will be able to take a decision if the majority of rTMs have enough information to take exactly the same decision. If the TM crashes at this point, another rTM can take over the transaction. The decision will be *commit* if the majority of TPs voted for commit. It will be *abort* otherwise. Once the decision is received by the TPs, locks are released.

The protocol provides atomic commit on all replicas with fault tolerance on the transaction manager and the participants. As long as the majority of TMs and TPs survives the process, the transaction will correctly finish. These are very strong properties that will allows us to run TransDraw on a decentralized system without depending on a server.

5. Decentralized TransDraw

Our conjecture about the way Google Docs is designed in order to provide fault-tolerance is strongly based on replication and the possibility of replacing a crashed server with another machine. Not having Google's infrastructure, we can achieve replication and fault-tolerance by building TransDraw on top of a peer-to-peer network, and by decentralizing the synchronization of locks and data storage. Our proposal is to build TransDraw on top of Beernet.

Peers are self-organized using the relaxed-ring topology implemented by Beernet. Data is stored using the DHT with symmetric replication. The transactional layer provides synchronized access to the shared state solving conflicts due to race conditions. But the Paxos-consensus protocol as described in section 4 is not sufficient to provide exactly the same functionality of TransDraw as it was described in section 2. The main difference lies on the moment where the locks are granted. As it is currently, locks are granted too late for TransDraw, because it is not possible to inform users about the intention of the others.

The first modification we have to do to the transactional protocol is to allow eager locking request. One idea is to request the locks when read/write operations are sent to the transaction participants during the *read-phase*. If locks are not granted, the transaction is immediately aborted. The problem introduced by this modification is that if leader TM crashes after requesting the locks, there is no rTM yet to take over the transaction, and items would be locked forever. Considering this, the registration of rTMs must also be moved up to the read-phase. After this two modifications we realized that in fact it is better to avoid the read-phase and start immediately with a extended commit phase that first needs to gather the participants.

The second modification is an eager notification mechanism. Currently, out transactional layer is meant for asynchronous access to the share state. When a peer write a new value for item, other users are not notified unless they read the item. In the case of TransDraw, other users needs to be notified not only of every modification on the value of items, but also on the intention of other users when they

lock items. To achieve this, the leader must broadcast its decision to the network once it get enough locks, and once the final decision is taken.

Note that eager locking and the notification mechanism are only needed on synchronous collaborative work. If the collaborative application relies on asynchronous collaboration it is enough with the Paxos protocol presented in the previous section. Scalaris [9] is an implementation of Wikipedia running on top a structured overlay network with Paxos transactional protocol. This shows that we could already add fault-tolerance and decentralization to TransDraw if the goal is work on asynchronous fashion. The suggested modifications are meant for achieving real-time collaborative work.

6. Conclusion and Future Work

We have seen that several synchronous collaborative applications are currently based on centralized synchronization. This strategy is efficient but not fault-tolerant because it strongly relies on the stability of the server. Some applications achieve fault-tolerant by replicating the state of the server, but this requires a more sophisticated infrastructure and it is still inherently centralized. Single point of control is a single point of failure.

We propose to implement these kind of applications on top of structured overlay networks with symmetric replication, and a transactional layer based on consensus. This strategy provides synchronization and fault-tolerance by decentralizing the control of the work flow. We present our approach by taking the TransDraw application and the Beernet peer-to-peer network.

Beernet as is, can help to decentralize asynchronous collaborative applications. In order to achieve the functionality of TransDraw, which is synchronous, eager locking and a notification mechanism needs to be added to the current transactional protocol.

We still need to study in detail the new transactional protocol, implement it and compare the performance with the centralized approach. We expect to have a small degradation in performance at the level of the transactional protocol due to replication cost, but with a huge gain in fault-tolerance. There is no degradation in performance for the user in case of no conflicts, because its changes are done optimistically, eliminating the problem of network latency.

7. Acknowledgements

This work has been mainly funded by the European Commission FP6 IST Project SELFMAN (Contract 034084), with support of the UsiXML project.

References

[1] 0x539 dev group. The gobby collaborative editor. http://gobby.0x539.de, 2009.

[2] Distoz group. Beernet - the relaxed peer-to-peer network. http://beernet.info.ucl.ac.be, 2009.

[3] Google. Google docs. http://docs.google.com, 2009.

[4] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[5] D. Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[6] 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.

[7] M. Moser and S. Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[8] B. Pagès. The bouml tool box. http://bouml.sourceforge.net, 2009.

[9] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. In A. Clemm, L. Z. Granville, and R. Stadler, editors, *DSOM*, volume 4785 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 2007.

[10] 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 Decentralized Transactional Collaborative Drawing *(demo)*

Decentralized Transactional Collaborative Drawing*

Boris Mejías, Jérémie Melchior, Yves Jaradin
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

1 Demonstrator

DeTransdraw is a decentralized collaborative vector-based graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

DeTransDraw is a redesign of TransDraw [1], a client-server application providing similar features. Due to its centralized architecture, TransDraw has a single point of failure and does not scale beyond the capacity of the server. DeTransDraw is built on top of a peer-to-peer network, Beernet [4], allowing users to join and leave the application at any time, without relaying on any central point of control. The decentralized architecture of DeTransDraw makes it more scalable and fault-tolerant. Other collaborative applications, either synchronous or asynchronous, can benefit from these properties by reusing the transactional layer over an equivalent peer-to-peer network.

The transactional service we use is based on an eager protocol that is an adaptation of Paxos consensus algorithm [3]. The peer-to-peer network we built uses the relaxed-ring topology [2].

During the demonstration we will built an ad-hoc peer-to-peer network that will be interfaced by three clients running on three different computers. The three clients will run the graphical interface of DeTransDraw, accessing the shared drawing area. Apart from simple drawing actions, conflict resolution will be tested by trying to modify the same graphical objects by more that one client. Fault-tolerance will be tested by killing some of the peers during the drawing actions. For the demonstrations we will need space and power to set up three laptops and a router.

*This research is funded by SELFMAN (contract number: 034084).

2 Innovations

- Replicated storage achieved by decentralized transaction over peer-to-peer networks providing distributed hash table (DHT), providing eager notifications to the participants of a collaborative application
- Prevention of performance degradation due to network latency. Users work on the application almost as if it was a local application.
- Self-management of storage achieved with symmetric replication over a structured overlay network.
- Self-healing of transactions participants. A transaction always terminate if the majority of the peers is alive during the execution. Fault-tolerance is guaranteed depending on the majority.

References

[1] Donatien Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[2] 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.

[3] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[4] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network - *http://beernet.info.ucl.ac.be*.

A.14 Beernet: A Relaxed-Ring for Self-Managing Decentralized Systems with Transactional Replicated Storage

Beernet: A Relaxed-Ring for Self-Managing Decentralized Systems with Transactional Replicated Storage

Boris Mejías

November 3, 2009

I am glad to be able to tell you in front of all my guests – despite the fact that their presence here is proof to the contrary – that your theory is intelligent and sound.

"The Master and Margarita" - Mikhail Bulgakov

Abstract

The increment of network bandwidth and computing power has definitely made an impact on distributed systems which are becoming larger, more complex and therefore, difficult to manage. Although classical client-server architecture provides a simple management scheme with centralized control of the whole system, it does not scale because the server becomes a point of congestion and a single point of failure. If the server fails, the whole system collapses.

The key to deal with the complexity of large-scale distributed systems is to make it decentralized and self-managing. Peer-to-peer networks, and specially in their form of structured overlays, offer a fully decentralized architecture which is self-organizing and self-healing. These properties are very important to build systems that are more complex than file-sharing, which is currently the most common use of peer-to-peer. Despite the nice design of many existing structured overlay networks, many of them present problems when they are implemented in real-case scenarios. The problems arise due to basic issues in distributed computing such as partial failure, imperfect failure detection and non-transitive connectivity.

This dissertation is about how to build self-managing decentralized systems. It presents a novel structured overlay network topology called Relaxed-ring, that provides cost-efficient ring maintenance without relying on transitive communication. The Relaxed-Ring is the base for Beernet, a pbeer-to-pbeer network prividing replicated and distributed transactional storage.

Fault-tolerant distributed hash tables requires some replication mechanism so as to deal with the failure of a peer without loosing data. Maintaining the replicas is not just costly but it is also difficult to guarantee their coherency. Beernet uses a transactional protocol based on Paxos consensus algorithm over symmetric replication that guarantees that at least the majority of the replicas is kept coherent. The transactional layer is adapted to provide synchronous and asynchronous collaboration between peers at the application level.

Acknowledgements

Considering that this document is still a draft, the acknowledgments to individuals is reserved for the final version. However, I would like to acknowledge the projects that have supported this research. The European project SELFMAN has provided not only monetary support, but large part of the ideas and results presented in this research has been developed in the context of SELFMAN. I also got support from other European projects EVERGROW and CoreGRID.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

All animals are equal

"Animal Farm" - George Orwell

There are many technological and social factors that make peer-to-peer systems a popular way of conceiving distributed systems nowadays. From the technological point of view, the increment of network bandwidth and computing power has definitely made an impact on distributed systems which are becoming larger, more complex and therefore, difficult to manage. Although classical client-server architecture provides a simple management scheme with centralized control of the whole system, it does not scale because the server becomes a point of congestion and a single point of failure. If the server fails, the whole system collapses.

The key to deal with the complexity of large-scale distributed systems is to make it decentralized and self-managing. Peer-to-peer networks, and specially in their form of structured overlays, offer a fully decentralized architecture which is self-organizing and self-healing. These properties are very important to build systems that are more complex than file-sharing, which is currently the most common use of peer-to-peer. Despite the nice design of many existing structured overlay networks, many of them present problems when they are implemented in real-case scenarios. The problems arise due to basic issues in distributed computing such as partial failure, imperfect failure detection and non-transitive connectivity.

Coming back to basics on distributed programming, let us review two definitions that target some key concepts concerning distribution. According to Tanenbaum and van Steen [TV01]:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system"

This definition suggests using *distribution transparency*, where all the effort of distributed programming is moved to the construction of a middleware that supports the distribution of the programming language entities. But network and computer failures cause unexpected errors to appear at higher abstraction levels, which breaks transparency and complicates programming.

The key issue in distributed programming is partial failure. It is what makes distributed programing different from concurrent programming. This is why we would like to quote Leslie Lamport and his definition of a distributed system:

> "A distributed system is one in which the failure of a computer you did not even know it existed can render your own computer unusable"

It does not matter how much transparency can be provided in distributed programming, it will always be broken by partial failure. This is not particularly bad, but it means that we need to take failures very seriously, understanding that perfect failure detection is unfeasible in Internet style networks, and that a failure does not mean only the crash of a process, but also a broken link of communication between two processes, implying non-transitive networks.

Because of failures, we cannot trust the stability of the whole system to a single node, or to a reduce set of nodes with some hierarchy. We need to build self-managing decentralized systems, where data storage needs to be replicated and load balanced across the network in order to provide fault tolerance.

This dissertation is about how to build self-managing decentralized systems. It presents a novel structured overlay network topology called Relaxed-ring, which deals with non-transitive connectivity, making it suitable for Internet applications. We also provide support for a transactional distributed hash table that allows programmers to write applications for synchronous and asynchronous collaboration between users. Beernet, our implementation of the Relaxed-Ring, is a pbeer-to-pbeer network providing replicated and distributed transactional storage.

1.1 Thesis and Contribution

Thesis: a solution to deal with the complexity of dynamic distributed systems is to build self-managing decentralized systems. They have to provide fault tolerance and deal with non-transitive connections. Global state has to be replicated and consistent.

Contributions:

- The design of a protocol for self-organizing peer-to-peer networks creating a network topology called relaxed-ring. The network is able to

deal with false suspicions in failure detection and with non-transitive networks such as the Internet, improving lookup consistency with respect to existing peer-to-peer networks. The relaxed-ring also provides self healing by triggering a failure recovery mechanism when the crash of a peer is detected.

- The relaxed-ring protocol is cost-efficient because it does not rely on periodic stabilization to repair the network when it is affected by churn. The relaxation introduces branches to the ring topology, but it keeps the routing algorithm competitive with $log(N)$ hops to reach any peer.

- We provide a self-adaptable routing topology that allows the relaxed-ring to take advantage of full connectivity in small networks, and logarithmic routing in large networks. The system can scale up and down making it suitable for many different applications independent of the size of the network.

- We present the algorithms of the relaxed-ring using feedback loops to analyse and validate its self-management properties. The feedback loops help us to understand how the system monitors itself, analyses the information, and triggers the needed action to modify the system.

- We study and validate the Paxos consensus algorithm for atomic transactions on a replicated DHT, and we compare it with the well known solution for distributed transactions called Two-phase commit.

- We adapt Paxos consensus algorithm to provide eager locking of the transaction participants, and we extend it with a notification layer to make other peers aware of the modifications. This new protocol allows us to design application where users can collaborate synchronously.

- As proof-of-concepts, we have implemented Beernet, the pbeer-to-pbeer network, a relaxed way of doing peer-to-peer. It is an implementation of the relaxed-ring where peers are organized as a set of distributed-transparent actors. These actors represents components with encapsulate state and that communicates only via message passing, avoiding share state concurrency. Beernet also takes advantage of the fault-stream model for failure handling improving its modularity and network transparency. These characteristics provide a better programming framework for self configuration of components.

- We have implemented and presented to the research community three different demonstrators to introduce the concepts of the relaxed-ring, atomic transactional DHT, and synchronous collaboration with eager transactions.

- We develop two applications on top of Beernet to exploit optimistic and pessimistic transactions, and the notification layer. These application provide a community-driven recommendation system, and a collaborative drawing tool. Two other applications designed and developed by third parties are also presented so as to emphasize the impact of the contribution of the relaxed-ring and its transactional layer.

1.2 Publications, Software and Award

The work presented in this dissertation is the result of incremental progress made through discussions and presentations in several workshops and doctoral symposia. The implementation of software as proof of concepts has contributed to polish algorithms and ideas about how decentralized systems should be designed. Several of the results presented here have also been published in conferences and a journal. In this section we present the most important publications that support this dissertation, together with references to software demonstrators that validates the implementation of the ideas. There is also an award to be mentioned that the author has receive for his presentation in a doctoral symposium.

- **Journal** *"The Relaxed-Ring: A fault-tolerant topology for structured overlay networks"*. Boris Mejías and Peter Van Roy. In Parallel Processing Letters, Vol. 18(3):411–432, World Scientific, September 2008. This publication presents most of the results that we will describe in detail in Chapter 3, presenting part of the evaluation measurements that will be presented in Chapter 5.

- **Conference** *"A Relaxed-Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks"*. Boris Mejías and Peter Van Roy. In Proceedings of XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8-9 November 2007, Iquique, Chile. This publication focus more on the analysis of the self-management behaviour of the Relaxed-Ring, using feedback loops as a mean to describe an analyse the algorithms for ring maintenance and failure recovery. This ideas are further discussed in Chater 4.

- **Conference** *"PALTA: Peer-to-peer AdaptabLe Topology for Ambient intelligence"*. Alfredo Cádiz, Boris Mejías, Jorge Vallejos, Kim Mens, Peter Van Roy, Wolfgang de Meuter. In Proceedings of XXVII IEEE International Conference of the Chilean Computer Science Society (SCCC'08). November 13-14, 2008, Punta Arenas, Chile. This paper complements Relaxed-Ring's algorithms with an efficient self-adaptable routing table. The algorithm is described in detail in Chapter 3, and its evaluation is included in Chapter 5.

- **Demonstrator in Conference** *"PEPINO: PEer-to-Peer network INspectOr"* (abstract for demonstrator). Donatien Grolaux, Boris Mejías, and Peter Van Roy. In Proceedings of P2P '07: Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing. September 2-7, 2007, Galway, Ireland. This software demonstrator has helped us to visualize and polish our implementation of the Relaxed-Ring, being closely related to Chapter 7 and the applications of Chapter 8.

- **Demonstrator in Conference** *"Visualizing Transactional Algorithms for DHT"* (abstract for demonstrator). Boris Mejías, Mikael Högqvist and Peter Van Roy. In Proceedings of P2P '08: Proceedings of the Eighth IEEE International Conference on Peer-to-Peer Computing. September 8-11, 2008, Aachen, Germany. This software demonstrator validates the design and implementation of the transactional layer for atomic commit on DHTs with symmetric replication. It validates the claims we discuss in Chapter 6.

- **Award** The author has won the *"Best Presentation Award"* in the Doctoral Symposium of the "XtreemOS Summer School", held at the Wadham College of the University of Oxford, Oxford, UK, on September 10, 2009. The presentation was entitled "Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT" [Mej09], and it summarized the contribution of this dissertation.

1.3 Roadmap

This dissertation is organized as follows. Chapter 2 makes a review of all three generations of peer-to-peer systems, being structutured overlay networks the most important focus of the analysis. The systems we reviewed are not only studied from the point of view of their overlay graph, but also from their self-managing properties. We also review distributed storage and the connection of peer-to-peer with Grid and Cloud Computing. Chapter 3 presents the protocols and algorithms of the Relaxed-Ring, being an important part of the contribution of this dissertation. The Relaxed-Ring is also studied using feedback-loops in Chapter 4 so as to understand its self-managing properties from a architectural and software design point of view. Evaluation of the Relaxed-Ring, specially in comparison with other overlay graphs, is done experimentally using a concurrent multi-agent simulator. The results of such evaluation are presented in Chapter 5.

Once we have presented the Relaxed-Ring, the dissertation continues with the study of distributed storage in Chapter 6. We analyse Two-Phase commit, Paxos consensus algorithm, and we describe our contribution with Eager Paxos and the notification layer. Chapter 7 describes the design deci-

sions and implementation details of Beernet, which implements the Relaxed-Ring and its layer for transactional distributed hash tables using symmetric replication. Before the concluding in Chapter 9, we present a set of applications designed and developed using Beernet and the ideas of the Relaxed-Ring. Some of the applications are developed by the authors, and some of them are contributions of third parties, emphasizing the impact of this dissertation.

Chapter 2

The Road to Peer-to-Peer Networks

> All animals are equal, but some are more equal than others
>
> *"Animal Farm"* - George Orwell

The goal of distributed computing is to achieve the collaboration of a set of different autonomous processes. A process is an abstraction of an entity that can perform computations. This entity can be a computer, a processor in a computer, or a thread of execution in a processor. We will use *nodes* or *peers* to also mean a process. The most basic problem that has to be addressed is to establish the connection between two processes and to provide programming language abstraction to allow programmers to perform distributed operations. As more processes come into communication, enlarging the network, it is necessary to correctly route messages between processes that are not directly connected. And as the network grows larger, it is necessary to design system architectures that can ease the collaboration between processes. Even though the first issues we mentioned are not completely solved, the existing solutions are good enough to let us focus on the architecture of the system.

We are interested in designing and building overlay networks to organize processes that are already able to communicate between each other and to route messages through an underlaying network. Therefore, this chapter is dedicate to analyze existing overlay networks to contextualize the contribution of this dissertation. Even though this work is developed on a high level of abstraction, we still consider many of the basic principles of distributed computing, such as *latency* or *partial failure*. These principles go across all level of abstractions on distributing computing, and not taking them into account would be like an architect discarding physical rules that would invalidate its design of a building. We will mention some issues concerning routing messages on the underlaying network during this chapter, and we

will discuss more about language abstractions for programming languages when we describe the implementation of Beernet in Chapter 7.

In the introduction we motivated the use of peer-to-peer networks for building dynamic distributed systems, because of their decentralized, fault-tolerant and self-organizing structure. We also claimed that increasing self management in such systems is the only way of dealing with their high complexity. The self-management properties of peer-to-peer networks are so intrinsic to them that we will start this chapter by briefly introducing some concepts of self management, and then we will use them to analyze the related work.

2.1 Self Management

The complexity of almost any system is proportional to its size. This rule also holds for distributed system. As systems grow larger, they become more and more difficult to manage. Therefore, increasing system's self management appears as a natural way of dealing with high level complexity. By self management, we mean the ability of a system to modify itself to handle changes in its internal state or its environment without human intervention but according to high-level management policies. This means that human intervention is lifted up to the level where policies are defined.

Typical self-management operations are: tune performance, reconfigure, replicate data, detect failures and recover from them, detect intrusion and attacks, add or remove parts of the system, which can be components within a process, or a whole peer, and others. Each of those actions or a combination of them can be identified as *self-configuration, self-organization, self-healing, self-tunning, self-protection* and *self-optimization*, often called *self-* properties* in literature. We will use this properties to analyze the related work and the contribution of this work, but self-protection is not in the scope of this dissertation.

One of the key operations that a system must perform to achieve self-managing behaviour is to monitor itself and its environment. Once relevant information is collected, it can take decision over which action to trigger to achieve its goal. Once the action is triggered, the system needs to monitor again to observe the effect of its action, developing a constant feedback loop. We will review more about feedback loops in Chapter 4. In peer-to-peer systems, monitoring is distributed and based only on the local knowledge that every peer has. Peers monitor each other and trigger actions in other peers. Global state can be infered but always as an approximation, because there is no central point of control that observes the whole system at once. Self-managing behaviour must be observed as a property of the whole network, and not as an isolated property of a single peer.

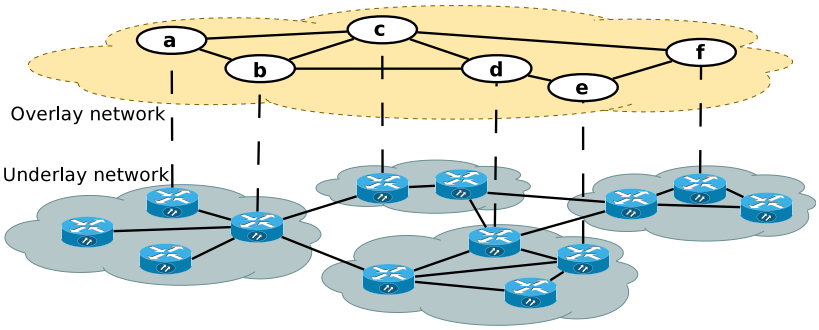


Figure 2.1: Overlay network.

2.2 Overlay Networks

A computer network, is a group of interconnected processes able to route messages between them. Internet is a group of interconnected networks, routing messages between processes independently of the network where they belong. An *overlay network* is a network built on top of another network or set of networks. For instance, a group of processes using the Internet to route their message is said to be an overlay network, where the Internet is the *underlay network*. Actually, the Internet itself can be seen as an overlay network running on top of the group of local area networks.

Figure 2.1 depicts the architecture we are describing. An important issue to discuss here is the analysis of the routing of messages. We can observe that nodes **f** and **e** are directly connected in the overlay network. Therefore, a message sent from node **f** to **e** is considered to be sent in one hop. However, if we look at the topology of the underlaying network, we observe that the message would take at least three hops, and it go through the node identified as **d**, which is not even connected with **f** in the overlay. This different in the amount of hops is understandable if we consider that the overlay network complete abstract the underlaying network. The same overlay network depicted in Figure 2.1 could have been deployed over a different underlaying network where nodes **f** and **e** are really directly connected, or completely far away. Since there is no direct correlation between overlay and underlay in the amount of hops needed to route a message, we will consider these two analysis as independent. This does not mean that the design of an overlay network can take the underlaying network into account in order to optimize routing. In this dissertation, when we discuss the amount of hops to route a message we will mean at the level of the overlay network, unless it is explicitly stated.

2.2.1 Client-Server

Client-Server is the one of the most basic and popular architecture to build distributed systems. It is very simple and it allows the designer of the server to have control over the system, because all messages have the server as participant. It can also be seen as an centralized overlay network with a *star* topology. Unfortunately, it relies too much on the server which becomes a point of congestion and a single point of failure. The system relies entirely on the server. If the server crashes, there is no application. The size of the application also relies on how powerful the server is to handle the connection of all the clients. Therefore, it does not scale very much. If there is any self-management property that we want to analyze here, it would be entirely focused on the server, and we would remain with the problem that if the server is gone all self-* properties will be gone too.

Currently, companies that base their business model on the client-server architecture have extended it to run more code on client's machine, allow some communication directly between clients, and more fundamentally, replicate their servers in order to scale and provide more fault tolerance. If we focus the self-management analysis on the group of servers, then we would not be studying client-server architecture anymore, because the group of servers would actually form a different network. Yet, if the access to the group of servers is broken, there is no application. To achieve more fault tolerance it is necessary to decentralize the system. Decentralization will also increase scalability but at the cost that there will be no central point of control. Increasing self management will allow the system to control itself.

2.2.2 Peer-to-Peer First Generation

Napster [Nap99] is the first peer-to-peer system to be widely known. It was a file-sharing service that allowed users to exchange files directly, without sending them though the server. It is said to belong to the first generation of peer-to-peer networks where we also find AudioGalaxy [Aud01] and OpenNap [Ope01]. This generation was not entirely peer-to-peer. It was based on a mixed architecture which still relied on a server to work. Peers connected to a server in order to run queries over media files. The server replied with the addresses of the peers storing the requested file so that peers could connect directly. If the server failed, the exchange of files could continue, but it was not possible to run new queries. Therefore, the application would stop to work as soon as all current downloads where completed. It was not possible to route messages to other peers through the currently connected peers, and this is why it is not considered to be entirely peer-to-peer. Only the exchange of files was done peer-to-peer . OpenNap, an open source derivate work from Napster, allowed communication between different servers improving robustness of the system, but the network remained centralized on the group of

servers handling the queries. Because large part of the exchanged content was copyrighted, Napster ceased its operations in 2001 due to legal issues. The service was easily shut by just stopping the servers.

2.2.3 Peer-to-Peer Second Generation

> The more it stays the same, the less it changes!
>
> *"The Majesty of Rock"* - Spiñal Tap

The second generation of peer-to-peer networks is considered to be the first real peer-to-peer system, because it is fully decentralized, it does not rely on any server, and it is able to route messages using peers on the overlay network independently of the underlay network. This generation is mainly represented by Gnutella [Gnu03] and Freenet [Fre03], and it is also developed having file-sharing as goal. It was actually a solution to Napster shut down, because there was no server to stop. These systems are also known as *unstructured overlay networks* because peers are randomly connected without any particularly defined structure. As we have discussed already, nowadays almost any machine can behave as a client and a server. Therefore, every peer can trigger queries as a client, and handle queries from other peers, playing the role of a server.

The algorithm to route messages in such unstructured network is called *flooding*. It is very simple but highly bandwidth consuming. It works as follows: the peer that triggers the query sends it to all its neighbours with a time to live (TTL) value. The TTL can be expressed in seconds or hops. We will use hops for our example. The receiver of the query determines if it is the first time that has seen it and if the TTL is greater than 0. If so, it transmit the query to all its neighbours except for the sender, with a decremented TTL. If the peer can resolve the query, it answers back following the path to the original sender. In Figure 2.2 we can see the flooding algorithm initiated by peer **h** with a TTL of 2. The extra circles around the peers on the figure represents the amount of hops that the message needed to reach the peer. All coloured peers participated in the routing algorithm. Peers with a darker coloured means that the peer receive the message more than once. Let us imagine that the query triggered by peer **h** can be answered by node **m**. The query is first sent from **h** to nodes **f**, **g**, **j** and **k**. Every peer will send it to its neighbours, so **m** receives the message from **k**, and the answer travels back following **m**→**k**→**h**.

There are several issues that make this algorithm less scalable and not suitable for the kind of systems we want to build. If we observe peer **f** on this example, first, it receives the query from peer **h** and then from peer **j**. A similar situation occurs with the other nodes on the first level of flooding. On the second level, nodes **i** and **n** receive the message twice too. In conclusion,

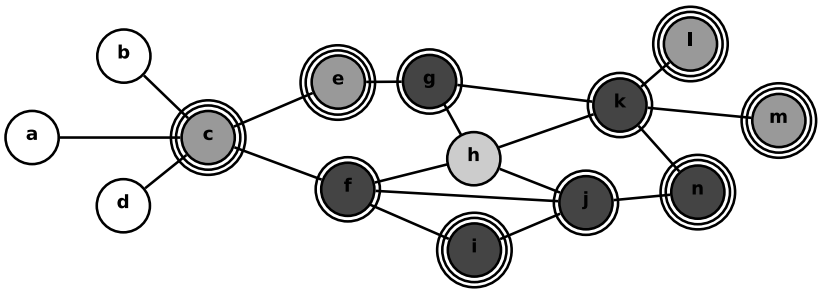


Figure 2.2: Flooding routing in a unstructured overlay network.

many nodes process the query unnecessarily more than once, inefficiently using their resources.

A second issue is the amount of messages being sent. In this example, the messages is sent 16 times without counting the responses. In the ideal case, it was only necessary to follow the path **h**→**k**→**m**. To do such routing, peers would need a routing table with information about their neighbours. However, the absent of routing tables for the sake of simplicity is considered to be one of the advantages of unstructured overlay networks.

Determine a correct TTL is also an issue. If the query could have been resolved immediately on the first level, all the messages sent to the second level and further would have been unnecessary. If the TTL on this example would have been set to 3, the whole network would have been flooded to resolve the query. However, if the query could have been only resolved by **a**, **b** or **d**, a TTL of 2 would not have been sufficient to find the answer. This is one of the reasons why unpopular items are more difficult to find in file-sharing services based on flooding routing, even though the items are stored somewhere in the network.

Another issue that influences the success of resolving a query is the place of the originator. Nodes closer to the center of the network will reach more nodes that nodes living at the border of the network. For instance, in Figure 2.2, node **h** floods the whole network in three hops, but nodes **l** and **m** would need to use a value TTL of 5 to reach **a**, **b** or **d**. In a very large network, a TTL of 6 seems to be reasonable following the *six degree of separation* theory of human connected networks.Â Using a formula from [AH02] to count the amount of messages sent in a query (with the responses), with an average C connections per peer, and using TTL of 6, we obtain:

$$2 * \sum_{i=0}^{TTL} C * (C-1)^i = 54610 \tag{2.1}$$

Despite all these issues, unstructured overlay networks are still very popular to run file-sharing services because bandwidth consumption is mainly considered a problem for ISP providers and not for the users, and because

popular items can be found in a reasonable amount of hops, again, following the rule of *six degree of separation*. Another reason is that items stored in file-sharing systems do not update their values. If many peers store the same item, it is necessary to find at least one of the peers. There is no such thing as the *latest value* of the item, so there is no issue with respect to consistency. Any replica of an item is a valid one as long as the value stays the same.

Flooding routing works fine for networks with a tree topology, because it avoids that peers receives messages more than once, but it is very costly for unstructured overlay networks, being inefficient in bandwidth and processing-power usage. Another problem is that there is no guarantee of reachability or consistency, properties that we consider important to build decentralized systems that constantly update the values of the stored items. Even though Gnutella can keep large amount of peers connected, it does not mean that scalable services can be built on top of it because of the problems on efficiency, reachability and consistency [Mar02, RFI02]. Also, according to [DGM02], it is not difficult to perform a query-flood DoS attack in Gnutella-like networks, but their success depend on the topology of the network and the place of the originator of the attack, which is related to the reachability issue we already discussed.

Freenet also uses flooding routing but it presents some improvements with respect to Gnutella. The main difference is that the queries can be done with anonymity in Freenet. Since these systems were conceived as file-sharing services, the motivation for providing anonymity is basically legal, so no user can be sued. There is a degradation in performance because everything is sent encrypted. Freenet also keeps some information with respect to locality on the routing tables of the peers so as to improve routing speed. But, since it is flooding based, it is still very expensive.

With respect to self-* properties, we observe a basic self-organization mechanism despite the fact of not having any structured topology. There is no global or manual mechanism to organize the peers. Peers joining the network are just connected to the peers it gets introduced by its entry point. When nodes disconnect from the network, the other nodes simply stop forwarding queries to them, and therefore, there is no need for a self-healing strategy. Routing protocols in Gnutella and Freenet are under continuous improvement by their communities, but we will not refer to them because they go beyond the simplicity of the basic unstructured network, and they do not solve the more fundamental problems already discussed. It is possible to provide some self-tunning of the TTL value, and some self-optimization in the flooding paths by adding more information to the routing tables of the peers, but we will study better choices on the structured overlay networks in the next section.

2.3 Structured Overlay Networks

The third generation, also known as *structured overlay networks* (SONs), is the result of academia's interest in peer-to-peer networks. It clearly aims to solve the problems of unstructured networks by providing efficient routing, guaranteeing reachability and consistent retrieval of information. Adding structure makes possible to achieve these improvements, but it also creates new challenges such as dealing with disconnections of peers and non-transitive links. SONs typically provide a distributed hash table (DHT) where every peer is responsible for a part of the hash table. There are two basic operations that every DHT must provide: `put(key, value)` and `get(key)`. The `put` operation stores the value associated with its key such that every peer can retrieve it with the `get` operator. If another value was already stored under the same key, the value is overwritten. We define now some of the concepts we will use in this section and in the following chapters. These terms are also common to other surveys found in literature [AH02, BL03, GGG+03, EAH05, LCP+05, AAG+05].

- **Item.** The *key/value* pair stored in the hash table.
- **Identifier space.** The key of an item is always mapped into a hash key, which is the *identifier* of the item (abbreviated as id). The range of possible values of ids is the *identifier space*. Peers in the network are also associated with an id. Due to that, the identifier space is also named *address space*
- **Lookup.** It is the operation performed by any peer to find the responsible peer of a given key.
- **Join.** A new peer getting into the network.
- **Leave.** A peer disconnects from the network either voluntarily (gentle leave) or because of a *failure* (also named *crash*).
- **Churn.** Measurement of peers joining and leaving the network during a given amount of time.
- **Iterative routing.** The originator of a lookup sends its message to its best contact on the overlay, with respect to the searched key. The contact answers back with its best contact, so the originator iterates until reaching destination. This seems to be inefficient, but if a node in the path fails, the originator nodes exactly where to recover from. See Figure 2.3(a).
- **Recursive routing.** The originator of a lookup sends its message to its best contact on the overlay, with respect to the searched key. The

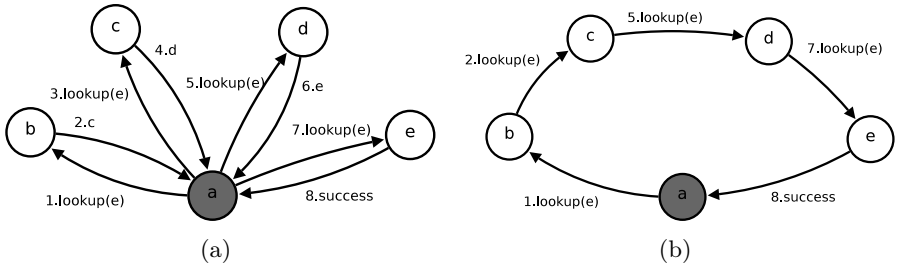


Figure 2.3: Example of (a) iterative and (b) recursive routing.

contact forwards the lookup request to its best contact, and so continues the search. When destination is reached, the peer answer back to the originator. It is more efficient than iterative routing, but in case of a failure, the whole process needs to be restarted. See Figure 2.3(b).

Every system assumes that nodes willing to join the network have a reference to at least one peer in the system, which we call the first contact. We will start by discussing ring-based networks because it is where our contribution is made. We will deeply analyze Chord because it influences many other systems, which are basically variations with improvements to Chord.

2.3.1 Chord

Chord [SMK+01, DBK+01] is one of the most known and referenced SON. In Chord, peers are self-organized forming a ring with a circular address space of size N. Hash keys are integers from 0 to $N-1$. The ring can be seen as a double-linked list with every peer having two basic pointers: *predecessor* and *successor* (abbreviated as *pred* and *succ*). Figure 2.4(a) depicts an example of a Chord ring. Only pointers of peer identified with id q are drawn on the figure but every peer holds equivalent pointers. Peers p and s corresponds to *pred* and *succ* respectively. This means that $p < q < s$, where '$<$' is defined on the circular address space following the ring clockwise.

DHT The ring provides a DHT where every peer is responsible for the storage of a set of keys determined by its own id and its predecessor. In the case of q, the peer is responsible for the range $(p, q]$, (i.e., excluding pred's id and including its own). If the ring is perfectly linked, there is no overlapping of peers' responsibilities, and therefore, every lookup operation gives consistent results.

Fingers To provide efficient routing, Chord uses a set of extra pointers called *fingers* or *long references*. They are chosen dividing the address space in halves. The farther finger of q is the responsible of key $(q + N/2)\ mod\ N$.

In our example in Figure 2.4(a), we consider $q = 0$ to label finger keys, and therefore, the ideal farther finger key is $N/2$. In the figure there is no peer holding exactly that key, but peer k is currently the responsible. Closer fingers are chosen using the same formula but dividing N by powers of 2. The fingers, together with pointers to pred and succ, form the routing table of a peer. Ideally, every peer holds references to $log_2 N$ fingers.

Lookup When any peer receives a lookup request for a given key, it first determines if the key belongs to the range between its own id and its successor. If that is the case, it answers the lookup query giving its successor id as answer. If it is not, it forwards the lookup to the closest preceding finger. It never uses the predecessor to route to avoid cycles. The routing mechanism is therfore recursive. In our example of Figure 2.4(a), if `lookup(m)` arrives to peer q, q forwards it to peer k. If it is `lookup(b')` with $b < b' < c$, then q forwards the message to b, being the closest preceding finger of b'. The, b answers that c is the responsible, because $c' \in (b, c]$. The fact that b answers that its successor c is the responsible of c' will be the source of inconsistencies under special cases of churn and connectivity problems, as it is described in [Gho06, MV08, SMS+08]. This occurs basically because b could not be aware of new node between c and the key c'. We will come back to this issue on Chapter 3.

Fingers fragments the address space into halves, therefore, every forwarding of a lookup request shortens the distance to approximately the half of it. Considering that the address space is discrete, the routing of the lookup converges to the responsible of the key in $O(log_2 N)$ hops. This routing cost is very scalable because if the network doubles its size, the routing takes in average only one extra hop.

Churn Figure 2.4(b) shows three different events producing churn: peer j joins as k's predecessor, peer b leaves voluntarily the network, and peer m crashes. In the case of the join, k accepts j only if j belongs to k's range of responsibility. Since $N/2 > j > q$, peer j becomes the new responsible of $N/2$, and therefore, it is a more suitable finger for q. This value needs to be updated somehow. A similar situation occurs when b leaves, because c becomes the new responsible of $N/4$. The difference here is that now q has temporary no finger for that value until it knows about c. The crash of m does not affect q's routing table, but it surely affect other peers' routing table, and the responsibility of m's successor.

Join When new peers want to join the network, they have to do it as predecessor of the responsible of its own key. Looking at the example in Figure 2.4(b), we observe that peer j joins as predecessor of peer k. To know where to join the ring, j has to previously request a `lookup(j)` to

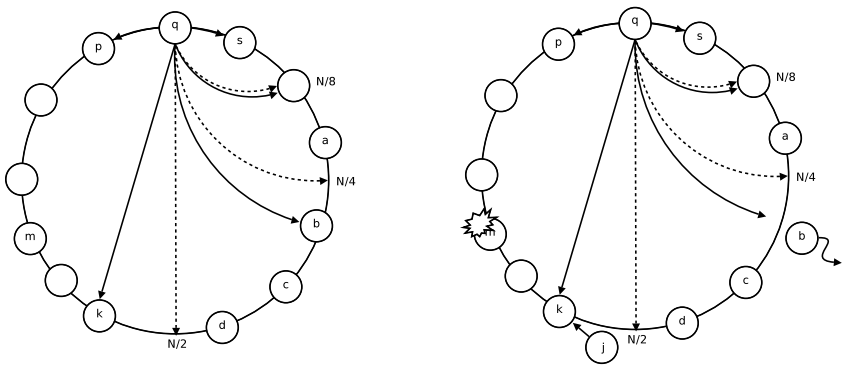


Figure 2.4: Example of (a) Chord ring and (b) some events causing churn.

its first contact, which can be any peer in the network. Since the answer to the lookup is k, peer j set its succ pointer to k and notifies it. Then, k determines that $j \in (d, k]$ and update its pred pointer to j. Half of the process of joining is done here. It is continued by periodic stabilization, which we describe now.

Periodic stabilization We can divide a peer's routing table into two groups: fingers, which are used for efficient routing, and pred/succ, which are needed for correctness. These references become invalid after some time due to churn. Therefore, it is necessary that every peer periodically checks the validity of the references.

A peer periodically asks its succ for the value of succ's pred. If it is the same as itself, there is nothing to change to do. If it is a new one, it is probably a better successor, or something when wrong and there is an inconsistency in the responsibilities of the DHT. Coming back to our join example in Figure 2.4(b), when it is time for d to run periodic stabilization, it asks k the value of its predecessor. Peer k answers j. Peer d realizes that $j \in (d, k]$, and then, d changes its succ pointer to j, fixing the ring. Then, d notifies j about itself becoming j's pred. That is how j gets to know d and the joining of j is completed. This mechanism relies entirely on *network transitivity* (i.e., if d can talk to k and k can talk to $j \Rightarrow d$ can talk to j). This property is often assumed as guaranteed, but it does not hold all the time, being source of errors in real implementations [FLRS05]. Another problem with this join algorithm based on periodic stabilization is that two peers joining simultaneously in the same range of keys will introduce lookup inconsistency even if connectivity is perfect, as analyzed in [Gho06]. We will discuss more about these two issues later on this section and on Chapter 3.

To check validity of fingers, a peer asks to every finger its pred value.

If pred is the new responsible for the ideal finger key, the finger pointer is updated. In Figure 2.4(b), q asks k for its pred. Peer k answers j, where $j \in (N/2, k]$, and then, j is a better finger because it is the responsible for key $N/2$. Note that this mechanism also relies on transitive connectivity between q, k and j. If the finger is found to have left the ring, as in the case of b leaving the network, a new lookup for the key $N/4$ must be performed.

Successors list Every peer holds a list of peers that follows its successor clockwise, which we call successors list. The size of this list is $log_2 N$, as in the finger table. When the current successor leaves the network, either voluntarily or due to a crash, the peer takes the closest peer is the successors list as its successor candidate, fixing the ring. It is possible that the successors list is not accurate due to churn, and some peers will be missing, but this problem is corrected by periodic stabilization.

Network partitions Chord can survive a network partition as long as every peer can find a valid successor candidate in its successor list. This means that no more than $log_2 N - 1$ consecutive peers have to reside on the same partition. Even when the ring survives the partition, it is not possible to provide consistency and availability at the same time. This is not a particular problem of Chord but of every network following Brewer's conjecture on partition-tolerant web services, formalized and proven in [GL02]. Even though Chord and other ring-based systems can survive network partitioning, none of these systems addressed correctly the problem of merging the rings when the partition is gone. Recently, a nice gossip-based solution was presented in [SGH07, SGH08], being general enough to apply it to many ring-based networks.

Self management Note that no central entity organizes peers' position in the ring. Every joining peer finds autonomously its successor, and every peer runs periodic stabilization independently. We identify this behaviour as *self-organization*, and it is essential to almost every SON. Periodic stabilization also reconfigures the finger table to provide efficient routing on the network. Considering fingers update only from the point of view of a single peer, we identify this behaviour as *self-configuration*. If we consider the global result, we observe that the network route messages more efficiently, therefore, we identify this behaviour as a basic *self-optimization*. The resilient information of the successors list, combined with periodic stabilization, can be clearly identify as *self-healing*.

It is important to remark that these self-management behaviour are intrinsic to Chord, as they are too almost every decentralized peer-to-peer network. Without these properties the system basically does not exist. This contradicts some views on self management that attempt to analyze de-

centralized systems as autonomic systems that have evolved from manually controlled systems. For instance, some methodologies [Mil05, LML05, ST05] define their model to evaluate the system *with* and *without* autonomic behaviour. They define the maturity of the system by the ability of turning on and off each autonomic behaviour. As we said, Chord would not work correctly without periodic stabilization, and one could not *turn off* self-organization of the ring. According to those methodologies, that would mean that the system is not mature enough, which actually does not reflect what a decentralized system is.

It would be more interesting to discuss how hidden a self-* property can be. For instance, the lookup procedure is orthogonal to the protocols that maintain the ring and the routing table. Therefore, the four self-* properties we already mentioned are hidden to the lookup. When the message arrives to the peer and it needs to be forwarded, the mechanism does not need to know how the pointers were defined. It just takes the most convenient finger. We realize that even when lookup is a low-level primitive in SONs, it is at a higher level with respect to routing table and ring maintenance.

Observations One of the advantages of Chord is that their protocols for maintaining the ring are quite simple and lock free. But, they rely on periodic stabilization to fix lookup inconsistencies, and on transitive connectivity to complete the protocols. It is shown in [KA08] that exist a given value for the ratio of churn with respect to the frequency of periodic stabilization, where the longest finger of any peer is always dead at the moment of performing a lookup. This means that routing efficiency is highly degraded preventing the correct execution of any application built on top of the network. To solve this problem, periodic stabilization has to be triggered more often. We already mentioned that the join algorithm is not lookup-inconsistency free. Since periodic stabilization fix those inconsistencies, making it run more often also contributes to a better ring maintenance. The big disadvantage is that periodic stabilization is very costly, making an inefficient use of the bandwidth.

The circular address space, as it is used by Chord and many other ring-based systems, relies on the uniformity of the identifiers of the peers. If the keys present a skewed distribution, many fingers will point to the same peer creating points of congestion. Another problem that can appear, even if peers' identifiers are uniformly distributed, is that the keys of the stored items have a skewed distribution. For instance, consider the words on a dictionary. If every peer has to store the words of a given letter, some peers will have to store a lot more information than others, unbalancing the network. This issue is address in Oscar [GDA06].

Chord was designed to scale to very large networks and it does it well, providing logarithmic routing cost. Unfortunately, if we would like to use

Figure 2.5: Partition of the address space in DKS.

Chord to create very small systems, the topology and routing tables would be too sophisticated and less efficient than a full mesh, which is completely reasonable to use in very small networks. Chord scales up very well, but it is not its goal to scale down.

There are several Chord implementations and services built upon it, among which we find the main implementation [Cho04] and *i*3 [SAZ+02], a DNS service [CMM02] and a cooperative file-sharing service [DKK+01].

2.3.2 DKS

Overlay DKS [AEABH03] is also a ring-based peer-to-peer network with a circular address space as Chord. Its design presents improvements in routing complexity, cost maintenance of the ring, and replication of the data. DKS stands for Distributed k-ary Search. As it names suggests, the address space is divided into k intervals rather than 2 as in Chord. An example of the division strategy can be seen in Figure 2.5. In the example, node n divides the space into $k = 4$ intervals, having a finger to each peer at the beginning of every interval. The closest interval is again divided into k subintervals with the correspondent fingers. The figure shows a new division of the closest interval, even though the arrows of the fingers are not drawn. The division continues until an interval is not dividable by k any more. The lookup process works exactly as in Chord, forwarding the message to the closest preceding finger. Since there are always k intervals, the lookup process converge in $O(log_k N)$ hops, which is better than Chord. The larger the value of k, the smaller the amount of hops, but the larger the size of the routing table, which is a disadvantage because its maintenance becomes more costly. We can say that DKS generalizes Chord, where Chord becomes an instance of DKS where $k = 2$.

Correction-on-change and correction-on-use Another fundamental improvement with respect to Chord is that DKS does not rely on periodic stabilization. This can be achieved by having atomic join/leave operations, and more interestingly, it introduces the principles of *correction-on-use* and *correction-on-change*. Correction-on-use means that every time messages are routed, information is piggy backed to correct fingers. The more the network is used, the more accurate the routing tables become. Correction-on-change is more concerned with the detection of nodes joining or leaving the network.

Every time such an event is detected, the correction of pointers is triggered immediately instead of waiting for the next round on periodic stabilization.

Atomic join/leave To solve the problem of correcting the succ and pred pointers, DKS uses an atomic join/leave algorithm [Gho06] based on distributed locks. Providing atomic join/leave operations does not only reduce the need for periodic stabilization, but it also reduces lookup inconsistencies, which is a more important contribution. Previous attempts to provide atomic join/leave operations [LMP04, LMP06] failed to provide safety and liveness properties. The main problem was their use of three locks: succ, pred, and the joining/leaving peer. In DKS instead, only two locks are needed. To join or leave, every peer needs to get its own lock and the lock of its successor. In the case of joining this is simpler, because nobody knows about the joining peer except for itself. Then, its join operation is guaranteed as soon as it gets the lock of its successor. Leaving is relatively more difficult, because a peer cannot depart from the network if its predecessor or successor is leaving as well, and getting their locks first. We consider this an important drawback.

The algorithm guarantees safety and liveness properties. It is proven to be free of deadlocks, livelocks and starvation. However, all the proofs are given in a failure-free scenario which is unrealistic for a peer-to-peer network. If a peer crashes holding the lock of its successor, it will prevent its successor from answering lookup requests, increasing unavailability. If the peer is falsely suspected of having failed, errors can be introduced by having duplications of locks. The algorithm is also broken in presence of non-transitive connections, because peers will not be able to acquire the relevant locks in order to perform a join or a leave. Distributed locks must not be used unless it is unavoidable, as we will see when we discuss consistent replication of storage.

Storage With respect to storage, DKS also introduces an interesting strategy to locate replicas symmetrically on the network [GOH04], instead of placing them on the successor list as Chord does. Symmetric replication contributes better to load-balancing and makes recovery on failure more efficient. We will discuss more about symmetric replication in Chapter 6.

Self management DKS presents very similar self-management properties to Chord but their mechanism to achieve them differ. Both rings are self-organized, self-optimized and self-healing, and peers' routing tables are self-configured. Chord achieve many of these properties through periodic stabilization and some through immediate reaction on join events and failure detection. DKS achieve self-organization through atomic join/leave algorithms, self-optimization and self-configuration through correction-on-use and correction-on-change, and self-healing through correction-on-change.

2.3.3 P2PS and P2PKit

Overlay P2PS [MCV05] is also a ring-based peer-to-peer system with logarithmic routing. It is the predecessor of Beernet [Pro09] and the relaxed-ring. P2PS uses the Tango protocol [CM04] for building the finger table and routing messages across the network. Tango is very similar to Chord and DKS, but it takes into account the redundancies found in the finger tables of different nodes. One can observe that there are different paths from peer i to j with the same amount of hops, and any of those paths could be taken to resolve a lookup operation. Tango exploits these redundancies providing a more scalable and efficient solution. As drawback, it needs to take into account the information of other nodes, and therefore, it is expensive to maintain the routing tables up to date. To compensate this cost, by exploiting redundancy, routing tables in Tango are smaller than those of DKS and Chord. The average cost for routing messages is also $O(log_k N)$ hops, but it is claimed to be 25% faster that Chord in worse cases.

Ring maintenance Taking into account the fact that two simultaneous join with the same successor candidate created lookup inconsistencies in Chord, P2PS designed its own join algorithm. Similarly to DKS, P2PS does not use periodic stabilization to fix succ and pred pointers. It uses correction-on-change instead. Contrary to DKS, P2PS does not use distributed locks to guarantee atomic join/leave operations, which is in fact an advantage. An important contribution to fault tolerance is that graceful leaves where not considered in the design of the protocol. They are treated as failures. The reason is that if a node fails while performing a graceful-leave protocol, a failure-recovery strategy must be design for that particular case, adding more complexity to the ring maintenance. If leave due to a failure is already handle by correction-on-change, an algorithm for graceful leaves is not needed.

The join algorithm of P2PS is claimed to be atomic, and in fact, it does not introduce lookup inconsistencies even if two nodes join the ring simultaneously within the responsibility range of a given peer. Even so, the algorithm relies on network transitive to complete. Although the algorithm is atomic for two simultaneous join, we proved in [MJV06] that the algorithm did not work in particular cases of three and more simultaneous join, and that the inconsistency persist until new the compromised peers left the network. Unfortunately, we wrongly conclude in that technical report that a lock-based algorithm, such as the one of DKS, was needed so as to guarantee atomicity for join and leaves. Later, taking inspiration from P2PS's lock-free algorithm we developed the relaxed-ring, which is presented in Chapter 3.

Architecture The implementation architecture of P2PS is designed in layers going bottom-up from the communication layer to more general services. Figure 2.6 is based on the architecture presented in [MCV05], and it is com-

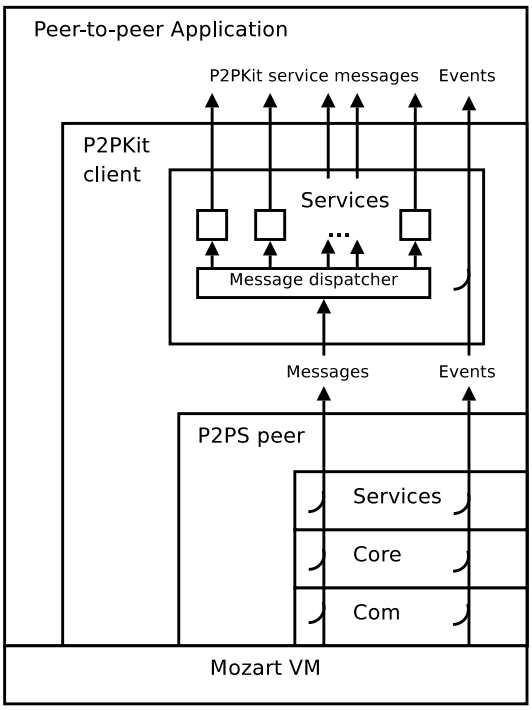


Figure 2.6: P2PS/P2PKit architecture.

plemented with P2PKit's architecture, which we will soon describe. P2PS is implemented in Mozart [Moz08], which is an implementation of the Oz language [MMR95, Smo95]. The Mozart virtual machine is at the bottom of the architecture and it is accessed no only by P2PS, but also by P2PKit and the peer-to-peer application. Messages are divided into two groups: *events* and *messages*. Events are those corresponding to the maintenance and functioning of the network, such as joins, leaves, acknowledgements, etc. Messages are those sent by the application and propagated to other peers through the network. Both sets of messages goes across the three layers if P2PS, where each layer triggers new messages and new events. At the bottom of P2PS we find the `Com` layer, which is in charge of providing a reliable communication channel between peers. The `Core` layer is in charge of the ring maintenance, handling join and leave events, and keeping the routing table up to date. Functionalities such as general message sending, multicast, broadcast, and others, and provide in the `Services` layer.

P2PKit The API of P2PS was considered to be too basic so as to develop peer-to-peer applications in an easy way. The design of P2PKit [Gly05] aims to simplify the task of developers providing high-level abstractions to deploy peer-to-peer services. Although P2PKit is independent of the underlaying peer-to-peer system, we present it together with P2PS because of their tight

implementations [Pro08, Gly07]. Continuing with the analysis of the architecture of Figure 2.6, we observe that P2PKit creates a client separated from the peer, giving the possibility of using multiple clients for the same peer. Network events are still triggered to the application level, and new events are added by P2PKit. The crucial part of P2PKit's approach is the use of the message stream. It creates many different services as channels. These services are provided by the application working with a publish/subscribe mechanism. There is a message dispatcher in charge of filtering all messages received by the peer, putting them into the correspondent service. If the application decides not to listen to a service any more, those messages simple won't reach the application layer as in P2PS.

Storage One of the earliest work on generic decentralized transactional protocols for replicated ring-based SONs was done on P2PS [MCGV05]. The protocol is based on two-phase locking and provides fault tolerance for the peers holding replicas. Partial fault-tolerance is provided for the transaction manager, but if the manager die once it has taken the decision, the protocol run into inconsistencies. No implementation and no API was provided for this protocol. We will discuss again this protocol in Chapter 6.

Self management P2PS and DKS share many means to achieve self-management properties. Both systems rely on correction-on-change and correction-on-use to obtain self-configuration, self-optimization and part of self-healing. They basically differ in the way of handling join, leave and failure events for self-organization and self-healing. DKS attempts to provide atomic join/leave with a lock-based algorithm without handling failures very well. P2PS treats leaves and failures as the same event focusing more on fault-tolerance, using a lock-free algorithm. Both systems has problem with non-transitive connectivity.

2.3.4 Chord$^{\#}$, SONAR and Scalaris

Overlay The ring-based peer-to-peer systems we have presented until now rely on the keys having a uniform distribution in order to balance the network. If the keys present a different distribution, some peer will be more loaded than others, causing degradation in the performance of the system. Chord$^{\#}$ [SSR07] proposes a change on the address space and support for multiple range queries transforming the ring into a multi-dimensional torus. Chord$^{\#}$has been derived from Chord by substituting Chord's hashing function by a key-order preserving function. The address space goes from characters A to Z, continuing with characters from 0 to 9. These characters are just the first on of a string that determines the key. The address space is therefore infinite, circular and with total order. It has a logarithmic routing performance and it supports range queries, which is not possible with Chord.

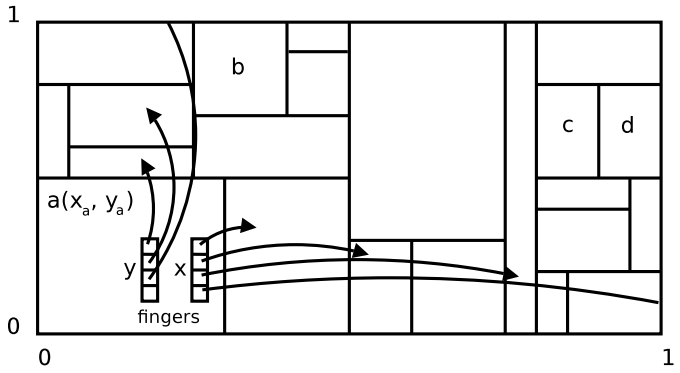


Figure 2.7: Two-dimensional data space in SONAR with routing fingers.

Its $O(1)$ pointer update algorithm can be applied to any peer-to-peer routing protocol with exponentially increasing pointers.

The change on the address space from integers to strings of characters can be applied to any of the previously discussed rings, so as to provide better load-balance when it is known that the application to be developed has not a uniform distribution of keys. The support for multiple dimensions is less evident and we will discuss it more in detail when we describe SONAR. Problems with non-transitive connectivity remain unsolved by Chord$^{\#}$.

SONAR 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, 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. The case scenario corresponds to an overlay network storing the locations of 1,904,711 cities. The graphical location of the cities follows a Zipf distribution [Zip29].

Figure 2.7 is adapted from [SSR07]. It depicts a routing table in a two-dimensional data space. Instead of a ring, it can be seen as a torus. The keys are specified by coordinates (x, y) and hypercuboids cover the complete key space. In the example, peer `a` has coordinates (x_a, y_a). The hypercuboids are presented in the figure as rectangular boxes which are managed by the peers. 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.

Neighbours are determined by adjacent areas, as in CAN [RFH^{+}01], with the difference that not only the neighbours are used for routing but also additional fingers as in Chord. These fingers are chosen along the x and y

axes as it is shown in Figure 2.7 for peer `a`. Note that every finger doubles the distance to the peer. This routing table allows a routing of any message in $O(logN)$, which is much better than $O(\sqrt[d]{N})$ of CAN. What it is not that evident is to decide which neighbour will be taken as finger, because peers can have many neighbours along the same axis. Failure recovery also remains unclear. On the figure, if peer `d` crashes, it is clear that peer `c` who can take over the responsibility of the area, even though the storage will remain unbalanced. In a more difficult case, when peer has more that one neighbour per axis, as peer `b`. Which peer should take over the responsibility? None of them could do it without loosing the square figure. Some none obvious changes of keys would be necessary, and therefore, failure recovery becomes very expensive. This is the main drawback we observe on this approach.

Storage and Scalaris Storage in SONAR is done with multidimensional ranges as we just described, but because their fault tolerance mechanism remains unclear, we will describe the storage in unidimensional Chord$^{\#}$. If we just consider a plain DHT, Chord$^{\#}$'s storage is just like Chord, except that the responsibility of every peer is infinite, although with well define boundaries. Chord$^{\#}$provides one of the most mature fault-tolerant general-purpose storage service. They have implemented symmetric replication as described in [Gho06], where replicas are kept consistent with their own transactional layer [MH07]. The layer provides atomic transactions with the guarantee that at least the majority of the replicas stores the latest value of every item. The transaction would abort otherwise. We will analyse this layer in details in Chapter 6.

Scalaris [SSR08, PRS07] uses this transactional layer over a unidimensional address space in Chord$^{\#}$ to provide a decentralized peer-to-peer version of the Wikipedia. The system has been tested on Planetlab [The03] with around 20 nodes, proving to be more scalable that Wikipedia [Wik09] itself. Wikipedia works with a server farm running MySQL database [AB95]. Their structure is centralized and not very flexible. It does not allow them to easily add new servers to the farm in case their capacity is reached. Scalaris on the contrary, thanks to its peer-to-peer architecture, can easily add new nodes to the network to increase the capacity of the service, scaling without problems.

2.3.5 Pastry and Tapestry

Pastry [RD01b] and Tapestry [ZHS^{+}03] also use a circular address space, but we can classify them as trees because of their finger table and routing algorithm. Apart from trying to achieve a logarithmic routing on the overlay network, they also use IP locality to achieve a better proximity on the underlay network. Pastry and Tapestry differ basically in the way they try to achieve IP locality. We continue the description of Pastry.

Overlay Pastry has three set of nodes: leaf set, locality set and routing table. The leaf set is composed by the $L/2$ successors and predecessors used for replication and failure recovery. The locality set is form by M nodes selected with a metric coming from the underlay network. The routing table is like a tree derived from the way of creating the identifier for every peer, which is a string of digits of base 2^b. The first level of the tree keeps a pointer to $2^b - 1$ peers, with each peer having a different starting digit on its string key. The second level of the tree share the same first digit with the peer, but differ in the second one. Consecutively, the tree has $log_{2^b}N$ levels to achieve logarithmic routing. Since there can be several nodes matching the criteria to be part of the routing table, the one with the better IP locality is chosen. To route messages, the peer forwards it to the peer with the most similar string. The values L, M and b are arbitrarily defined by the implementation.

Join, leaves and failures When a peer joins the network, it triggers a lookup for its key and join next to its responsible according to the circular address space. Responsibility is determined by proximity between the string key of a peer and key of the item. Once the peer join, it takes the leaf set of its entry point as its own leaf set, and initialize its routing table base on the routing table of its neighbour. Then, it sends a message to every peer in each of the three sets, making the joining operation very costly. Even though Pastry works as a tree, failure recovery is less expensive than expected. This is basically because tree layers do not depend on the previous one, but only on the key of the owner of the routing table. When a peer fails, it is removed from the three sets, and it is replaced by asking another peer in the same row to provide a new candidate. Furthermore, a total of L nodes needs to update their leaf sets. It is unclear what happens if a peer is falsely suspected of having crashed only by some peers. This situation arrives in case of non-transitive connections.

Lookup Considering the tree structure of logarithmic height, we can deduce that the lookup process takes $O(log_{2^b}N)$ hops to be resolved. This is competitive with DKS, which is a generalization of Chord, but the routing table it is much larger and more expensive to maintain. Even though there is a nice effort to improve IP locality, we consider that the construction of the routing table leads to some ambiguities that makes unclear its implementation.

2.3.6 OpenDHT and Bamboo

Bamboo [RGRK04] is very similar to Pastry and Tapestry. We will not review its topology because of their similarities. We present it in a different section because its implementation, OpenDHT [RGK+05], have a nice contribution by offering a standard API for general purpose DHT. It include

security aspects and it has been running on Planetlab [The03] for several years already, allowing researchers to run their experiments by implementing only the clients to connect to the peer-to-peer infrastructure, without having to implement the whole overlay.

API OpenDHT extends the basic `put(key, value)` and `get(key)` operators to support storage using a secret, which can be used to remove the item. There is also a TTL value to make the item expire. A function `H(x)` is the SHA-1 hash of `x`. The three new operators are:

- `put(k, v, H(s), t)` : write item `(k, v)` using secret `s` that expires after time `t`.
- `get(k)` : returns `{v, H(s), t}`
- `remove(k, H(v), s, t)` : removes item `(k, v)` using secret `s`. Time `t` > TTL remaining for put.

It is also possible to put immutable items that cannot be removed. They only expires. The API for doing this is the following:

- `put-immut(k, v, t)` : write immutable item `(k, v)` that expires after time `t`. Note that since the value cannot be removed, there is no secret associated.
- `get-immut(k)` : returns item `(k, v)`.

There is a third interface for authentication which we will not describe because it is only an option, not a requirement. An important observation is that OpenDHT only guarantees eventual consistency on these three interfaces. Programmers should learn to deal with this property, which we believe is one of the drawbacks of OpenDHT.

Another interesting interface is the inclusion of a *namespace* to identify different applications running independently on the same peer-to-peer network. The *namespace* is used to join and lookup for keys. It is a simple addition that ease the isolation of applications.

- `join(host, id, namespace)` : add `(host, id)` to the list of host participating on the application identified with `namespace`.
- `lookup(k, namespace)` : returns `(host, id)` in `namespace` where `host` is the responsible for key `k`.

It is important to remark that this API can be used by almost any peer-to-peer system we are reviewing on this chapter, specially to improve security. Consequently, we can also use it to improve relaxed-ring's API, but since the research on these two topics can be done independently, we will not use it for this dissertation for the sake of simplicity.

2.3.7 Kademlia

Overlay Kademlia [MM02] and Pastry has the same way of partitioning the address space, but the routing strategy is quite different. The partition is done by assigning every peer an identifier made out of a chain of digits. Peers use a binary tree for routing where they organize other peers according to their shortest unique prefix. Figure 2.8 shows an example of an address space of size N, where we observe the routing table from the point of view of peer with prefix 8, which binary representation $(1000)_2$. The figure is inspired in an example taken from [EAH05]. Note that peers on the tree are just at the leaf level, then, removing peers because of leave or failure is not too costly.

Each peer, as it is shown for peer 8, has at least one finger to each subtree doubling the size of the subtree each time. Ideally, a peer should have k pointers to a subtree, and this is called a k-bucket. Kademlia does not store any close references such as the leaf-set in Pastry or the successor list in Chord. The height of the tree is also logarithmic due to its binary construction. These properties guarantee a logarithmic routing, more precisely $O(log_2 N)$. The criteria for chosen the k fingers in any subtree can depend on every application, and it gives room for IP locality. Kademlia in particular uses metrics to determine which peer has a probably to stay longer connected.

A good point in Kademlia, as in Pastry and Tapestry, is that leaves are treated as failures. To provide fault-tolerance, the k-bucket is very important. Because of its size, it is very unlikely that a peer loses all pointers to a subtree. Although this strategy is robust, it is very expensive to maintain.

Storage and implementations Items are originally stored on the peer that is closest to its key. To determine the distance, Kademlia uses the *bitwiseexclusiveor*, XOR distance. The item is replicated k times to the closest set of peers. There is a periodic stabilization based on republication of the value of the item. The closest peer to the item republish it to the other $k-1$ peers holding a replica. Routing works iteratively and answers as soon as it finds a valid item. Since this is not synchronized with republication, some inconsistencies might occur, but they are eventually fixed thanks to republication. We consider these kind of networks better suitable for file-sharing application where the items do not change their values.

The main implementation of Kademlia is Coral [FFME04], and its DHT protocol has also been used outside the academic world, by the file-sharing applications Emule [Emu04] and Overnet [Ove04].

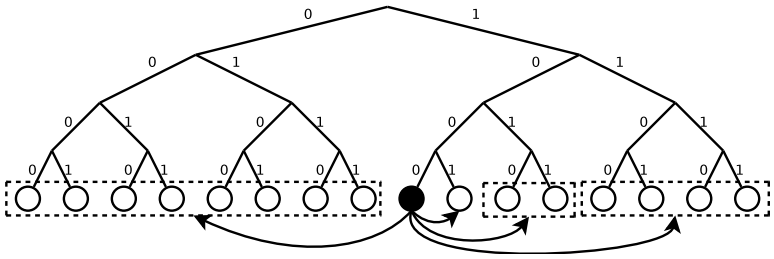


Figure 2.8: Routing strategy in Kademlia from peer 8 $(1000)_2$.

2.3.8 CAN

CAN [RFH+01] is the predecessor of SONAR [SSR07], partitioning the address space exactly as we presented in Section 2.3.4. It overlay graph can be interpreted as an hypercube or a multi-dimensional torus. Our example of SONAR in Figure 2.7 is similar to an instance of CAN with two dimensions. The main differences are the way of dividing the areas of responsibility and the construction of the routing table. SONAR makes a division of the area load balancing the existing stored items, whereas CAN always divides the region in two halves. SONAR has d sets of long fingers along every dimension but not references to all direct neighbours, whereas CAN does not have such fingers, and it only holds references to direct neighbours. The implication of these differences is that CAN is simpler to partition, the routing tables are smaller, but the lookup cost is higher. SONAR resolves lookups in $log_d N$ hops, whereas CAN resolves them in $O(\sqrt[d]{N})$ hops, where d is the dimension of the torus. Even when routing works well in both systems, the hypercube structure is not very tolerant to failures. Adding nodes is simpler because it just increments the division of the dimensions, but removing a node makes the recovery not evident in many cases, as we described already in Section 2.3.4.

2.3.9 Viceroy

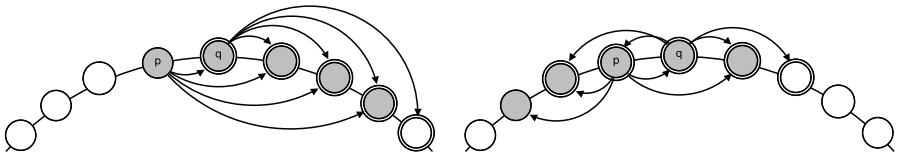
Overlay Viceroy [MNR02] partitions the address space exactly as Chord, mapping items to every peer according to their responsibility, defined by the key of the predecessor and its own key. However, the routing strategy makes us classify it with a different topology, because it is based on the Butterfly graph [Mat04]. All nodes are organized in $log_2 N$ levels, being N the size of the address space. The routing table of every peer has only three fingers. One finger goes to the immediate upper level (*up-finger*), and two fingers to the immediate lower level (*down-fingers*). One down-finger is short, in order to converge lookups, and the other one is long, in order to make significant forwarding. Obviously, peers at the top level do not have up-finger, and peers at the bottom level do not have down-fingers.

The lookup process is quite simple. The lookup request is always forwarded to the upper level until it reaches the top level, and then it follows the down-fingers until reaching the responsible. Each change of level is sent to the closest preceding finger to avoid cycles. The lookup message goes up and down only once, and at least $2log_2N$ hops are performed, therefore, routing is also $O(log_2N)$. Even though the cost of this routing algorithm is very good with a very small routing table, it is not efficient in networks with high amount of traffic, because all messages has to go through the peers at the top level, creating congested nodes. Peers in the second most upper level are also congested, but in a lower degree. As a rule, the higher the level, the more congested the node it is.

Storage and churn The decision about where to store the items is exactly the same as in Chord, so any consistency and replication strategy can be reused in Viceroy. Churn is although treated different. The maintenance of successor and predecessor pointers is done as in Chord, relying on periodic stabilization. The routing table is more difficult to maintain. When a peer joins, it needs to find its level according to some load-balancing estimation. Then, it has to choose its up and down pointer with a combination of lookups, it has to be known by other peers in the two directly upper and down layer. Removing several nodes from the same level may imply more congestion in some level because of the shorter amount of peers.

2.4 Distributed Storage

The initial and still most common goal of peer-to-peer system is file-sharing. Users share with other peers files they store locally. As soon as another peer gets the file, it becomes a *replica* and starts sharing the file with the rest of the network. The rest of the network can access any of the two replicas to get the file, and add a new replica to the network as soon as the transfer is finished. Improvements on file sharing, as in BitTorrent [PGES05], allow peers to share files divided into small chunks. Peers do not need to wait until the whole file is transferred to start sharing the chunks already downloaded. As soon as they get a chunk, they can offer their replica to the network. An even better advantage is that peers can get chunks from different peers at the same time, so the file is transferred much faster. All these protocols work very well based on the assumption that files do not change, which is a very important issue. With that assumption, any replica is a valid one, and therefore, peers need to find only one replica to get the file. This simplifies a lot of problems such as efficient routing and lookup consistency. For routing, you can follow several different paths in parallel until you find one replica. There are no consistency concerns because every replica is a valid one. This is one of the reasons why unstructured overlay networks are still popular for



(a) (b)

Figure 2.9: Strategies for replica placement using the neighbours of the responsible peer: (a) using the successor list, and (b) using the leaf set.

file sharing.

Our work considers items to be not just files, but any sort of data, and particularly application-specific data, which is constantly updated. Most applications rely on reachability and consistency of their data, which is not guarantee on unstructured overlay networks. That is why we focus on SONs providing DHTs, with the basic `put(key, value)` and `get(key)` operations, which store and retrieve items using *key*'s responsible. Naturally, using only one responsible to store an item associated to a certain key is not enough to provide fault tolerance. It is necessary to provide some kind of replication. Since we are more concerned about ring-based SONs, we are going to discuss some replication techniques used in literature in the following section.

2.4.1 Replication strategies

Successor List There are several techniques to organize replicas on a ring. The most basic mechanism is probably the first one proposed in [SMK+01], where $log_2 N$ replicas are stored on the successor list of the responsible of each key. When a node fails, the successor takes over the responsibility, and therefore, it is a good idea that the successor store the replicas of the items. Like that, it does not need to ask the value to other peers to continue hosting the item. This implies that each peer in the successor list must have the latest value of the item. The strategy is depicted in Figure 2.9(a). We can observe in the figure peers p and q, and their replicas stored in their respective successor lists. The replicas of peer p are shown in colour grey. The replicas of peer q are shown with a double circle. The pointers to the replica set are also added, because they are part of the resilient information that every peer needs to have, so it does not add extra connections. An important observation is that q belongs to the replica set of p, but p does not belong to q's replica set. Actually, every replica set is different.

Leaf Set A very similar strategy is the one used by networks having an overlay topology like Pastry, using the leaf-set [RD01b, RD01a] for storing the replicas. Figure 2.9(b) shows the replica set of peers p and q following this strategy. As in Figure 2.9(a), peers in grey are the replicas of p, and

peers in double circle are the replicas of q. This strategy also generates a different replica set for each peer. It has the same advantages as in the successor list strategy, because it does not add extra connections, and the peer that should take over the responsibility in case of a failure already has the values of the replicas.

There are two main disadvantages on these two schemes. First of all, churn introduces many changes on the participants of the replica sets. Each join/leave/fail event introduces changes in $log_2 N$ replica sets, affecting peers that are not directly involved with the churn events. When a new peer n joins the network, it becomes part of the replica set of all peers of which n becomes a member of the successor list. This is still reasonable, but it also implies that the farthest peer on each successor list affected by n will stop being part of the replica set. In a similar way, every leave or failure will imply that a new peer needs to be added at the border of the successor list, and therefore at the border of the replica set. These changes make replica maintenance more costly. The second disadvantage is that there is a unique entry point for each replica set. To find the successor list of the responsible of a key, first you need to find who is the responsible. This means that the main peer of the replica set is a point of congestion. And if the main peer fails, first, the network needs to recover from the failure in order to give access to the other replicas.

Multiple Hashing CAN [RFH+01] and Tapestry [ZHS+03] used multiple hashing as replication strategy. The idea is that every item is stored with different hash functions known to all peers in the network. In Figure 2.10(a) we observe replicated items with keys i and j. The result of applying hash function h_1 to key i results in having peer p as responsible. Applying $h_2(i)$ and $h_3(i)$ gives peers a and d as responsible of the other replicas (painted in grey). Similarly, peers q, b and c (drawn with a double circle) represent the replica set of the item stored using key j. Note that replicas can be stored any where, and that hash function does not represent any order on the ring. One disadvantage claimed in [Gho06] is that you need to know the inverse of the hash functions to recover from failures. For instance, if peer p crashes, peer a would need to know the inverse function of $h_2(i)$ in order to retrieve the value of i, and discover where to store the replica $h_1(i)$. This problem can be solved by also storing the original key of the item, instead of only the hash key, as it is discussed in Section 2.4.2.

A more crucial disadvantage is the lack of relationship between the replica sets per item. In the example of Figure 2.10(a), the replica set of item i is formed by p, a and d. If we take another item with key k, where the responsible peer of $h_1(k)$ is also peer p, it is very unlikely that $h_2(k)$ and $h_3(k)$ would result in hash keys within the responsibility of peers a and d. Therefore, there will be a different replica set for almost every item stored in

the network, making the reorganization of replicas under every churn event very costly. When a new peer takes over the responsible of another peer, either because the other peer failed, or because the new peer joined the network as its predecessor, the new peer will have to contact the responsible peers of h_2 and h_3 of every item stored in the range involved in the churn event.

Symmetric Replication This is a simple and effective replication strategy presented in [Gho06] with several advantages and few disadvantages. First of all, it does not have an entry point of congestion as with the successor list and the leaf sets. Members of the replica set are not indirectly affected by churn, and as in multiple hashing, the replicas are spread across the network, with the advantage that they are symmetrically placed using a regular polygon of f sides, where f is the chosen replication factor. This strategy provides an easier way of finding the replicas, and it balances the load more uniformly.

An example of symmetric replication is depicted in Figure 2.10(b). The replicas of the items where peer p is the original responsible are painted in grey using $f = 4$ as replication factor. Replicas of peer q are drawn with a double circle as in previous examples. A small disadvantage is that it is not possible to guarantee that all replicas of all items stored on p will have exactly the same replica set. It will depend on the distribution of the address space amount the nodes. Even though this is guaranteed in strategies using the successor list or the leaf sets, there is another advantage of symmetric replication that overcome this drawback. As we saw in the analysis of the successor list, every peer has a different replication set. In symmetric replication, it is possible that f peers share the same replication set in the ideal case, and in real cases, they will share most of the replicas with the same peers. In Figure 2.10(b) we have added peer m as if it were an ideal case. In this example, the replica set of peer m is exactly the same as the one of peer p, where every peer stores the replica of the other members of the set. This property cannot be guaranteed for all keys, but it minimizes enormously the amount of nodes that a peer needs to contact in order to recover from a failure.

A disadvantage shared by multiple hashing and symmetric replication is that both rely on a uniform distribution of peers on the address space. However, this assumption is very reasonable since many SONs also rely on this property in order to achieved the promised logarithmic routing. In case of very skewed distribution, one could observe that one peer is the responsible of two replica keys, decreasing the size of the replica set, and therefore, decrease the fault tolerance factor. This is less probable in symmetric replication, and the larger the network, the less probable this event is to happen, so it scales up without problems.

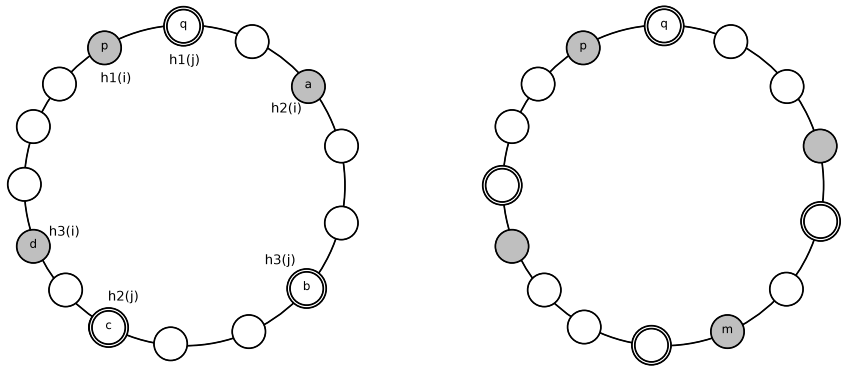


Figure 2.10: Strategies for replica placement across the network: (a) using multiple hash functions, and (b) using symmetric replication.

2.4.2 How to store an item

One of the most basic operations offered by a DHT is `put(key, value)`, where a hash function $h(key)$ is used to determine the *hash key* so as to find the responsible of the item to be stored in the network. But how is the item stored in the peer? Let us consider the example of the operation `put(foo, bar)`. Then, let us say that $h(foo) = 42$, then, most networks assume that the item to be stored is $i = (42, bar)$. That is why in [Gho06] it is claimed that you need the inverse hash function to work with multiple hashing as replication strategy. Another problem is that the diversity of keys that can be used is limited by the chosen size of the address space. If $N + 1$ keys are used in an address space of size N, there will be at least two keys having the same hash key, meaning that one of the two values will be lost. Choosing a very large value of N will unnecessarily result in a large routing table based on the value of N, and it still represents a limit on the maximum amount if keys to be used.

A better way of storing an item, however more costly, it is to store the hash key and the key together with the value. Like that, our `put(foo, value)` operation would result in storing the item $i = (42, foo, bar)$. If another operation has the same *hash key*, say `put(alice, bob)`, with $h(alice) = 42$, it would simply add item $j = (42, alice, bob)$. The chosen value of N would not limit the amount of keys to be used, but only the amount of peers that can join the network. This simple analysis is actually often omitted by several network description, but as we can see, it can has some implications on the need for inverse hash functions.

2.4.3 Transactions

A general architecture for building DHTs organizes the functionalities in bottom-up layers. The overlay graph maintenance together with lookup resolution is placed at the bottom layer. The DHT functionalities `put` and `get` are normally built on top of the bottom layer. Replica management, with the chosen replication strategy as we discussed in Section 2.4.1, is a layer built on top of the basic DHT. We will continue discussing this architecture in the following chapters but now we will say a word about keeping the replicas consistent. There are basically two choices: adding the consistent maintenance at the replica layer, or built a transactional layer on top of it that can guarantee not only that the replica set of an item is kept consistent, but also that the replica set of several items is kept consistent in an atomic operation.

We will discuss transaction in detail in Chapter 6, but first we will review what has been done in the subject. The transactional layer has the goal of providing the ACID properties to data storage on the DHT. This goal also holds even if the transaction is applied to only one item. ACID properties concern: Atomicity, Consistency, Isolation and Durability. Orthogonally to the election of where to store the replicas, the main issue to solve is how to manage the update of the replicas to provide a consistent access to the state. The classical approach is Two-phase-commit, which is not suitable for peer-to-peer because it presents a single point of failure. This problem can be overcome using replicated transaction managers, and Paxos consensus algorithm where the majority of the replicas decides on the update of the value of an item. Another alternative is Three-phase-commit, which even though uses less messages per round, it introduces an extra round to the protocol which is undesirable on peer-to-peer networks. Later on we will discuss the validation of such algorithms, and how Paxos consensus can be adapted to provide a more eager way of performing a distributed transaction. This is optimistic vs pessimistic approach.

Ivy [MMGC02] is one of the earliest work having a transaction-like system for distributed storage. It is built on top of Chord [SMK+01], and it is based on versioning logs per peer. Updates on the replicas do not guarantee consistency, but the log information is meant to be useful for conflict resolution. A more complete and fully transactional protocol [MCGV05] was designed for P2PS [MCV05], but it was never implemented. This protocol was based on two-phase commit having the problem of relying on the survival of the temporary transaction manager, in order to complete the transaction. The goal of their protocol was to show that it was feasible to build decentralized transactions. Paxos consensus algorithm was presented in [MH07] and implemented in Scalaris [SSR08] and Beernet [MHV08].

2.5 Summary of Overlay Networks

In previous sections of this chapter, we have described the main features, advantages and disadvantages of several overlay networks. In this section we summaries them trying to apply the same criteria to all of them. For the sake of visibility, sometimes we will present the information in tables, and sometimes as lists.

2.5.1 Unstructured and Structured Overlays

As we already presented it, most of peer-to-peer networks can be classified into three different generations. The first one still relied on an hybrid architecture where servers were needed to bootstrap any peer-to-peer service, and peers were not able to route message. We are more interested in the second and third generations also called unstructured and structured overlay networks, both of them being completed decentralized and able to self-organize. Table 2.1 shows a summary of the main features of these two generations. Both of them use the most suitable routing strategy according to the overlay topology uses to organized the peer. SONs provides stronger guarantees, and by providing a DHT, file sharing is also possible, and not only for popular items, because reachability is also guaranteed. An interesting issue related to the topology used is the fact that unstructured networks do not need transitive connectivity, whereas SONs really upon this property which cannot be guaranteed in all cases on the Internet, basically due to latency and NATs. The fact that the overlay topology is more relaxed in unstructured network provide this advantage, with the cost that lack of structure prevent from providing strong guarantees. This conclusion will become very important when we motivate the relaxed-ring in Chapter 3.

2.5.2 Structured overlay graphs comparison

We now have a look at the features of the structured overlay networks described in Section 2.3. Table 2.2 summarizes their overlay graph and the complexity of their routing cost. The routing cost considers only the amount of hops a lookup request needs to reach the responsible of a key. It does not considering the total amount of messages sent. For instance, in Kademlia, the lookup request follows several paths in parallel until it reaches one of the replicas. But, the table only considers the amount of hops of the successful path. We have classified Pastry, Tapestry and Kademlia as trees, because the graphs defined by the fingers and leaf sets form a tree. However, they also use a circular address space organizing peers in a ring. This means that failure handling is similar to the networks we have classified as rings, with the addition of the extra updates that are needed on the leaf sets.

We have classified Chord$^{\#}$and SONAR differently, even though both of

Table 2.1: Comparing unstructured and structured overlay networks.

	Unstructured	Structured
Topology	Random	Ring, Hypercube, Tree, etc.
Routing	Flooding	Directed using meaningful routing tables
Guarantees	Lookup converges most of the time	Routing cost is bound, mostly logarithmic. All peers are reachable
Provides	File sharing	DHT
Transitive connectivity	Not needed	It relies upon it

them support multiple range queries, and therefore, both could be considered multidimensional torus. The reason why we have put Chord$^{\#}$together with other rings, is that its most common used is a ring as in Chord, with a different distribution of the address space. We can observe in Table 2.2 that almost all networks guarantee logarithmic routing, except for CAN, even though its complexity is also very good. The base of the logarithm is the main difference between networks, and it is directly influence by the election of the routing table. Our conclusion is that in general, structured overlay networks are very competitive in terms of routing.

With respect to fault tolerance, failures in a ring mainly affect the successor and predecessor of the failed node. Fingers are also affected but this only comprises efficiency, not correctness. Failures in other overlays, such as trees, hypercubes or butterfly graphs imply changes in a lot more peers that need to reorganize the overlay graph. Adding nodes to these structures can be done very efficiently, but removing peers is very costly. Therefore, we arrive to the same conclusion given in [GGG^{+}03], that ring-based networks are competitive in routing cost, but they are more tolerant to failures.

2.5.3 Ring based overlays

We focus now in the analysis of the different peer-to-peer networks built using a ring as overlay graph. We know that Chord, Chord$^{\#}$ rely on periodic stabilization to fix successor, predecessor and finger pointers. Such strategy has the advantage of treating leaves as failures. Therefore, there is no need to define a protocol for gentle leaves, because pointers will be fixed in the next round of stabilization. However, this implies that stabilization

Table 2.2: Properties of structured overlay networks.

Network	Overlay	Routing
Chord, Chord$^{\#}$, OpenDHT	Ring	$O(log_2 N)$
DKS, P2PS	Ring	$O(log_k N)$
Pastry, Tapestry	Tree	$O(log_{2^b} N)$
Kademlia	Tree	$O(log_2 N)$
CAN	Multidim. Torus	$O(\sqrt[d]{N})$
SONAR	Multidim. Torus	$O(log_d N)$
Viceroy	Butterfly graph	$O(log_2 N)$

Table 2.3: Failure handling on ring-based overlay networks

Network	Mechanism	Leaves and Failures
Chord, Chord$^{\#}$	Periodic stabilization	Leave = Failure
DKS	Correction-on-change	Gentle leave is fundamental
P2PS	Correction-on-change	Leave = Failure

needs to be run often enough, increasing bandwidth consumption. It has been shown in [Gho06] that lookup inconsistencies can appear in Chord just because of churn, even if failures do not occur. These is a serious problem for correctness. To avoid this problem, DKS introduces the concept of correction-on-change, meaning that pointers are fixed as soon as a failure, leave or join is detected. Peers do not wait for a periodic check. To avoid lookup inconsistencies, DKS defines a protocol for atomic join/leave, requiring that a peer respect the locking protocol before it leaves the network. Unfortunately, this strategy is not very fault-tolerant, and it relies on peers not leaving the network before they acquire the needed locks. P2PS also addresses ring maintenance with correction-on-use, but it does not require gentle leave in order to fix the departure of peers. This approach allows P2PS to solve the problem of leaving peers by handling peer failures. A peer that leaves the network could send messages to provide a more efficient fix of pointers, but it is not needed for correctness. The problem with P2PS is definitely not the approach, but the protocol does not work in some cases, as it is shown in [MJV06]. A more fundamental issue, also shared by the other networks, is that P2PS relies on transitive connectivity, and it expects to create a perfect ring with respect to successor and predecessor pointers. Table 2.3 summarises these differences, but it only talks about fixing successor and predecessor pointers. Fingers are also solved with periodic stabilization in Chord and Chord$^{\#}$, but they are addressed with correction-on-use in DKS and P2PS.

2.6 Self-Management properties

In this dissertation we claim that self-management is essential for building dynamic distributed systems. We mentioned six different axes to analyze self-management, and we try to identify them in the overlay networks we have reviewed in this chapter. Since several of these axes are shared by many of the networks, with some small differences, we will not build a comparative table, but we will summary what we have observed for each axis.

Self organization This is the property that it is most intrinsically present in the overlay networks we have reviewed, even in networks such as Gnutella and Freenet, where self-organization is very basic considering the lack of structure. Self organization comes naturally as there is no central point of control, and peers needs to agree their position only by talking to the direct neighbours. The only pre-existing infrastructure needed to boot the system is the Internet, or any underlying network that provides mean to establish point-to-point communication channels. None of these networks have human intervention that assign peers their position in the network. Everything is self-organized. Table 2.3 gives us some hint about some of the mechanisms used to achieved self-configurations. Networks such as Chord and Chord$^{\#}$ use periodic stabilization to keep the ring organized. DKS and P2PS use correction-on-change for the same purpose.

Self configuration First of all we need to clarify what is that we want to configure. This property is usually used to mean self-configuration of components. But that would imply already a design decision of implementing the system using components. Although that would be a very good decision, the networks we have reviewed could be implemented in many different ways. We will discuss self-configuration of components in Chapter 7, here, we are interested in the configuration of the overlay graph. Someone could argue that this is already covered by the property of self-organization, but, if we compare Chord with Pastry, both are organized as rings, but their overlay graphs if configured differently, making Pastry work as a tree. What needs to be configured is the routing table of each peer, which could actually be seen as a component. Self-configuration is achieved in Chord, Chord$^{\#}$, and the majority of the networks, with periodic stabilization. Peers constantly check the status of the neighbours of their fingers, and update the routing table according to these information. Similarly to self-organization, DKS and P2PS use correction-on-change to re-configure their routing tables, but also correction-on-use, piggy-backing configuration messages to the regular messages that are routed through the network, without increasing unnecessarily bandwidth consumption.

Self healing In the context of peer-to-peer networks, this property deals with handling disconnection between peers. Disconnection can be produced not only by the failure of peers, but also due to problems in the communication channel between them. Because of this, it is very important to know if the protocols for failure handling rely on perfect failure detection or not. Internet style are typically strongly complete and eventually accurate, therefore, eventually perfect. As a consequence, a expensive failure recovery mechanism can become very impractical if the network is to be built on top of the Internet. As we conclude earlier on this section, ring-based overlay networks can handle failures more efficiently. Not surprisingly, Chord-like rings achieve self-healing with periodic stabilization. If a node disappears from the network, ring pointers will be fixed in the next stabilization round. If it was a false suspicion, the incorrectness will be fix in the stabilization round after the accuracy is achieved. P2PS and DKS rely on correction-on-change to provide self-healing. This approach requires that the failure detector triggers the corresponding *crash* and *alive* events to react to failures and false suspicions. DKS has the drawback of relying on gentle leaves of peers to respect the locking protocol providing atomic join/leave. What is missing from all the networks we have reviewed, it is a correct handling of non-transitive connections, which can be seen as permanent false suspicions. This is an area where this dissertation also makes a contribution.

Self-healing is not only about fixing pointers on the overlay graph, it also covers repairing stored items. We discussed that plain DHT was not enough to guarantee fault tolerance, and therefore, some replication mechanism is needed. We described several strategies to place replicas, such as leaf sets or symmetric replication. The mechanism that trigger self-healing in any of these strategies is orthogonal to the mechanism for fixing the overlay graph pointers. For instance, Chord$^{\#}$and DKS can implement symmetric replication having the same self-healing mechanism to restore replicas, but they have different mechanism to fix the overlay graphs, namely periodic stabilization and correction-on-change.

Self tuning This topic is not really discussed by the designers of the networks we have studied, but we can identify variables which values can be tuned in order to provide a better behaviour. Periodic stabilization is a clear example where self-tuning makes sense, because the frequency of each stabilization round can be too short, consuming too much bandwidth. Or it can be too long, having always dead pointers that have not been updated, breaking the guarantees the network provides, as it is shown in [KA08]. Other variables than can be tuned are those involved in failure detection: how often a peer needs to be pinged? Or how long does the timeout has to be? These parameters will affect how fast correction-on-change will react. Evidently, since every network needs failure detection, tuning these parameters will not

help only P2PS and DKS, but any network we have described here.

Self optimization The most important value to optimize in peer-to-peer networks is routing. It is very important than fingers are constantly updated to work with routing tables that are optimize. Therefore, there is some overlap with self-configuration, because both properties actuates over the routing tables. As we mentioned before, DKS and P2PS clearly provide self-optimization by means of correction-on-use. The more the network is used to route messages between peers, the more optimized it becomes.

Self protection The only network we have presented in this section that explicitly treated security issues is OpenDHT with its API prepared for storing data with encryption. However, this API does not provide self-protection, because once the keys are compromised, there is not mechanism to detect and repair the encrypted storage. One of the few works on self-protection we are aware of in the scope of peer-to-peer networks, is the study of Small world networks [HWY08] to provide self-protection against the Sybil attack [Dou02]. The work claims that overlay graphs built as small world network is the only solution to the Sybil attack, and therefore, rings, trees or hypercubes, are subject to such attack. In our view, the work is actually based on the good properties of social networks that can identify when an attacker is trying to pretend being different persons to gain reputation. Social networks typically built small world graphs, but the self-protection does not come from the graph, but from the fact that real people is behind the nodes participating in the network.

2.6.1 Scalability

With respect to scalability, there are no major differences between the different structured overlay networks. Almost all of them scale well to very large networks. Viceroy could have some problems considering the bottle neck presented on the highest layer of routing, but the larger the network becomes, more peers will be placed at the highest layer, balancing the load of routing messages. Still, the bottle neck we previously identified remains an issue. What is interesting to observe is that none of these networks seems to be suitable to scale down. In small peer-to-peer networks one could benefit from full mesh networks, instead of using sophisticated routing tables to provide logarithmic routing. This dissertation also makes a contribution in this topic with a self-optimized routing table that is able to scale up and down, a property that can also result interesting in Cloud Computing, as we will discussed in Section 2.8.

2.6.2 Replicated Storage

While discussing self-healing, we mentioned that several strategies could be adopted in order to provide storage replication. A summary of the four replication strategies described in Section 2.4.1 is presented here based on where are the replicas stored, which network implements it, what is the semantic of the replica set, advantages and disadvantages.

- **Successor List**.
  - **Implemented** in Chord.
  - **Replica set** per peer. Placed in the successor list.
  - **Advantages** In case of failure, the successor has already the replicated values.
  - **Disadvantages** To find the replicas, the responsible of the hash key needs to be found first. There is a central point of congestion. Churn affects border members of the set.
- **Leaf Set**.
  - **Implemented** in Pastry.
  - **Replica set** per peer. Placed in $f/2$ successors + $f/2$ predecessors.
  - **Advantages** Same as successor list, plus, since some replicas can be found earlier because they are place on the predecessors.
  - **Disadvantages** Same as successor list.
- **Multiple Hashing**.
  - **Implemented** in CAN and Tapestry.
  - **Replica set** per key. Placed on the responsible of every hash function.
  - **Advantages** Spread across the network. No point of congestion.
  - **Disadvantages** Too many replica sets. Depending on the storage, the inverse hash function might be needed.
- **Symmetric Replication**.
  - **Implemented** in Chord$^{\#}$and Beernet.
  - **Replica set** per key but approximately organized in sets of f peers. Placed symmetrically across the network.

- **Advantages** Load balancing of replicas symmetrically spread across the network. Any replica can be accessed without having to contact the main responsible. Most of the time one replica set groups several items and f peers, making groups more closely connected. No point of congestion. No need for inverse hash function.
- **Disadvantages** It relies on uniform distribution of peers on the address space.

2.7 A Note on Grid Computing

Peer-to-peer networks and Grid computing have the common goal of sharing resources and make them available to their users. Both systems need to locate the resources coming from different heterogeneous processes. Despite these similarities, the approaches differ in several aspects due to their different characteristics. Peer-to-peer is conceived to allow collaboration among untrusted and anonymous users, whereas Grid users are typically trusted and identified members from research institutions. Peer-to-peer is fundamentally decentralized and can scale to very large networks. Grid networks on the contrary, are comparatively smaller and do not scale because Grids are mainly built gathering sets of clusters from well connected organizations. This means that there is almost no churn and failure detection is much more accurate. Such scenario has allowed Grids to be designed as centralized and hierarchical. However, Grids are becoming larger and larger, and the centralized approach will have to leave its place to decentralized self-management. As mentioned in [FI03, TT03], Grid computing and peer-to-peer will eventually converge. There is already research that targets Grid computing from a peer-to-peer approach [IF04, TTZH06a, TTZH06b, TTF$^+$06, MGV07, TTH$^+$07], providing fault-tolerance, resource discovery, resource management and distributed storage. These works indicate us that the results from this dissertation can be applied to Grid computing if the design is merged with peer-to-peer systems. Even so, our research is orthogonal to Grid.

2.8 A Note on Cloud Computing

Cloud computing is one of the latest emerging research topics in distributed computing, and therefore, it is necessary to contextualize it in this dissertation. Cloud computing has many definitions with different views within industry and academia, but everybody agrees on that cloud computing is the way of making possible the dream of unlimited computing power with high availability. Cloud computing has been active in the IT industry for a couple of years already, calling immediately the attention of the research community thanks to its possibilities and challenges. Projects such as Reservoir [Res09],

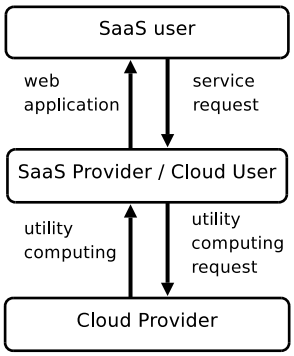


Figure 2.11: General Cloud Computing architecture.

XtreemOS [Par09] and OpenNebula [Dis09] are just examples of the interest of the research community. However, defining Cloud computing it is not that simple. One of the interpretations sees Cloud computing as ubiquitous computing, focusing on *high availability* and on the idea that computations are mainly done elsewhere, and not on user's machine. Another view considers any application provided as a web service to be *living in the cloud*, where the cloud is simply the Internet. We share Berkeley's view of Cloud Computing [AFG+09] and the conclusions of 2008 LADIS workshop [BCvR09]. We see Cloud Computing as the combination of hardware and software that can provide the illusion of infinite computing power with high availability.

Large companies provide this illusion of infinite computing power by having real large data centers with software capable to provide access on demand to every machine on the data center. Industrial examples supporting Cloud Computing are Google AppEngine [Goo09a], Amazon Web Services [Ama09] and Microsoft Azure [Mic09]. This three companies follows the architecture described in [AFG+09] where the base of the whole system is such a large company being the cloud provider. Cloud users are actually smaller companies or institutions that use the cloud to become Software as a Service (SaaS) providers. The end user is actually a SaaS user, which is indifferent to the fact that a cloud is providing the computational power of the SaaS.

Figure 2.11 depicts the general architecture described in [AFG+09]. The *cloud provider* is at the base of the architecture offering *utility computing* to the *cloud user*. Utility computing can be understood as a certain amount of resources during a certain amount of time, for instance, a web server running for one hour, or several Tera bytes of storage for a certain amount of days. The cloud user, which is actually a *SaaS provider*, has a predefined utility computing request, which can vary enormously depending on its users demands. At the top of the architecture we find the *SaaS user* which requests services from the SaaS provider. The service that the SaaS provider offers to its users is usually presented as a web *application*.

Given such architecture, there are two parts that might be relevant to

this dissertation: the cloud provider and the SaaS provider. First of all, it is necessary to build a network capable of managing the resources at the bottom of the architecture. XtreemOS [Par09] focuses on that part, and sees Cloud computing as an extension to Grid computing. OpenNebula [Dis09] not only allows the management of local cloud infrastructure, but it also abstracts the cloud provider introducing an interface layer to the SaaS provider. Like this, the SaaS provider could choose to manage its own infrastructure, or hire a large cloud provider such as Amazon. The interesting thing here is that once the cloud provider is abstracted, it raises the possibility of using multiple cloud providers behind the interface layer. The interface layer would then work as a resource broker that decides to which provider the request will be sent. Like this, a company can rely only on its own resources until a peak in users' demand appears, and then, it hires some extra resources from Amazon, only the necessary. Schemes like this are starting to be developed by projects such as Reservoir [Res09] and Nimbus [The09b]. In the later, they also talk about *sky computing environment* where several clouds are constantly providing the needed resources. A very similar idea is presented as work in progress in XtreemOS under the names of *community cloud* and *cloud federation*, making the difference on who is providing the resources. Having several cloud providers behind the interface layer will make the use of a centralized resource broker unusable considering fault tolerance and scalability, giving room for research on decentralized systems.

From the point of view of the SaaS provider, it has to be able to request and release resources according to the demand of its SaaS users. In other words, its own network must quickly scale up and down to maximize quality of service, and to minimize costs. This looks as a network with controlled churn which might be interesting to investigate as a peer-to-peer network.

2.9 Conclusion

This chapter summarizes large part of the work that has been done in decentralized systems. We went through the three generations of peer-to-peer networks, making a deep analysis of structured overlay networks, known as the third generation. We still identify interesting properties from the second generation, represented by unstructured overlay networks. We have reviewed them in section 2.2.3, analysing how they where able to build a completely decentralized system where peers did not need any central point of management to organize themselves. Although unstructured overlay networks are successful for building file-sharing services, their applicability is quite limited to that. This is, among other reasons, due to their lack of guarantees of reachability and lookup consistency. SON is the attempt to provide these properties in peer-to-peer networks, and the way to achieve them is by giving structure and increasing self-management.

From all SON topologies, we have observed that all of them are very competitive in terms of routing complexity, and that ring-based networks performed better in terms of failure recovery. We have also analyzed the networks using six axes of self-management: self-organization, self-configuration, self-healing, self-tuning, self-optimization and self-protection. We have identify that within ring networks there are two main tendencies to achieve several of the self-management properties. Chord-like networks based their network management on periodic stabilization, whereas DKS and P2PS use correction-on-change and correction-on-use, have a more efficient bandwidth consumption.

We have also discussed the problem that nearly all protocols we have reviewed rely on transitive connectivity. It has been observed even by the authors of Chord, Kademlia and DHT, that non-transitive connectivity generates several problems when these networks have been deployed as real systems running on top of the Internet. The lesson learn was that non-transitive connectivity should be taken into account from the design of the network. We also add imperfect failure detection as one of the issues that needs to be considered in the designed. We also noted that transitive connectivity is not really an issue in Gnutella or Freenet, partly because of their relaxed overlay network.

The most characteristic feature of SONs is that they provide DHTs. Being aware that plain DHT support is not enough to provide storage fault tolerance, we showed several replication strategies in Section 2.4. Two of them try to take advantage of the overlay graph placing replicas either on the successor list or on the leaf set. The disadvantages are that in both cases there is a single entry point that creates a bottle neck, and that churn generates a lot of traffic associated to replica maintenance. To balance the load of replica storage, we have analyzed multiple hashing and symmetric replication. This last one is the simplest one to implement and minimizes the differences between related replica sets. Therefore, symmetric replication appears as the most convenient replication strategy.

Finally, we have briefly reviewed how peer-to-peer networks are related to Grid computing and Cloud Computing. Basically, peer-to-peer has mainly been used to implement resource-discovery services on the Grid but they are rarely used as overlay network to organize them. There is no direct connection to Cloud Computing as it is now, having its business model based on a single large cloud provider. We believe that the new academic tendency to conceive Cloud Computing with several small cloud providers will give a chance to decentralized systems and peer-to-peer networks to contributed to the self-management of the system.

Chapter 3

The Relaxed-Ring

One ring to rule them all
One ring to find them
One ring to bring them all
and in the network bind them

freely adapted from *"The Lord of the Rings"* -
J.R.R. Tolkien.

Chord [SMK+01] is the canonical structured overlay network using ring topology. Its algorithms for ring maintenance handling joins and leaves have been already studied [Gho06] 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 [LMP04, LMP06] 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 [Gho06], using DKS [AEABH03] 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 di-

rection because it request the locks of only two peers instead of three (as in [LMP04, LMP06]. More details about how it works are given in Section 2.3.2. One of the problem with this algorithm is that it 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.

As we have already discussed in Chapter 2, the fundamental problem that affects all these algorithms is that they all rely on transitive connectivity, a property that is not guaranteed by Internet style networks. If we observe the matrix of delays of the King Data Set [GSG02], we observe that 0.8% of the nodes cannot talk to each other, introducing non-transitive connectivity. The authors of Chord, Kademlia and OpenDHT described in [FLRS05] that their measurements on PlanetLab [The03] presented 2.3% of all pairs having problems to communicate. The conclusion is that non-transitive connectivity must be taken into account from the design of a decentralized system, and this is what the Relaxed-Ring pretends to do. Create a ring-based network with good properties on lookup consistency, despite the existence of non-transitive connectivity.

3.1 The Relaxed-Ring

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 component. 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

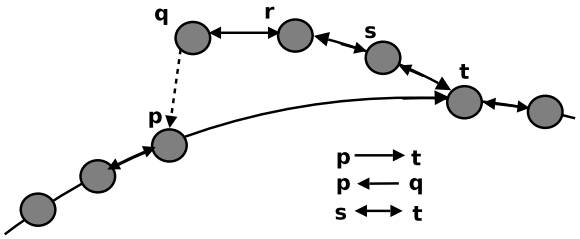


Figure 3.1: A branch on the relaxed-ring created because peer q cannot establish communication with p. Peers p and s consider t as successor, but t only considers s as predecessor.

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 3.1 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.

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 3.1, 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 [MV07], and it has been recently confirmed by Shafaat [SMS+08].

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 Chapter 5 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 3.2.

Algorithm 1 Starting a peer and the lookup algorithm

```
procedure init(accesspoint) is
    self := getRandomKey()
    succ := nil
    pred := nil
    predlist := ∅
    succlist := ∅
    send ⟨ lookup | self, self ⟩ to accesspoint
end procedure

upon event ⟨ lookup | src, key ⟩ do
    if (key ∈(pred, self]) then
        send ⟨ reply_lookup | self ⟩ to src
    else
        p := getBetterResponsible(key)
        send ⟨ lookup | src, key ⟩ to p
    end if
end

upon event ⟨ reply_lookup | i ⟩ do
    send ⟨ join | self ⟩ to i
end
```

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 3.1, 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.

3.2 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 3.2, 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.

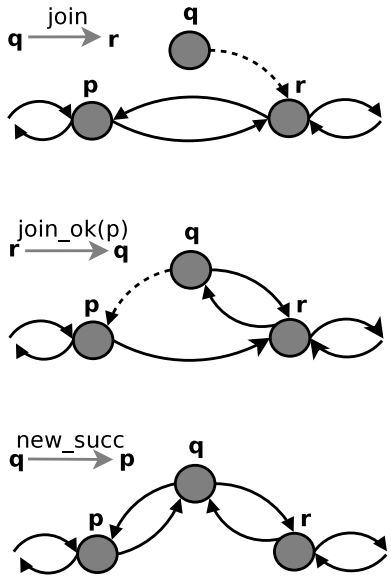


Figure 3.2: The join algorithm.

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 3.4. 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 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 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 3.4. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Algorithm 2 Join step 1 - adding a new node

```
upon event ⟨ join | i ⟩ do
    if succ == nil then
        send ⟨ try_later | self ⟩ to i
    else
        if betterPredecessor(i) then
            oldp := pred
            pred := i
            predlist := {oldp} ∪ {predlist}
            send ⟨ join_ok | oldp, self, succlist ⟩ to i
        else if (i < pred) then
            send ⟨ goto | pred ⟩ to i
        else
            send ⟨ goto | succ ⟩ to i
        end if
    end if
end

upon event ⟨ join_ok | p, s, sl ⟩ do
    succ := s
    succlist := {s} ∪ sl \ getLast(sl)
    if (pred == nil) ∨ (p ∈ (pred, self)) then
        pred := p
        send ⟨ new_succ | self, succ, succlist ⟩ to pred
    end if
end

upon event ⟨ goto | j ⟩ do
    send ⟨ join | self ⟩ to j
end
```

Theorem 3.2.1 *The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.*

Proof 1 *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*

Algorithm 3 Join step 2 - Closing the ring

```
upon event ⟨ new_succ | s, olds, sl ⟩ do
    if (succ == olds) then
        oldsucc := succ
        succ := s
        succlist := {s} ∪ sl \ getLast(sl)
        send ⟨ join_ack | self ⟩ to oldsucc
        send ⟨ upd_succlist | self, succlist ⟩ to pred
    end if
end

upon event ⟨ join_ack | op ⟩ do
    if (op ∈ predlist) then
        predlist := predlist \ {op}
    end if
end
```

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.

3.2.1 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 4 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.

Algorithm 4 Update of successor list

```
upon event ⟨ upd_succlist | s, sl ⟩ do
    newsl := {s} ∪ sl \ getLast(sl)
    if (s == succ) ∧ (succlist ≠ newsl) then
        succlist := newsl
        send ⟨ upd_succlist | self, succlist ⟩ to pred
    end if
end
```

3.2.2 Pruning Branches

Let us consider again figure 3.1. 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.

To implement the hint message, a first naive approach was to send the `hint` to all the peers in the *predlist* as soon as *join_ok* was sent to the joining node. There is a possible mix of successors here, and even some possible lookup inconsistencies that could be introduced. It is related to non-transitivity, but it could also be caused just by the wrong order of messages. A correct solution is to send `hint` only when *join_ack* arrives to the root of the branch. Moreover, the `hint` is sent to only one peer and not to the whole *predList*. The message is sent to the closest peer in that set.

3.3 Leave Algorithm

There is no algorithm for handling graceful leaves. This is because any protocol designed to handle these kinds of leaves will have to deal anyway with partial failure during the leave, so the work will be redundant. We consider graceful leave a special case of a failure. There can be some leave message in order to speed up failure detection, but that would be just for efficiency.

3.4 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 5. 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.

Algorithm 5 Failure recovery

```
upon event ⟨ crash | p ⟩ do
    succlist := succlist \ {p}
    predlist := predlist \ {p}
    crashed := {p} ∪ crashed
    if (p == succ) ∨ (p == succ_candidate) then
        succ := nil
        succ_candidate := getFirst(succlist)
        send ⟨ join | self ⟩ to succ_candidate
    end if
end

upon event ⟨ alive | p ⟩ do
    crashed := crashed \ {p}
end
```

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 chapter

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 [CV06]. 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.

Figure 3.3 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.

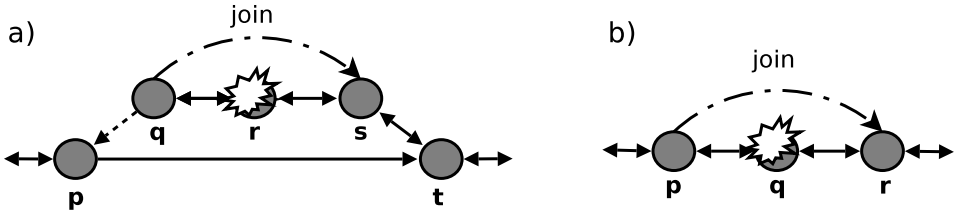


Figure 3.3: Failures simple to handle: (a) In a branch, q and s detect that r has crashed. Only q triggers failure recovery. (b) Pers p and r detects q has crashed. Peer p triggers the recovery mechanism.

Having now the knowledge of the *crashed* set, algorithm 6 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 6 Verifying predecessor candidate

```
function betterPredecessor(i) is
    if (i ∈ (pred, self)) then
        return (true)
    else
        return (pred ∈ crashed)
    end if
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 7. 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.

Algorithm 7 Modified goto

```
upon event ⟨ goto | p ⟩ do
    if (p ∉ crashed) then
        send ⟨ join | self ⟩ to p
    else
        send ⟨ join | self ⟩ to succ_candidate
    end if
end
```

Figure 3.4 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 analysed. Figure 3.5 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.

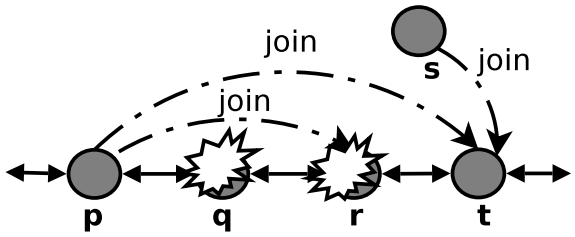


Figure 3.4: Multiple failure recovery and simultaneous join. Peer p detects the crash of its successor q. First successor candidate r has also crashed. Peer p contacts t at the same time peer s tries to join the network. Both *join* messages are the same.

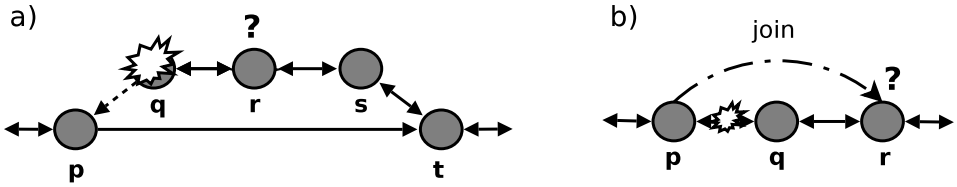


Figure 3.5: Failures difficult to handle: (a) failure of the tail of branch, nobody is responsible for range $(p, q]$ (b) broken link generating a false suspicion of p about q.

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 3.4.1 *Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.*

Proof 2 *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 3.4), 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 [Gho06], and it is related to the proof given in [Bre00] about limitations of web services in presence of network partitioning.

Figure 3.6 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 3.4.1 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.

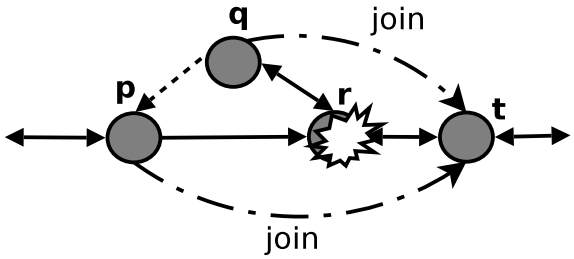


Figure 3.6: The failure of the root of a branch triggers two recovery events

Theorem 3.4.2 *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 3 *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 3.4.2 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.

3.5 Adaptable Routing-Table Construction

In this section, we propose a hybrid reconfiguration mechanism for the routing table called PALTA*: a Peer-to-peer AdaptabLe Topology for Ambient intelligence*. It was originally conceived for an ambient intelligent scenario, but it can be able to any kind of networks. This algorithm takes advantage of the best features of a fully connected network when the number of peers is small enough to allow the devices manage this kind of topology. When the network becomes too large to maintain a fully connected topology, the algorithm will automatically adapt the network configuration to become a relaxed-ring, which can handle a large number of peers by executing more complex algorithms for self-managing the distributed network. We consider different aspects concerning the transition between networks: adaptation of the base algorithms, maintaining the network's coherence and self-healing from inefficient configurations. Evaluation follows in Chapter 5.

PALTA finger table will change its organization when the network size reaches a defined threshold. We will refer to this limit as ω. Even when during the discussion of this chapter we consider the value of ω as uniform for all nodes, the algorithm is designed such that every node can define its own ω to adapt its behaviour according to its own capacities.

To successfully implement this dynamic schema, we need to analyze how the topology will evolve when peers join or leave the network. When the network is created and the number of peers is below ω, the joining peers will perform the *fully connected* algorithm which simply creates a full mesh of peers. When peers detect a network size above ω, all the incoming joining requests will be handled using the *Relaxed Ring* algorithm. The same methodology is followed when, after a number of disconnections, the network becomes smaller than ω. In such case, peers will change their joining algorithm from Relaxed Ring to fully connected.

To be able to make the transition from a fully connected network to a ring, peers need to precisely identify their successors and predecessors at all times. Algorithm 8 shows that the *join* even in PALTA can be seen almost as a method dispatcher, with the subtlety that it checks its predecessor and predecessor pointers in every join before triggering the *join* event of the FULL module. In case that ω is already reached, it is the algorithm of the relaxed ring who will take care of *pred* and *succ* accordingly.

Algorithm 8 Join for PALTA: Adapted fully connected algorithm with transition to relaxed-ring

```
upon event ⟨ join | new ⟩ do
    if size(peers) < ω then
        check_succ_pred(new)
        trigger ⟨ FULL.join | new ⟩
    else
        trigger ⟨ RING.join | new ⟩
    end if
end

procedure check_succ_pred(id) is
    if better_successor(id) then
        succ := id
    end if
    if better_predecessor(id) then
        pred := id
    end if
end procedure
```

The procedure that checks predecessor and successor uses the same functions *better_successor* and *better_predecessor*, which are equivalent to the one described earlier in this chapter, verifying if the key belongs to the corresponding range.

The value of ω can change right after sending message *join* and-or *join_ok*. Due to that, there is no way of knowing if a reply message *join_ok* will correspond to the full connected topology or to the relaxed-ring. Peers need to adapt dynamically to this situation whenever is needed. Algorithm 9 shows how this event is overloaded in PALTA. The first case corresponds to *join_ok* as in the ring, with information of the predecessor, successor and sucessor list. If the routing table is higher than ω, the event is delegated to the relaxed-ring module. If we are in a small network, predecessor and successor are accordingly check, and the successor list is used to trigger the full connected algorithm, which will be used until reaching ω.

Algorithm 9 Join for PALTA: Overloaded event *join_ok*

```
module PALTA

upon event ⟨ join_ok | p, s, sl ⟩ do
    if size(peers) < ω then
        check_succ_pred(new)
        trigger ⟨ FULL.join_ok | s, sl ⟩
    else
        trigger ⟨ RING.join_ok | p, s, sl ⟩
    end if
end

upon event ⟨ join_ok | src, srcPeers ⟩ do
    if size(peers) < ω then
        check_succ_pred(new)
        succList := succList ∪ srcPeers
        trigger ⟨ FULL.join_ok | src, srcPeers ⟩
    end if
end
```

3.6 Conclusion

In this chapter we have presented the 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 complexity of the lookup algorithm is a bit degraded with the introduction of branches. However, we will analyse the real impact of this degradation in Chapter 5, and we will see that it is almost negligible. In any case, 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.

This chapter was also dedicate to described our self-adaptable finger table called PALTA. It is based on the existing fully connected and Relaxed Ring topologies, with adaptations to make them work together. This hybrid

topology features self-organizing and self-adapting mechanisms to ensure a complete connectivity among the connected peers and take advantage of the current network state to have a better use of the available resources. It benefit from fully connected small networks and it makes a smooth transition to larger ones, being able to scale as a large scale structured peer-to-peer network.

Chapter 4

The Relaxed-Ring in a Feedback Loop

The objective of modeling the Relaxed-Ring as a set of feedback-loops, is that it will allows us to understand and analyse its self-management behaviour in a better way. Feedback-loops, as we will see in the next section, are present in every self-managing system, and therefore, applying them in software design seems to be a reasonable path to follow.

4.1 Background

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 [Van06], where feedback loops are introduced as a designing model for self-managing software. 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. Figure 4.1 depicts the interaction of these three concurrent components in a feedback loop. These three components together with the subsystem forms the entire system. It has similar properties to *PID-controllers*, with the difference that the evolution of a running software application is measured discretely.

The goal of the feedback loop is to keep a global property of the system stable. In the simplest cases, this property is represented by the value of a parameter. This parameter is constantly monitored. When a perturbation is detected, a corrective action is triggered. A negative feedback will make the system reacts in the opposite direction to the perturbation. Positive feedback increases the perturbation.

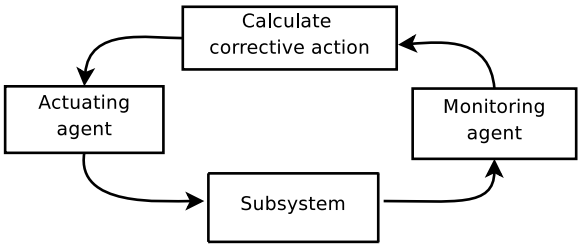


Figure 4.1: Basic structure of a feedback loop (taken from [Van06]).

Taking an air-conditioning as example, we can see the *room* where the system is installed as the subsystem. A thermometer constantly *monitors the temperature* in the room giving this information to a *thermostat*. The thermostat is the component in charge of computing the correcting action. If the monitored temperature is higher than the wished temperature, the thermostat will decide to *run the air-conditioning* to cool it down. That action corresponds to the actuating agent.

Since every component executes concurrently, the model fits very well for modelling distributed systems. There are many alternatives for implementing every component and the way they interact. They can represent active objects, actors, functions, etc. Depending on the chosen paradigm, the communication between components can be done for instance by message passing or event-based communication. The communication may also be triggered by pushing or pulling, resulting on eager or lazy execution.

Independent of the strategy used for communication, it is important to consider asynchronous communication as the default when distributed systems are being modelled.

As a rule for using feedback loops in the design of a system, actuators and monitors appear as verbs, while the subsystem and the computing component appear as substantives, as in the air-conditioning example. The reason why it is not like this in Figure 4.1, is because that is a description of the model, and not the model applied to a system.

4.2 Join Algorithm

In this section we describe the same join algorithm from Chapter 3, but now using a feedback loop. Thinking about the peer-to-peer network as self-managing system, the network is the subsystem we want to monitor, because we want it to keep is functionality despite the changes that can occur. The structure of the ring is the global property that needs to be kept stable. New peers joining, and current peers leaving or failing represent perturbations to the ring structure. Therefore, these events must be monitored.

Messages sent during the process of joining, and the update of the predecessor and successor pointers are shown in figure 3.2. In the example, node q wants to join the network having r as successor candidate. Peer r is a good candidate because it is the responsible for key q. Node q send a join request to r. Whereas event *join* triggered by peer q is a perturbation, event *join_ok* is a correcting action providing negative feedback. It is negative because it is an action that goes in the opposite direction of the perturbation. After *join_ok* is triggered, a branch is created. Then, a second correcting action is needed to entirely close the ring. This action is represented by the event *new_succ* sent from peer q to p.

Figure 4.2 describes the feedback loop that keeps the structure of the relaxed-ring stable. The monitoring agents are in charge of detecting perturbations in the network. Correcting actuators can be seen as three different actions: update routing table (successor and predecessor), trigger event (correcting ones) and forward request (in case a peer wants to join in the wrong place). The routing table does not only include predecessor and successor. It also includes fingers for efficient routing and resilient sets for failure recovery.

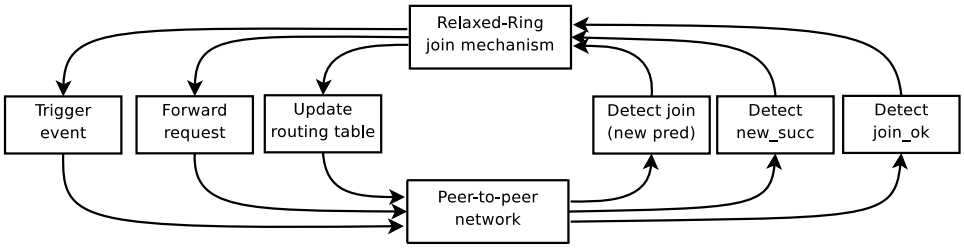


Figure 4.2: Join algorithm as a feedback loop.

Every peer is independently monitoring the network, and the correcting action performing the ring maintenance is running concurrently in every peer. Every event triggered by a peer is monitored by the destination peer, unless there is a failure in the communication. In that case, a *crash* event will be triggered and treated by the failure recovery mechanism.

4.3 Failure Recovery

Instead of designing a costly protocol for peers leaving the network, leaving peers are treated as network nodes having a failure. Like this, solving problem of failure recovery will also solve the issue of leaving the network.

Observing the relaxed-ring as a self-managing system, we identify that the crash of a peer also introduces perturbations to the structure of the ring. Therefore, crashes must be monitored. In order to provide a realistic solution, *perfect failure detectors* cannot be assumed. Perfect failure detec-

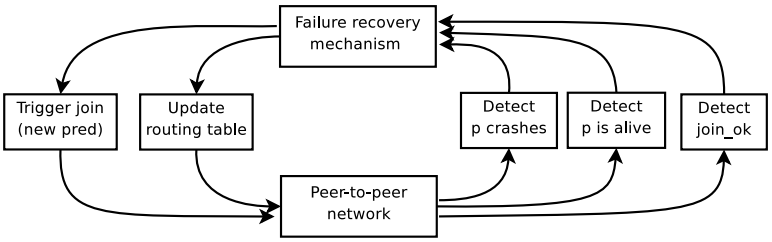


Figure 4.3: Failure recovery as a feedback loop.

tors are strongly complete and strongly accurate. Being complete means that every crashed node is detected. Being accurate means that a node being suspected of failure is effectively in failure. In reality, broken links and nodes with slow network connection are very often, generating a considerable amount of false suspicions. Because of this, not only crashed events must be monitored, but also "I am alive" messages. When these two events are appear as perturbations, the network must update routing tables and trigger correcting events.

In the relaxed-ring architecture we reuse the *join* event as correcting agent for stabilising the relaxed-ring. If the network become stable, the *join_ok* event will be monitored. This negative feedback can be observed in figure 4.3.

The interaction between feedback loops is an interesting issue to analyse because big systems are expected to be designed as a combination of several loops. Let us consider a particular section of the ring having peers p, q and r connected through successor and predecessors pointers. Figure 4.4 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.

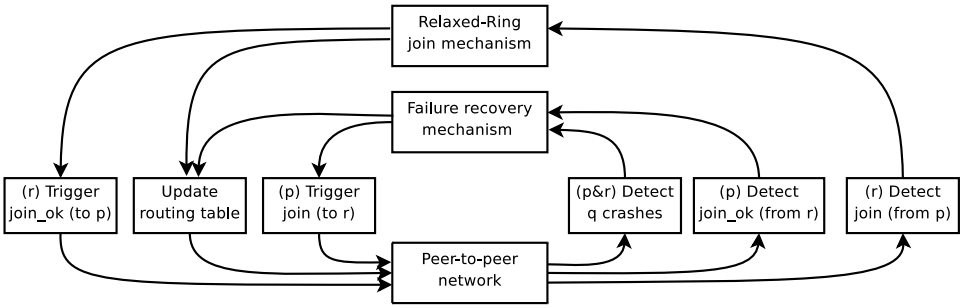


Figure 4.4: Peers p and r detect failure of q, fixing the ring with an interaction of feedback loops.

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.

4.4 Adaptable Routing-Table Construction

Following the strategy we use to model the Relaxed-Ring as a feedback loop in the previous sections, we can also model PALTA as shown in Figure 4.5. The monitors, actuators and the component that decides the corrective actions are placed at every node. The monitored subsystems correspond to the whole peer-to-peer network, and the routing table. The last one is also placed at the node.

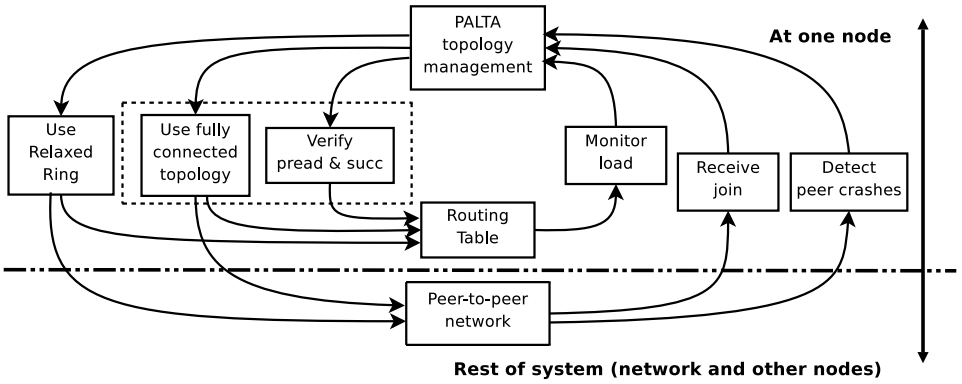


Figure 4.5: Self-Adaptable topology as a feedback loop.

As explained in Section 3.5, when a new node wants to enter the network, it sends a *join* message to its successor candidate. This message is sent through the network. Since every node is monitoring the network, the *join* message will be received by the PALTA component. PALTA is also monitoring the load of the routing table. This information is used to decide how to react to the *join* message. If the load is below ω, PALTA will use the fully connected mechanism together with its own verification of the predecessor and successor. Both actions are used to update the routing table, modifying its load, which will be monitored once again, as in every loop. The fully

connected mechanism will also trigger some messages in the peer-to-peer network in order to modify its state. If the load of the routing table has already passed the ω threshold, PALTA will use the relaxed-ring joining mechanism, which will also update the routing table and trigger some messages for the involved nodes.

We can observe some similitude between PALTA's feedback loop and the acclimatized room. The thermostat in the room will use the heating system or the air-conditioner depending on whether the temperature is below or above the desired goal. PALTA decides its actuators according to load of the routing table with respect to ω. In the acclimatized room, the temperature is measure periodically, being trigger by a timer. In PALTA, it is the join message the one that triggers the monitoring process and the rest of the loop.

The loop also monitors failures of peers triggering the corresponding failure recovery mechanism. This mechanism is chosen by PALTA according to the load of the routing table, as it is done with the joining process. This is coherent with what is explained in Section 3.5. All other messages related to the joining process and the failure recovery, such as $join\_ok$, new_succ, are also present as monitoring event, but they have been omitted from Figure 4.5 for legibility.

4.5 Conclusion

Decentralised systems in the form of peer-to-peer networks presents many advantages over the classical client-server architecture. Even though, the complexity of a decentralised system is higher, requiring the increase of self-management. In this chapter we show how feedback-loops, taken from existing self-managing systems, can be applied in the design of a peer-to-peer network. Using feedback-loops, we can observe that the system is able to monitor and correct itself, keeping the ring structure stable despite the changes due to regular operations of due to network and node failures.

We have also shown how feedback-loops are combined using the subsystem as a way of interacting from one loop to the other. The self-adaptable behaviour of the PALTA finger table becomes more accessible but modelling it as a feedback loop. It is clear that the ω value is constantly monitored in order to adapt the behaviour of the routing table whenever the threshold is reached. It is also clearer how actuators are chosen dynamically according to the values that were monitored.

Chapter 5

Evaluation

After the analysis we have done about the algorithms and self-management behaviour of the Relaxed-Ring, we do now the empirical evaluation comparing it with other networks, specially with Chord [SMK+01]. We start by describing CiNiSMO, our concurrent simulator, and then describe the results we have obtain measuring cost-efficient ring maintenance, lookup consistency, size and amount of branches, and efficient routing.

5.1 Concurrent Simulator

CiNiSMO is a Concurrent Network Simulator implemented in Mozart-Oz. It has been used for evaluating the claims made about the Relaxed-Ring in Chapter 3, 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 used 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 5.1. 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 the Relaxed-Ring [MV08], Chord [SMK+01], Fully connected networks and Palta [CMV+08]. 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

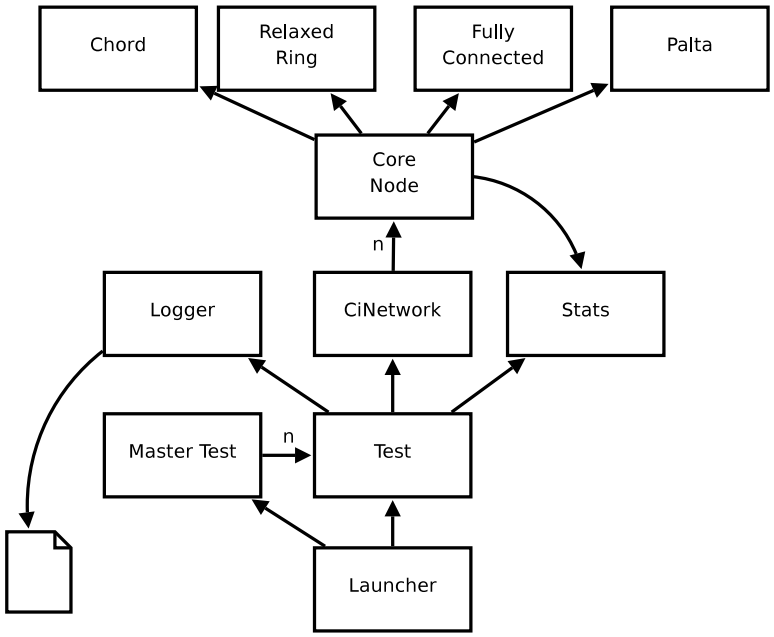


Figure 5.1: Architecture of CiNiSMO.

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 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. The software can be dowloaded at `http://beernet.info.ucl.ac.be/cinismo`, with documentation about how to use it.

5.2 Branches in the Relaxed-Ring

To test the amount of branches that appear on a network, we have bootstrapped networks with different sizes, and using different seeds for random number generation. Every networks starts with a single node, and peers are constantly joining until the network reaches the desire size. Even though there are no failures, the joining activity generates a considerable amount of churn. Since the Relaxed-Ring maintenance is based on correction-on-change, there is no rate we can provide for the churn until we compare it

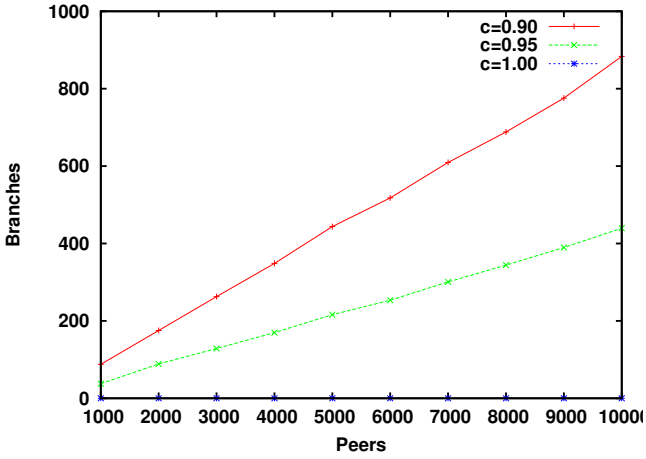


Figure 5.2: Average amount of branches generated on networks with connectivity problems. Networks where tested with peers having a connectivity factor c, representing the probability of establishing a connection between peers, where $c \in \{0.9, 0.95, 1\}$.

with Chord rings, because then we introduce periodic actions that can be compared with churn.

Figure 5.2 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 3.2.2, works well for peer l, then, the branch will remain on size 1. Having this in mind, let us analyse figure 5.3. 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

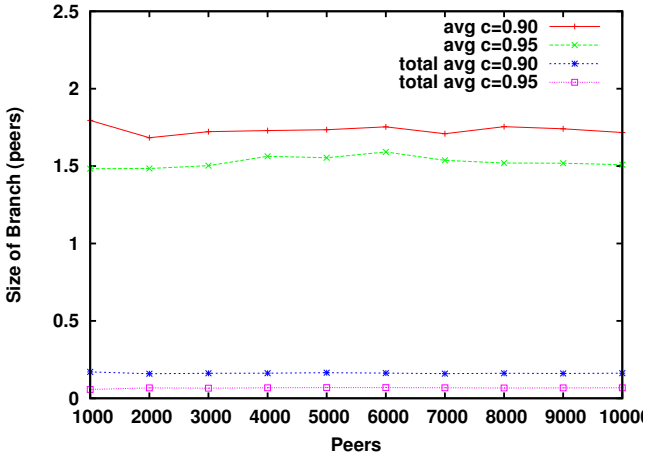


Figure 5.3: Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

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.

The chosen values for coefficient c are a bit worse than some real measurements on the internet. The King Data Set [GSG02] has 0.8% of the nodes that cannot connect with each other. According to [FLRS05], 2.3% of all pairs of nodes in PlanetLab [The03] cannot talk to each other. Therefore, our values for coefficient c seems to be reasonable to study the relaxed-ring.

5.3 Bandwidth consumption

In this section we try to answer the following questions: 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 5.4. We can observe that the amount of messages increases linearly with the size of the network keeping 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.

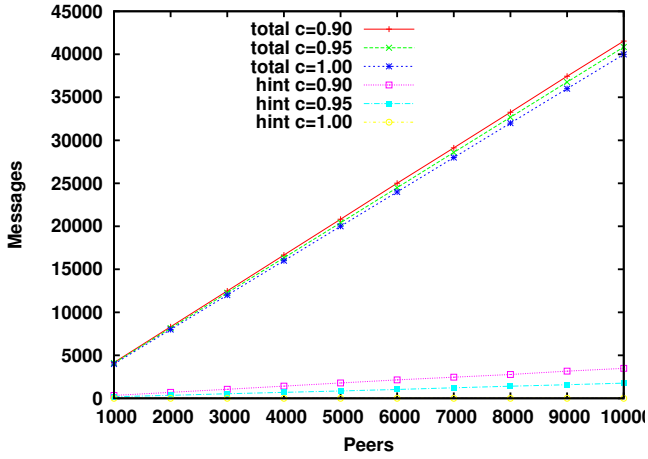


Figure 5.4: Number of messages generated by the relaxed-ring maintenance. Three curves labeled *total* represent the total amount of messages exchanged between all peers depending on the connectivity coefficient. Curves labeled *hint* represent the contribution of *hint* messages to the total amount.

5.4 Comparison with Chord

As we previously mentioned, we have also implemented Chord in our simulator CiNiSMO. 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. The idea is to respect the assumptions made by Chord authors as much as possible. We make our comparison considering two parameters: lookup consistency and bandwidth consumption.

5.4.1 Lookup Consistency

Even though the connectivity conditions of the experiments running Chord were much better than those of the relaxed-ring, 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 when only 4 nodes joined the ring in between every periodic stabilization. The amount of nodes joining the ring during that period is what we call stabilization rate. As seen in figure 5.5, 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-

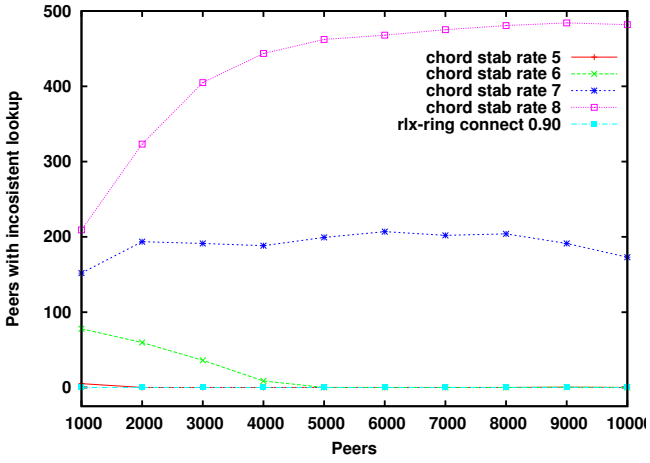


Figure 5.5: Amount of peers with overlapping ranges of responsibilities, introducing lookup inconsistencies, on Chord networks under different stabilization rates for different network sizes. Comparison with the Relaxed-Ring with a bad connectivity. The stabilization rate represent the amount of peers 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.

ring (connectivity factor 0.9) where no inconsistencies where found. In this experiment, every instance of a Chord network of a given size was run with six different random number generators. What it is shown in the graph is the average of those instances.

5.4.2 Bandwidth consumption

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 5.6 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 5.6 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

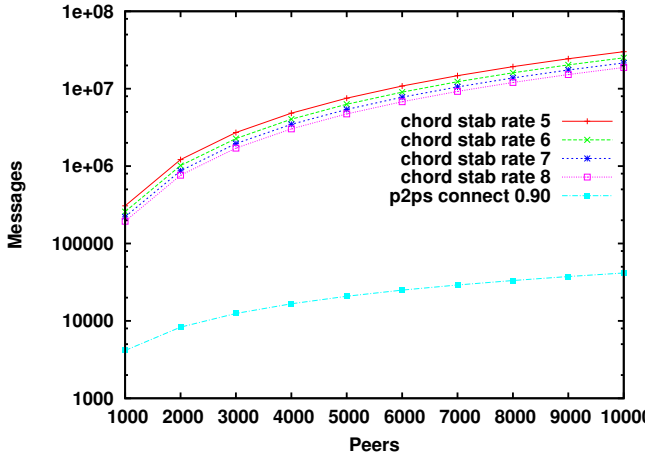


Figure 5.6: Load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

are only triggered when they are needed.

5.5 Efficiency of the Routing Table

This section presents an analysis of the results obtained by simulating PALTA, the self-adaptable finger table we described in Section 3.5. This finger table uses a full mesh graph when the estimated size of the network is smaller than a given value ω. When this threshold is reached, the finger table of new peers will only work with DKS fingers. If network's size decrements back below the value of ω, all peers with adapt their finger tables to get closer to a full mesh, taking more advantage of the connectivity.

To validate this finger table strategy, we simulate with two different values of ω, and we compare them with the Relaxed-Ring using plain DKS, and with fully connected network, which always have full mesh connectivity. In order to measure the efficient use of resources, we have measured the average amount of active connection a node has in every of these networks. To study the performance of the topologies, we have measured the total amount of messages needed to build the network, and the average hops needed to perform a lookup.

Every topology is tested by building networks from 20 to 1000 nodes, increasing the size by 20 nodes at every iteration. Plotted values represent the average of running every experiment with several seeds for random number generation. In the case of PALTA, we tested the algorithm using two different values for ω, being 100 and 200. Reaching 1000 nodes might be considered not large enough for large scale networks, but it is enough to observe the

behavior after the ω threshold is reached and extrapolate the scalability from the curves obtained.

5.5.1 Active connections

One of the goals of PALTA is to dynamically adapt its topology in order to optimize the use of the network. For small networks that means that we want to directly connect as much peers as possible, in order to reach every peer in the minimum amount of hops. Small is defined in terms of the ω value.

Figure 5.7 shows the average amount of active connections per peer in the different topologies. We can observe that the *fully connected* network increments the amount of connections linearly, and therefore, it does not scale at all. Part of the curve is missing, but it clearly corresponds to $n-1$, being n the size of the network, because every node is connected to all the other peers. As expected, the *relaxed-ring* with plain DKS appears as the topology where peers manage the smallest amount of connections, showing that it has good scalability for large networks. Let us analyze now the behavior of PALTA. In both cases, with ω 100 and 200, we observe that the amount of connections increases linearly as a fully connected network until reaching ω peers. From that point on, the average of connections decreases very fast, converging asymptotically to the values of the relaxed-ring. This is because all new nodes that join the network after the threshold of ω is reached, create only the amount of fingers needed by a relaxed-ring. In fact, ω peers manage $\omega - 1$ connections, and $N - \omega$ peers manage k fingers, with N being the size of the network. Meaning that the larger the network, the smallest the average. Of course, this decreasing behavior continuous until it almost reaches the curve of the relaxed-ring, then, the average can only increasing according to the size of the network.

In conclusion, Figure 5.7 shows us that PALTA uses actively more resources than a regular ring, but it is capable of self-adapting when the network becomes too large and provide a good scalability.

5.5.2 Network traffic

When peers enter in a distributed network, they generate a number of messages in order to correctly join without leading the network to an unstable state. In the case of a *fully connected* network, the joining peer will always need $2 * n$ messages to contact all peers in a network of size n. Therefore, the cost of a new joining peer increases as the size of the network increases. In our simulation we contact directly every peer. In case a broadcast mechanism is used to propagate the *join* of a new peer, n messages are needed to reach every peer, plus n message to acknowledge the new peer, making $2 * n$ messages. These measurements about fully connected network do not apply

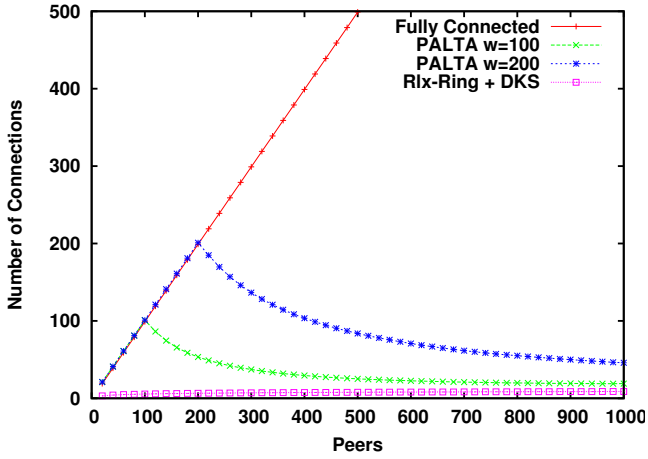


Figure 5.7: Average amount of active connections vs number of peers. This chart evaluates if a small network take advantage the proximity by opening more networks, and it validates the scalability of the solution.

to small networks using Bluetooth, where *one hoop* broadcasting is available. On the other hand, Bluetooth cannot currently handle the amount of connection we are testing. They actually belong to a different problem domain.

In the Relaxed-Ring with plain DKS, the joining peer needs to send messages for contacting the predecessor, successor and the k fingers. Therefore, the marginal cost of a joining peer is almost independent of the size of the network. The only difference occurs with the amount of messages needed for localizing the k finger, which increases logarithmically with respect to the size of the network, as we will see in Section 5.5.3.

Figure 5.8 does not show the marginal cost of joining a network, but the total amount of messages generated to construct every network we have studied in section 5.5.1. We can see that with less active connections, as in PALTA or the relaxed-ring, the number of messages remains small, generating less network traffic. The curve of the fully connected network increases quadratically, generating $n * (n - 1)$ messages, with n being the size of the network, we can conclude that this network cannot scale.

The curve of the relaxed-ring with DKS shows a constant and controlled increment in the amount of messages, keeping them at a very low rate, showing that it scales very well. Now, the results obtained from experiments with PALTA are very interesting because both perform better than the ring for larger networks. One can observe that PALTA with $\omega = 100$ and $\omega = 200$ increases quadratically the amount of messages, as in a fully connected network. This happens only until the network reaches a size of ω peers. Then, the amount of messages increases slower that in a ring, and furthermore, after

a certain size of the network, both PALTA networks remain at better values that the relaxed-ring. The explanation for this is that when a new peer join in the network, it needs less messages to find the k fingers. This is because PALTA has ω peers with a larger routing table ($\omega > k$), making a more efficient jump during the routing process. We study this further in the following section.

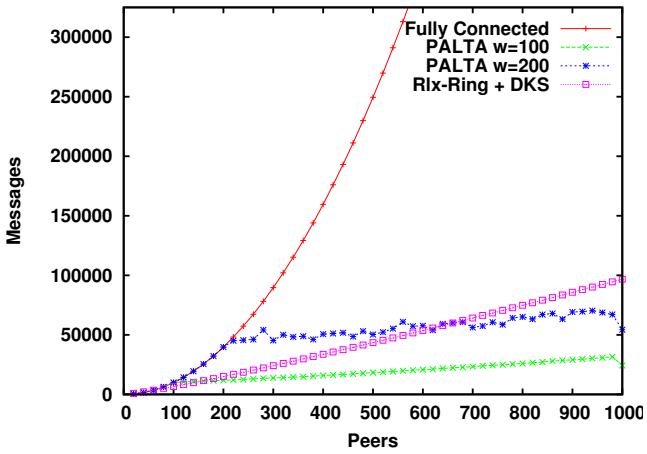


Figure 5.8: Total amount of messages to build the network *vs* number of peers. This chart evaluates how costly is every network. It is interesting to see that PALTA networks start by behaving as a fully connected network, but then they are more efficient than the relaxed-ring with DKS. This is explained in the next chart.

This means that the cost of maintaining a small fully connected network can help a larger network to be more efficient for routing, generating less network traffic. We observe that PALTA could not only be used for ambient intelligent networks as it was in their original conception, but also as the topology for large scale systems. This is why we have actually adopted for the implementation of Beernet.

5.5.3 Hops

In order to confirm our conclusions from the previous experiment, we decided to measure the average amount of hops needed for a message to reach its destination. This is known as a *lookup operation* in a ring. This experiment does not consider fully connected networks, because there is no concept of responsibility is such systems. In addition, because of its characteristics, peers in a fully connected network reach any other peer in the network in 1 hop.

In Figure 5.9 we can observe the results obtained. The relaxed-ring with DKS fingers shows that the number of needed hops increase logarithmically

when the network size increases. PALTA performs better than the relaxed-ring due to fact that some peers have a larger routing table, confirming the results from the previous experiment. In both cases, PALTA presents an average number of hops slightly smaller than 2 if the network consist of less than ω peers. This is because the network is fully connected, and therefore, in can reach the predecessor of the responsible of the looked up key in only one hop. The second hop is needed to reach the responsible. The average is smaller than 2 because the randomized experiments sometimes generates lookups where the responsible is the peer triggering the lookup. This section includes figure 5.9

After the value of ω is reached, the average increases faster in PALTA with $\omega = 100$ that with $\omega = 200$. This is clearly due to the amount of peers having a larger routing table. We observe that in both cases the system behaves much better than the ring. We expect that for larger networks the value would converge to the curve of the ring, but still performing better. What we cannot currently explain is the behavior of PALTA with $\omega = 100$ when the network is in between 100 and 200 nodes. It seems to perform even better than a $\omega = 200$.

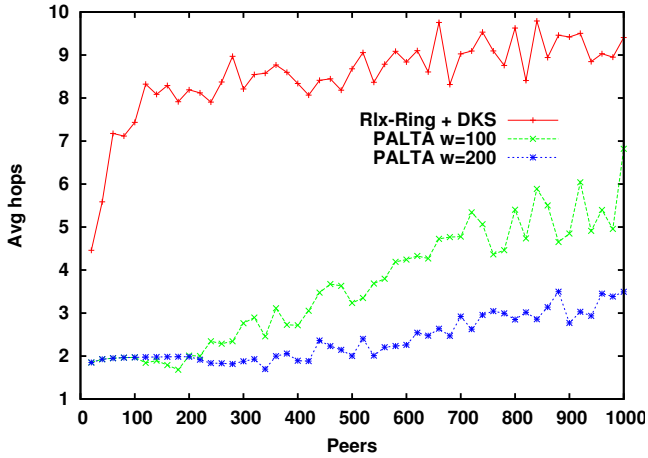


Figure 5.9: Average number of hops vs number of peers. The relaxed-ring with DKS fingers follows a logarithmic distribution in the amount of hops that are needed to reach any peer in the network. PALTA network converge to this value for large networks, but they remain lower because there are some peers with more connections than the average peer in the network, providing a more significant jump to complete the lookup request.

Something that we still need to investigate is the construction of a network where every peer define its own ω vale according to its own resources. We want to PALTA in ambient intelligent network formed by heterogeneous devices, each one with its own resources.

5.6 Conclusion

This chapter has allowed us to validate our claims presented in Chapters 3 and 4. We have shown that the relaxed-ring presents much less lookup inconsistencies compared to Chord with different stabilization rates. This result is observed even when the connectivity conditions of the experiment we used for the relaxed-ring where much worse that the conditions of Chord experiments. We also observe that ring maintenance is much efficient in the relaxed-ring thanks to the correction-on-change strategy.

The price that has to be paid to not introduce inconsistencies in the relaxed-ring, is the degradation of the routing complexity by adding the size of a branch to the $O(logN)$ achieved by Chord. The experimental result has shown us that the average size of a branch is less that two peers, and that the impact on the whole network is actually less than 0.5. This means that the degradation is minimal.

We have also seen that PALTA strategy to provide a self-adaptable finger table can be efficient in two directions. It makes a more efficient use of the connectivity in small networks, and it reduces the amount of hops to resolve lookups in larger network. This is thank to some peers that manage more peers that a regular DKS finger table, making more significant jumps to route messages.

Chapter 6

Transactional DHT

The most basic operations provided by a DHT are `put(key value)` and `get(key)`. We have seen in Chapter 2 that this is not enough to provide fault tolerance, and that a replication strategy should be used in order to guarantee data storage. Replicas are not simple to maintain independently of the chosen replication strategy. Therefore, it is very convenient to add transactional support to the DHT so as to manage the replicas, and to provide atomic operations over a set of items.

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 6.1. 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 will see how transactional support based on Paxos consensus [MH07, GL06] works well in decentralized systems. 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.

We extends the Paxos consensus algorithm with an eager locking mechanism, so as to fit the requirements we identify in synchronous collaborative applications. A notification layer is also added to the transactional layer support, which can be used by any of the transactional protocols we will describe. In this chapter we also make an analysis of replica maintenance, and how it is related to the transactional layer.

6.1 Two-Phace Commit

The pseudo-code in Algorithm 10 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. Since the operations have different semantics, as we will see in section 6.6, it is justified to use different keywords.

Algorithm 10 Swap transaction

```
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. Figure 6.1 depicts 2PC protocol showing two possible execution. The diagrams do not include the client, but they concentrate on the interaction between the TM and the TPs. Figure 6.1(a) shows a successful execution of the protocol where the TPs get the confirmation of the TM about the result of the transaction. Figure 6.1(b) spots the main problem of this protocol. If the TM crashes after collecting the locks of the TPs, the TPs remained locked forever if the algorithm is *crash-stop*. PostgreSQL [Pos09] implements 2PC as a *crash-recovery* algorithm, meaning that the TM can reboot and recover the state before the crash to continue with the protocol.

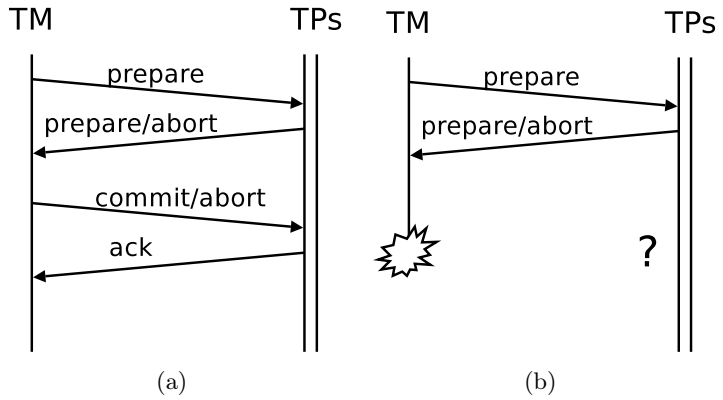


Figure 6.1: Two Phase Commit protocol (a) reaching termination and (b) not knowing how to continue or unlock the replicas because of the failure of the transaction manager.

Discussing with PostgreSQL developers, they have said that a transaction could hang for a whole weekend before the locks are released again. This kind of behaviour is not feasible in peer-to-peer networks when there is no certainty that a peer that leaves the network will ever come back.

6.2 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 [MH07] based on Paxos consensus [GL06].

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 6.2 describes how the Paxos-consensus protocol works. The client, which is connected to a peer that is part of the network, triggers a transaction in order to read/write some items from the global store. When the transaction begins, the peer becomes the transaction manager (TM) for that particular transaction. The whole transaction is divided in two phases: *read phase* and *commit phase*. During the *read phase*, the TM contact all transaction participants (TPs) for all the items involved in the transaction. TPs

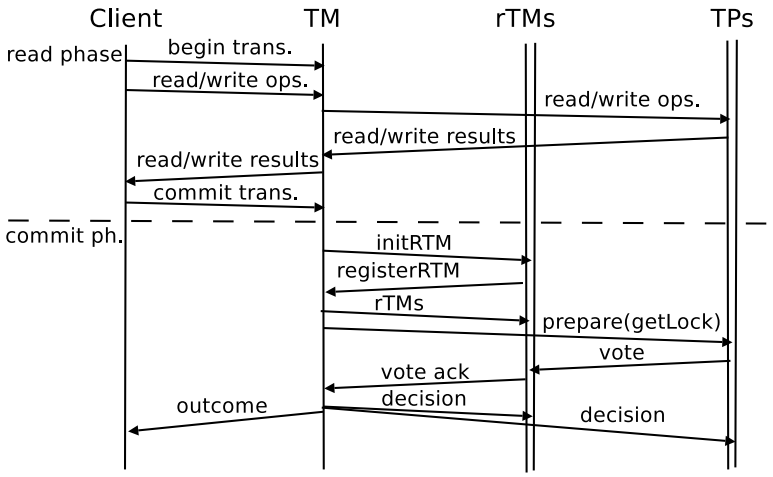


Figure 6.2: Paxos consensus atomic commit on a DHT.

are chosen from the peers holding a replica of the items. The modification to the data is done optimistically without requesting any lock yet. Once all the read/write operations are done, and the client decides to commit the transaction, the *commit phase* is started.

In order to commit the changes on the replicas, it is necessary to get the lock of the majority of TPs for all items. But, before requesting the locks, it is necessary to register a set of replicated transaction managers (rTMs) that are able to carry on the transaction in case that the TM crashes. The idea is to avoid locking TPs forever. Once the rTMs are registered, the TM sends a *prepare* message to all participants. This is equivalent to request the lock of the item. The TPs answer back with a *vote* to all TMs (arrow to TM removed for legibility). The vote is acknowledged by all rTMs to the leader TM. Like that, the TM will be able to take a decision if the majority of rTMs have enough information to take exactly the same decision. If the TM crashes at this point, another rTM can take over the transaction. The decision will be *commit* if the majority of TPs voted for commit. It will be *abort* otherwise. Once the decision is received by the TPs, locks are released.

The protocol provides atomic commit on all replicas with fault tolerance on the transaction manager and the participants. As long as the majority of TMs and TPs survives the process, the transaction will correctly finish. These are very strong properties that will allow the development of collaborative applications on a decentralized system without depending on a server.

6.3 Paxos with Eager Locking

We have observed how Paxos consensus algorithm for atomic transactions on DHTs is extremely useful for building systems with decentralized storage

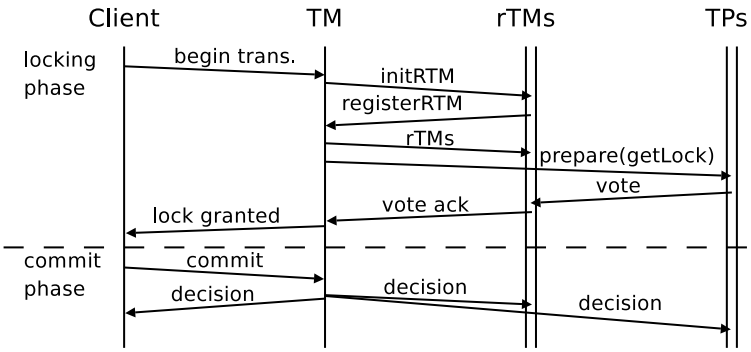


Figure 6.3: Paxos consensus with eager locking and notification to the readers.

based on symmetric replication. The protocol works very well for applications such as Wikipedia on Scalaris [SSR08, PRS07] or the recommendation system Sindaca, presented in Section 8.1. These systems are designed to support asynchronous collaboration between application's users. The fact that Paxos consensus protocol works with optimistic locking fits well asynchronous collaboration. However, this locking strategy limits the functionality of synchronous collaborative applications such as DeTransDraw, a collaborative drawing tool that we will describe in Section 8.2.

DeTransDraw has a shared drawing area where users actively make updates and observe the changes made by other users. If two users make modifications to the same object at the same time, at the end of the their work, when they decide to commit, only one of them will get her changes committed, and the other one will loose everything. Because users are working synchronously, the probability that this happens is much larger than in applications such as Wikipedia or Sindaca. This is why a pessimistic approach with eager locking is needed.

We have adapted Paxos to support eager locking adding a notification mechanism for the registered readers of every shared item. We have implemented this new protocol in Trappist, the transaction layer support of Beernet, with the possibility of dynamically choosing between the two Paxos protocols. Like that, the application can decide the protocol to be used depending on the functionality that is provided.

Figure 6.3 depicts the adapted protocol with eager locking. The read-phase and commit-phase from the original protocol has been replaced by *locking-phase* and *commit-phase*. The read phase disappears because the transaction manager tries eagerly to get the needed to lock to proceed with the transaction. Once the locks are collected, the client is informed of the result. The goal is to prevent users from trying to start working on items that are already locked. The client of the transaction starts working on the changes on the items as soon as the transaction begins. Starting to work on

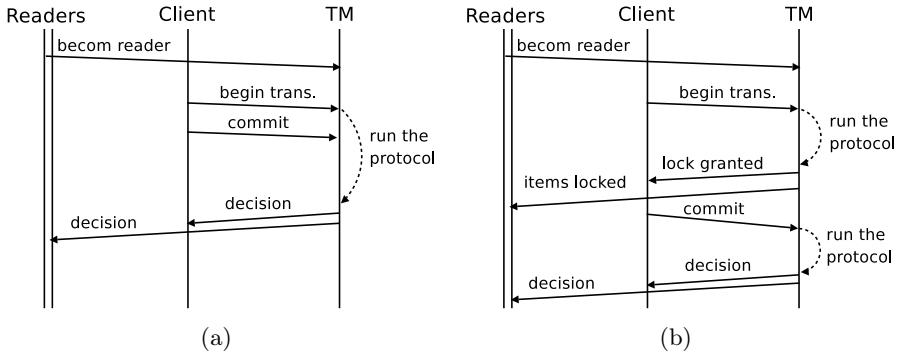


Figure 6.4: Notification layer protocol. Peers register to each item by *becoming readers*. Figure (a) shows one notification for the decision if the transaction is run with Paxos. Figure(b) shows notifications for locking and decision, if transaction is run with Eager Paxos.

an item is actually the trigger of the transaction.

When the user stops making modifications, it triggers the commit-phase. The transaction manager can take the decision immediately because the majority of the votes have been already collected at this stage. The decision is propagated to the client, the replicated transaction managers and transaction participants, as in the original Paxos algorithm. Because there is no read-phase, it is important that the decision is transmitted to the TPs and rTMs together with the new state of the item, and not only a *commit/abort* message.

6.4 Notification Layer

The modification with eager locking provides notification to the readers every time an item is locked and updated. Sometimes it is not necessary to get a notification on locking, and only the update is important. In such case, it is interesting to have a layer of notification independent of the protocol used to update the item. This kind of feature is useful to implement applications such an online score board, where only a few peers modify the state of the application, and many peers participate as readers. For the readers is not necessary to get a notification that some value is currently being update. They just need to get the last value of the item.

The layer consist on a reliable multicast that sends a notification to all subscribed readers of an item. In order to make the multicast efficient, if the amount of readers is smaller than $log(N)$, a direct message sending can be performed. If the amount is larger, the update message can be transmitted using the multicast layer of the peer-to-peer network.

6.5 Replica management

During the description of the transactional protocols, we have assumed that transaction participants are members of the replica set of each item, and that they are chosen by an underlying layer corresponding to the replica management. This layer needs to keep a consistent set of replicas even under churn. Systems such as Scalaris [SSR08] and DKS [Gho06] consider that the replica layer is completely orthogonal to the transactional layer. We understand it differently, because we see that restoring a lost replica needs the transactional support in order to know how to retrieve the latest value from the surviving replicas.

Let us analyse more deeply replica management. The replica set of each item is form with f replicas. One of the problems we can encounter is that $f + 1$ peers claim to hold a valid replica. There are a couple of things to consider here. First of all, why is it a problem to have $f + 1$ replicas? The problem is that you could potentially have two majorities with respect to f. In Beernet, as in Scalaris, we take f as an even number. Like that, you will need $f + 2$ replicas so as to have two majorities. With $f + 1$ is not enough to break majority if f is an even number.

Second question is, how can the system end up having $f + 1$ replicas? The first possible cause is lookup inconsistency: two nodes thinking that they have to store a replica. Reducing the amount of lookup inconsistencies is a way of addressing this issue, and this is what we do with the Relaxed-Ring. The second way of having $f + 1$ replicas is churn. The only problem actually comes when there is churn affecting the responsibility of one of the transaction participants. In that case, following the more detail description of Paxos consensus algorithm [MH07], the items involved in the transaction are actually locked, and they are not transfered to the new responsible until the transaction has finished.

One suggestion to avoid two majorities of replicas is to add group management to the replica sets. This idea can become too expensive to implement and maintain because there is a replica set per each stored item. It is true that because of symmetric replication many replica sets overlap, but there is no guarantee on this property so as to define a cost-efficient group maintenance. We consider that there is no need for group management on symmetric replication. Peers do not store references to every peer in the replica set of every item or replicated item is stored in the peer. It would be too expensive. This is actually one of the advantages of symmetric replication: when a peer needs to find the other replicas, it uses the symmetric function and lookup requests to identify the other members of the set.

Let us consider that there is group management for the replica set in order to make the replica layer completely orthogonal to the transactional layer. When there is churn, the peer that takes over a range of responsibility of one of the members of the replica set, needs to read from the majority

of the replicas in order to decide the value of the items it is going to host. Therefore, some transaction is still needed anyway when a new peer *"joins the replica set"*. We discuss now two possible ways of getting a new peer in the replica set: a new peer joins the network, and a peer fails and it is replace by the recovery mechanism.

6.5.1 New peer joins the network

Symmetric replication was introduced in [Gho06], without discussing any transactional support. To maintain the replica set, the new joining peer should ask its successor for the values of the items its storing. Note that the new peer replace its successor as member of the replica set of a certain amount of items. Asking the successor is fine, but it has the problem that relies on the fact that all replicas are up to date. Of course, if the successor does not have the latest value, it does not introduce real problems because it would replace a bad replica for another bad replica. Not a good replica for a bad one. Still, it might be a good idea to retrieve the value of the item from the replica set in order to replace a bad replica by a good one. However, performing read from majority every time a new peer joins the network is expensive.

6.5.2 Failure handling

Whenever there is failure, DKS [Gho06] makes use of the *startbulkown* operator which is in charge of finding the owner of a replicated key, and retrieve the values from that peer. We believe that here we arrive to a similar analysis of the joining peer: is it enough to retrieve the items from only one node? In the case of the failure recovery it is more important to read from the majority, because the recovery node cannot know if the dead peer was holding the latest value or not. Let us say the replica set is form by peers a, b, c, d and e, where a, b, and e have the last up to date value of item i. Therefore, c and d hold and old value of i. Let us suppose now that peer e dies, and f takes over its responsibility. Peer f reads i from c or d, and then, the system ends up having a majority c, d and f holding an old value of i, which is incorrect. This problem can already be improved by choosing an even amount of replicas, but if the read is done from the majority, it does not matter if the replication factor is odd or even.

6.6 Trappist

As we have previously mentioned in this chapter, the transactional layer implementing these three protocol is called Trappist, which stands for Transactions over peer-to-peer with isolation, where isolation means that transactions are atomic and with concurrency control. In this section we show how

to use the transactional support of Beernet, which is implemented with the Mozart [Moz08] programming system. By describing Trappist's API, we also analyse the high level abstractions provided by the system, and how replica maintenance is hidden from the programmer. We stat by creating a Beernet peer. Currently, nodes in Beernet are created by default with transactional support. However, to prevent conflicts with previous versions we explicitly flag transactions to be included in the following example:

```
functor
import
   Pbeer at 'Pbeer.ozf'
define
   Node = {Pbeer.new args(transactions:true)}
   ...
```

The most basic support provided by Beernet corresponds to the DHT operations *put* and *get*. This operations do not replicated the value of the item, but they are also part of the implementation of the transactional layer which actually realizes the replication. What follows is an example of how put and get can be used.

```
{Node put(key value)}
Value = {Node get(key $)}
```

To use the transactional layer, the user must write a procedure with one argument, typically named Obj. This argument represents a transactional object, which is an instance of the transaction manager that triggers the transaction. The object receives the operations *read* and *write*, which are almost equivalent to *put* and *get*. The main semantic difference between the operations is that if the transaction is aborted, *write* has no effect on the stored data. And if the transaction succeeds, the value is written at least on the majority of the replicas. Other operations received by the transactional object are *commit* and *abort*, to explicitly trigger those actions on the protocol. The operation *remove* is also implemented in order to delete an item from the DHT.

To run the transaction, user must invoke the method $executeTransaction$, which receives three arguments. The procedure containing the operations, a port to receive the outcome of the transaction, and the protocol to be used for running the transaction. Note that at the creation of the node, we did not specify the protocol to be use by every transaction. This is because the protocol can be chosen dynamically, allowing the users to choose the best suitable protocol for every functionality. Algorithm 11 is a complete example for writing two items with key/value pairs: $hello/\text{“}Charlotte''$ and foo/bar. The outcome of the transaction appears on variable $Stream$, which is the output of port $Client$. If the outcome of the transaction is `commit`, it guarantees that both items where successfully stored at least in the majority of the correspondent replicas.

Algorithm 11 Using transactions with Paxos consensus to write two items

```
declare
Stream Client
Trans = proc {$ Obj}
        {Obj write(hello "Charlotte")}
        {Obj write(foo bar)}
        {Obj commit}
      end
{NewPort Stream Client}
{Node executeTransaction(Trans Client paxos)}
if Stream.1 == commit then
   {Browse "transaction succeeded"}
end
```

To retrieve the values the user passes a variable which has no value yet. The value is bound by the transactional object. Algorithm 12 shows how to retrieve the values stored under keys *hello* and foo.

Note that it is not necessary to catch exceptions using Beernet, because the outcome is reported on the stream of the client's port. If there is a failure on the transaction, the outcome will be `abort`, and the user will be able to take the corresponding failure recovery action. If the item is not found, the variable used to retrieve the value is bound to a *failed value*. This language abstraction will raise an exception whenever is used. Like this, exceptions are triggered in the calling site, and not at any of the peers. Now, to prevent catching exceptions when using the value, the Mozart programming system provides boolean checkers to test whether a variable is bound to a failed value or not.

Algorithm 12 Using transactions with Paxos consensus to read two items

```
declare
V1 V2
Trans2 = proc {$ Obj}
        {Obj read(hello V1)}
        {Obj read(foo V2)}
       end
{Node executeTransaction(Trans2 Client paxos)}
{Browse "for hello I got"#V1}
{Browse "for foo I got"#V2}
```

6.7 Conclusion

In this chapter we have described the transactional protocols that are used to build Trappist, the transactional layer of Beernet. Trappist is built on top of the Relaxed-Ring and it uses symmetric replication, however, it is independent of the overlay graphs and the replication strategy that is in use. But, it influences how the replication layer handles replica maintenance. We have reviewed Two-Phase commit, which is one of the most classical protocols for transactions on distributed systems. We also explained why 2PC is not suitable for peer-to-peer and decentralized applications in general. By introducing Paxos consensus algorithm we described a solution to the limitations of 2PC. Paxos introduces replicated transaction managers and the concept of majority to improve fault-tolerance and still remain efficient.

Apart from the analysis and validation of these protocols, our contribution is the extension of Paxos consensus algorithm by adding an eager locking protocol, and a notification layer that helps the development of synchronous collaborative applications. During this chapter we also discussed replica management techniques, and how it is affected by the transaction layer. We finally described Trappist's API to analyse its design as high level abstract to use transactional DHTs.

Chapter 7

Beernet's Implementation

Beernet stands for pbeer-to-pbeer network, where words peer and beer are mixed to emphasise the fact that this is a peer-to-peer network built on top of a *relaxed*-ring topology, considering that beers are usually a mean to achieve relaxation. This chapter describes Beernet's implementation. We will review its architecture and design decisions putting more attention to some of the components. First, we will need to review some general concepts on concurrent and distributed programming so as to understand the design decision we have taken. This chapter is written not only in the context of self-management and decentralized systems, but also with an interest for programming language abstraction and software engineering.

7.1 Distributed Programming and Partial Failure

The key issue in distributed programming is partial failure. It is what makes distributed programing different from concurrent programming. This unavoidable property causes uncertainty because we cannot know whether a remote entity is ever going to reply to a message. It is also the reason why remote procedure call and remote method invocation (RPC and RMI) are difficult to use. In "A note on Distribution" [WWWK94], four main concerns on distributed programming are discussed: *latency*, *memory access*, *concurrency* and *partial failure*. Latency is not a critical problem because it does not change the semantics of performing an operation on a local or a distributed entity. It just makes things go a bit slower. Memory access is solved by using a virtual machine that abstracts the access, and then it does not change the operational semantics either. A more difficult problem is concurrency. The middleware has to guarantee exclusive access to the state in order to avoid race conditions. There are different techniques such as data-flow, monitors or locks, that makes possible the synchronization between processes achieving a coherent state. Given that, and even though it

is not trivial to write concurrent programs correctly, it is not a critical problem either. What really breaks transparency is partial failure. Basically, distribution transparency works as long as there is no failure.

A partial failure occurs when one component of the distributed system fails and the others continue working. The failure can involve a process or a link connecting processes, and the detection of such a failure is a very difficult task. In distributed environments such as the Internet, it is impossible to build a perfect failure detector because when a process p stops responding, another process p' cannot distinguish if the problem is caused by a failure on the link connecting process p or the crash of the process p itself. This explanation might be trivial, but it is usually forgotten. Failures are a reality on distributed systems, this is why we consider the definition of a distributed system given by Leslie Lamport very accurate:

> "A distributed system is one in which the failure of a computer you did not even know it existed can render your own computer unusable"

Even though this definition does not describe the possibilities of a distributed system, it makes explicit why distributed computing is special. It is very important to know how the system handles the failure of part of the system. We have already discussed this concept in the motivation of this dissertation. We have used the concept along the analysis of the state of the art, and we have use it in the design of the relaxed-ring and the transactional layer.

The classical view of distributed computing sees partial failure as an error. For instance, a remote method invocation (RMI) on a failed object raises an *exception*. This approach actually goes against distribution transparency, as it is explained in [GGV05] because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another less fundamental issue but still relevant, is that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives.

New trends in distributed computing, such as ambient intelligence and peer-to-peer networks, see partial failure as an *inherent characteristic* of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. We believe that this approach leads to more realistic language abstractions to build distributed systems. We believe that the most convenient mechanism to develop peer-to-peer applications effectively is by using *active objects* that communicate via *asynchronous message*

passing. These active object are very similar to actors [AMST97] . We also use *fault streams* per distributed entity in order to perform failure handling. In this chapter we show that this works better than the usual approach of using RMI. We define our peers as lightweight actors and we use them to build a highly dynamic peer-to-peer network that deals well with partial failure and non-transitive connectivity. Our model is influenced by the programming languages Oz [Moz08, VH04] and Erlang [Arm96], and by the algorithms of the book "Introduction to Reliable Distributed Programming" [GR06], which we already introduced in Chapter 3 to describe our algorithms for the Relaxed-Ring. In the following section we describe the model more in detail, focusing also in their component architecture.

7.2 Event-driven Components

The algorithms for reliable distributed programming presented in [GR06] are designed in terms of components that communicate through events. Every component has its own state, which is encapsulated, and every event is handled in a mutually exclusive way. The model avoids shared-state concurrency because the state of a component is modified by only one event at the time.

Every component provides a specific functionality such as point-to-point communication, failure detection, best effort broadcast, and so forth. Components are organized in layers where the level of the abstraction is organized bottom-up. A higher-level abstraction requests a functionality from a more basic component by triggering an event (sending a request). Once the request is resolved, an indication event is sent back to the abstraction (sending back a reply). Algorithm 13 is taken from the book, where only the syntax has been slightly modified. It implements a best-effort broadcast using a more basic component, (*pp2p*), which provides a perfect point-to-point link to communicate with other processes.

Algorithm 13 Best Effort Broadcast

```
upon event ⟨ bebBroadcast | m ⟩ do
    for all p doother_peers
        trigger ⟨ pp2pSend | p, m ⟩
    end
end

upon event ⟨ pp2pDeliver | p, m ⟩ do
    trigger ⟨ bebDeliver | p, m ⟩
end
```

The best-effort broadcast (*beb*) component handles two events: *bebBroadcast* as requested from the upper layer, and *pp2pDeliver* as an indication coming from the lower layer. Every time a component requests *beb* to broadcast a

message m, *beb* traverses its list of other peers, triggering the *pp2pSend* event to send the message m to every peer p. Every p is a remote reference, but it is the *pp2p* component which takes care of the distributed communication. At every receiving peer, the *pp2p* component triggers *pp2pDeliver* upon the reception of a message. When *beb* handles this event, it triggers *bebDeliver* to the upper layer, as seen in Algorithm 13. It is important to mention that *beb* does not have to wait for *pp2p* every time it triggers *pp2pSend*, and that *pp2p* does not wait for *beb* or any other component when it triggers *pp2pDeliver*. This asynchronous communication between components means that each component can potentially run in its own independent thread.

Using layers of components allows programmers to deal with issues concerning distribution only at the lowest layers. For instance, the component *beb* is conceived only with the goal of providing a broadcast primitive. The problem of communicating with a remote processes through a point to point communication channel is solved in *pp2p*. If a process p crashes while the message is being sent, it does not affect the code of *beb*, thus improving the transparency of the component. There is no need to use something like

$$\textbf{try } \langle send\ m\ to\ p \rangle \textbf{ catch } \langle failure \rangle$$

It is the responsibility of *pp2p* to deal with the failure of p. It is also possible that *pp2p* triggers the detection of the crash of p to the higher level, and then it is up to *beb* to do something with it, for instance, removing p from the list of other peers to contact. In such a case, the failure of p is considered as part of the normal behaviour of the system, and not as an exception. Even though the code for the maintenance of *other_peers* set is not given [GR06], we can deduce it from the implementation of the other components. In Algorithm 14 the *register* event is a request made from the upper layer, and *crash* is an indication coming from the *pp2p* layer.

Algorithm 14 Best Effort Broadcast extended

```
upon event ⟨ bebRegister | p ⟩ do
    other_peers := other_peers ∪ {p}
end

upon event ⟨ crash | p ⟩ do
    other_peers := other_peers \ {p}
end
```

Even though we advocate defining algorithms using event-driven components using the approach of [GR06], there are some important drawbacks to consider. To compose layers, it is necessary to create a channel, connect the components using the channel, and subscribe them to the events they will handle. We find this approach a bit over sophisticated. It could be simplified by talking directly to a component and using a default listener only when necessary. A related problem concerns the naming convention of events. The

name reflects the component implementing the behaviour, making the code less *composable*. For instance, if we want to use a fair-loss point-to-point link (*flp2p*) instead of *pp2p*, we would have to change the *beb* code by replacing *pp2pSend* by *flp2pSend*, and instead of handling *pp2pDeliver* we would have to handle *flp2pDeliver*.

Since the architecture considers components and channels, an alternative and equivalent approach would be to use objects with explicit triggering of events as method invocation, instead of using anonymous channels. Using objects as collaborators, they could be replaced without problems as long as they implement the same interface. In such an approach, both *flp2p* and *pp2p* would handle the event *send* and trigger *deliver*.

The other problem of [GR06] is that there is no explanation of how to transfer a message from one process to the other. The more basic component *flp2p* is only specified in terms of the properties it holds, but it is not implemented. There is no language abstraction to send a message to a remote entity.

7.3 Event-driven Actors

As we have previously mentioned during this dissertation, we have implemented Beernet in Mozart/Oz [Moz08]. Mozart is an implementation of the Oz language, which is a multi-paradigm programming language supporting functional, concurrent, object-oriented, logic and constraint programming paradigms [VH04]. It offers support for distributed programming with a high degree of transparency. Thanks to the multi-paradigm support of Oz, we were able use more convenient language abstractions for distribution and local computing while building Beernet. In this section we discuss the basic language abstractions that we considered appropriate and necessary to implement event-driven components. In addition, we discuss the abstractions that allowed us to improve the approach towards an event-driven actor model, that we also call active objects.

7.3.1 Threads and data-flow variables

One of the strengths of the Oz language is its concurrency model which is easily extended to distribution. The kernel language is based on procedural statements and single-assignment variables. When a variable is declared, it has no value yet, and when it is bound to a value, it cannot change the value. Attempting to perform an operation that needs the value of such a variable will wait if the variable has no value yet. In a single-threaded program, that situation will block forever. In a multi-threaded program, such a variable is very useful to synchronize threads. We call it a data-flow variable. Oz provides lightweight threads running inside one operating system process with a fair thread scheduler.

The code in Algorithm 15 shows a very simple example of data-flow synchronization. First, we declare variables *Foo* and *Bar* in the main thread of execution. Then, a new thread is created to bind variable *Bar* depending on the value of *Foo*. Since the value of *Foo* is unknown, the '+' operation waits. A second thread is created which binds variable *Foo* to an integer. At this point, the first thread can continue its execution because the value of *Foo* is known.

Algorithm 15 Threads and data-flow synchronization

```
declare Foo Bar
thread Bar = Foo + 1 end
thread Foo = 42 end
```

This synchronization mechanism does not need any lock, monitor, or semaphore, because there is no explicit state, and therefore, no risk for race conditions. The values of *Foo* and *Bar* will be the same for all possible execution orders of the threads. Single-assignment variables are also used in languages such as E [MTSL05] and AmbientTalk [DVM+06, VMG+07], where they are called promises or futures. They are combined with the *when* operator as one of the mechanisms for synchronization.

The execution of a concurrent program working only with single-assignment variables is completely deterministic. While this is an advantage for correctness (race conditions are impossible), it is too restrictive for general-purpose distributed programming. For instance, it is impossible to implement a server talking to two different clients. To overcome this limitation, Oz introduces Ports, which are described in the following section.

7.3.2 Ports and asynchronous send

A port is a language entity that receives messages and serializes them into an output stream. After creating a port, one variable is bound to the identity of the port. That variable is used to send asynchronous messages to the port. A second variable is bound to the stream of the port, and it is used to read the messages sent to the port. The stream is just like a list in Lisp or Scheme, a concatenation of a head with a tail, where the tail is another list. The list terminates in an unbound single-assignment variable. Whenever a message is sent to the port, this variable is bound to a dotted pair containing the message and a fresh variable.

Algorithm 16 combines ports with threads. First we declare variables P and S. Then, variable P is bound to a port having S as its receiving stream. A thread is created with a *for-loop* that traverses the whole stream S. If there is no value on the stream, the *for-loop* simply waits. As soon as a message arrives on the stream, it is shown on the output console. A second

thread is created to traverse a list of beers (*BeerList*, declared somewhere else), and to send every beer as a message to port P. This is a like a barman communicating with a client. Everybody who knows P can send a message to it, as in the third thread, where the list of sandwiches is being traversed and sent to the same port. Beers will appear on the stream in the same order they are sent. Beers and sandwiches will be merged in the stream of the port depending on the order of arrival, so the order is not deterministic between them.

Algorithm 16 Port and asynchronous message passing

```
declare P S
P = {NewPort S}
thread
   for Msg in S do
      {Show Msg}
   end
end
thread
   for Beer in BeerList do
      {Send P Beer}
   end
end
thread
   for Sdwch in SandwichList do
      {Send P Sdwch}
   end
end
```

The send operation is completely asynchronous. It does not have to wait until the message appears on the stream in order to continue with the next instruction. The actual message send could therefore take an arbitrary finite time, making it suitable for distributed communication where latency is an issue. With the introduction of ports, it is already possible to build a multi-agent system running in a single process where every agent runs on its own lightweight thread. The non-determinism introduced with ports allows us to work with explicit state, and there is no restriction on the communication between agents.

7.3.3 Going distributed

Event though full distribution transparency is impossible to achieve because of partial failures, there is some degree of transparency that is feasible and useful. Ports and asynchronous message passing as they are described in the previous section can be used transparently in a distributed system. The semantics of `{Send P Msg}` is exactly the same if P is a port in the same

process or in a remote peer. In both cases the operation returns immediately without waiting until the message is handled by the port. If there is a need for synchronization, the message can contain an unbound variable as a future. Then, the sending peer waits for the variable to get a value, which happens when the receiving peer binds the variable. This implies that the variable, and whatever is contained in the message, is transparently sent to the other peer. Variable binding must therefore be transparent.

Algorithm 17 does a ping-pong between two different peers. The first lines of the code represent peer A who sends a ping message to peer B. The message contains an unbound variable *Ack*, which is bound by peer B to the value `pong`. Binding variable *Ack* resumes the *Wait* operator at peer A. Peer B, code below peer A and indented at the right, makes a pattern matching of every received message with pattern `ping(A)`. If that is the case, it binds A to `pong` and continues with the next message. The pattern matching is useful to implement a method dispatcher as we will see in the next section.

Algorithm 17 Ping-Pong

```
\% at Peer A
declare Ack
{Send PeerB ping(Ack)}
{Wait Ack}
{Show "message received"}
```

```
\% at Peer B
for Msg in Stream do
   case Msg of ping(A) then
      A = pong
   end
end
```

This sort of transparency is not difficult to achieve, except when a partial failure occurs. An older release of Mozart, version 1.3.0, takes the classical approach to deal with partial failures: it raises an exception whenever an operation is attempted on a broken distributed reference. Most programming languages take the same approach. This approach has two important disadvantages. First of all, it is cumbersome because it is necessary to add **try** ... **catch** instructions whenever an operation is attempted on a remote entity. More fundamentally, exceptions break transparency when reusing code meant for local ports. If a *distribution* exception is raised, it will not be caught because the code was not expecting that sort of exception.

AmbientTalk [DVM+06, VMG+07] adopts a better approach. In ambient-oriented programming, failures due to temporary disconnections are a very common thing, therefore, no exception is raised if a message is sent to a disconnected remote reference. The message is kept until the connection is

restored and the message is resent. Otherwise if the connection cannot be fixed after a certain time, it will be garbage collected. Failures are also a common thing in peer-to-peer networks. The normal behaviour of a peer is to leave the network after some time. Therefore, a partial failure should not be considered as an exceptional situation.

A more recent Mozart release, version 1.4.0, does not raise exceptions when distributed references are broken. It simply suspends the operation until the connection is reestablished or the entity is killed. If the operation needs the value of the entity, for instance in a binding, the thread blocks its execution. If a send operation is performed on a broken port, because of its asynchrony, it still returns immediately, but the actual sending of the message is suspended until the connection is reestablished. This failure handling model [CV06, Col07] is based on a *fault stream* that is attached to every distributed entity [MCPV05, KMV06]. An entity can be in three states, *ok*, *tempFail*, or *permFail*. Once it reaches the permanent failure state, it cannot come back to *ok*, so the entity can be killed. If the entity is in temporary failure for too long, it can be explicitly killed by the application and forced to *permFail*. To monitor an entity's fault stream, the idea is to do it in a different thread that does not block and that can take actions over the thread blocking on a failed entity.

7.3.4 Actors

The actor model [AMST97] provides a nice way of organizing concurrent programming, benefiting from encapsulation and polymorphism in analogous fashion to object-oriented programming. We extend the previous language abstractions with Oz *cells* which are containers for mutable state. State is modified with operator ':=', and it can be read with operator '@'. We do not need to add new language abstractions in order to build our event-driven actors. Without language support, actors are a programming pattern in Oz as is shown in Algorithm 18. Having ports, the cell is not strictly necessary but we use it to facilitate state manipulation. Every actor runs in its own lightweight thread and communicates asynchronously with other actors through ports. Encapsulation of state is achieved with lexical scoping, and exclusive access to state to avoid race conditions is guaranteed by handling only one event/message at a time.

Algorithm 18 is a working implementation of Algorithms 13 and 14 using the language abstractions we have described in this section. It is written in Oz without syntactic support for actors but the semantics are equivalent. The function *NewBestEffortBroadcast* creates a closure containing the state of the actor and its behaviour. The state includes a list of *OtherPeers* and another actor implementing perfect point-to-point communication, which is named *ComLayer* to make explicit that it could be replaced by any actor that understands event *send*, and not only *pp2p*.

The behaviour is implemented as a set of procedures where the signature of the event is specified in each procedure's argument. For instance, the declaration on code line 9 reads that procedure *Receive* implements the behaviour to handle **upon event** *deliver(Src Msg)*. The variable *Listener* represents the actor in the upper layer.

Algorithm 18 Beernet Best Effort Broadcast

```
fun {NewBestEffortBroadcast Listener}
   OtherPeers ComLayer
   SelfPort SelfStream
   proc {Broadcast broadcast(Msg)}
      for Peer in OtherPeers do
         {Send ComLayer send(Peer Msg)}
      end
   end

   proc {Receive deliver(Src Msg)}
      {Send Listener Msg}
   end

   proc {Add register(Peer)}
      OtherPeers := Peer | @OtherPeers
   end

   proc {Crash crash(Peer)}
      OtherPeers := {Remove Peer @OtherPeers}
   end
in
   OtherPeers = {NewCell nil}
   ComLayer = {NewPP2Point SelfPort}
   SelfPort = {NewPort SelfStream}
   thread
      for M in SelfStream do
         case {M.label}
         of broadcast then {Broadcast M}
         [] deliver then {Receive M}
         [] register then {Add M}
         [] crash then {Crash M}
         end
      end
   end
   SelfPort
end
```

Variable *SelfPort* is bound to the port that will receive all messages coming from other actors. A thread is launched to traverse the *SelfStream*. For every message that arrives on the stream, pattern matching checks the label

of the message in order to invoke the corresponding procedure. This part of the code represents the method dispatching of the actor. In the Beernet implementation, the creation of the port and the method dispatching are modularized to avoid code duplication, thus reducing the code size of every actor.

The book [GR06] contains complementary material including a Java implementation of the *beb* component. Discarding comments and import lines, the implementation takes 67 lines of code, with the component infrastructure already abstracted. It is worth mentioning that a large number of lines are dedicated to catch exceptions. Equivalent functionality within the Beernet actor model takes only 33 lines.

7.4 The Ring and the Problems with RMI

The architecture of Beernet is based on layers that abstract the different concepts involved in the construction of the peer-to-peer network. A closely related work is the Kompics component framework [AH08], which follows the component-channel approach of [GR06] using a similar architecture. The main difference with Beernet is that instead of having components that communicate through channels, we decided to use event-driven actors. We will describe more in detail Beernet's architecture in section 7.6. In this section we give more details about the lower layer concerning the overlay maintenance, so as to spot the differences between the Relaxed-Ring and the Chord ring [SMK+01]. This last one being the starting point of many other SONs.

In previous chapters we have explained how a ring-based DHT works. As a summary, peers are organized clockwise in a ring according to their identifiers, forming a circular address space of N hash keys. Every peer joins the network with an identifier. The identifier is used to find the correct predecessor and successor in the ring. When peer q joins in between peers p and s, it means that $p < q < s$ following the ring clockwise. Peer s accepts q as predecessor because it has a better key than p. Another reason to be a better predecessor, is that the current predecessor is detected to have crashed. Hence, the maintenance of the ring involves *join* and *crash* events, and it must be handled locally by every peer in a decentralized way.

In order to keep the ring up to date, Chord performs a periodic stabilization that consists in verifying each successor's predecessor. From the viewpoint of the peer performing the stabilization, if the predecessor of my successor has an identifier between my successor and myself, it means that it is a better successor for me and my successor pointer must be updated. Then, I notify my successor. Algorithm 19 is taken from Chord [SMK+01]. Only the syntax is adapted. The big problem with this algorithm is the instruction

```
x := successor.predecessor
```

Asking for successor's predecessor is done using RMI. This means that the whole execution of the component waits until the RMI is resolved. There is no conflict resolution if *successor* is dead or dies while the RMI is taking place. If there is a partial failure, the algorithm is simply broken.

Algorithm 19 Chord's periodic stabilization

```
upon event ⟨ stabilize | ⟩ do
    x := successor.predecessor
    if x ∈ (self, successor) then
        successor := x
    end
    successor.notify(self)
end
upon event ⟨ notify | src ⟩ do
    if predecessor is nil or src ∈ (predecessor, self) then
        predecessor := src
    end
end
```

An improved version of the stabilization protocol is given in Algorithm 20 using event-driven actors. The representation of a peer is a data structure having *Peer.id* as the integer identifying the peer, and *Peer.port* as the remote reference, being actually an Oz port. The '.' is not an operator over an actor or an object. It is just an access to a local data structure. The '...' in the algorithm hide the state declaration and the method dispatcher loop. The '$<$' operator defines the order in the circular address space. We use it here for simplicity without changing the semantics of the algorithm.

Stabilization starts by sending a message to the successor with an unbound variable X to examine its predecessor. The peer then launches a thread to wait for the variable to have a value, and once the binding is resolved, it sends a message to itself to verify the value of the predecessor. This pattern is equivalent to the *when* abstraction in E [MTSL05] and AmbientTalk [VMG+07]. By launching the thread, the peer can continue handling other events without having to wait for the answer of the remote peer. If the remote peer crashes, the *Wait* will simply block forever without affecting the rest of the computation. When the *Wait* continues, the peer sends a message to itself in order to serialize the access to the state with the handling of other messages. Otherwise there would be a race condition.

Beernet uses a different strategy for ring maintenance as it was explained in Chapter 3. Instead of running a periodic stabilization, it uses a strategy called *correction-on-change*. Peers react immediately when they suspect another peer to have failed. The failed peer is removed from the routing table, and if it happens to be the successor, the peer must contact the next peer in order to fix the ring. To contact the next successor, every peer manages

Algorithm 20 Chord's improved periodic stabilization

```
fun {NewChordPeer Listener}
   ...
   proc {Stab stabilize}
      X
   in
      {Send Succ.port getPredecessor(X)}
      thread
         {Wait X}
         {Send Self.port verifySucc(X)}
      thread
   end
   proc {Verify verifySucc(X)}
      if Self.id < X.id < Succ.id then
         Succ := X
      end
      {Send Succ.port notify(Self)}
   end
   proc {GetPred getPredecessor(X)}
      X = Pred
   end
   proc {Notify notify(Src)}
      if Pred == nil orelse Pred.id < Src.id < Self.id then
         Pred := Src
      end
   end
   ...
end
```

a successor list, which is constantly updated every time a new peer join or if there is a failure.

Algorithm 21 presents part of a *PBeer* actor, which is a Beernet peer. The algorithm has been already presented in Chapter 3 as language independent. In this section we presented how it is really implemented. Failure recovery works as follows: when peer P fails, a low-level actor running a failure detector triggers the $crash(P)$ event to the upper layer, where *PBeer* handles it. *PBeer* adds the crashed peer to the crashed set and removes it from its successor list. If the crashed peer is the current successor, then the first node from the successor list is chosen as the new successor. A *notify* message is sent to the new successor. When a node is notified by its new predecessor, it behaves as a Chord node, but in addition, it replies with the *updSL* message containing its successor list. In this way, the successor list is constantly being maintained.

Algorithm 21 Beernet's failure recovery

```
fun {NewPBeer Listener}
   ...
   proc {Crash crash(Peer)}
      Crashed := Peer | @Crashed
      SuccList := {Remove Peer @SuccList}
      if P == @Succ then
         Succ := {GetFirst SuccList}
         {Send Succ.port notify(Self)}
      end
   end
   proc {Notify notify(Src)}
      if {Member Pred @Crashed}
      orelse Pred.id < Src.id < Self.id then
         Pred := Src
      end
      {Send Src.port updSL(Self @SuccList)}
   end
   ...
end
```

7.5 Fault streams for failure handling

As described at the end of subsection 7.3.3, we use a *fault stream* associated to every distributed entity in order to handle failures. An operation performed on a broken entity does not raise any exception, but it blocks until the failure is fixed or the thread is garbage collected. This blocking behaviour is compatible with asynchronous communication with remote entities. In the fault stream model, presented by Collet et al [CV06, Col07],

the idea is that the status of a remote entity is monitored in a different thread. The monitoring thread can take decisions about the broken entity, in order to terminate the blocking thread. For instance, there are language abstractions to kill a broken entity so it can be garbage collected.

Algorithm 22 describes how we use the fault stream in the implementation of Beernet. There is an actor in charge of monitoring distributed entities called *FailureDetector*. Upon event *monitor(Peer)*, the actor uses the system operation *GetFaultStream* in order to get access to the status of the remote peer. The fault stream is updated automatically by the Mozart system, which sends heartbeat messages to the remote entity in order to determine its state. When the state changes, the new state appears on the fault stream. If the connection is working, the state is set to *ok*. If the remote entity does not acknowledge a heartbeat, it is suspected of having failed, and therefore, the state is set to *tempFail*. Since Internet failure detectors cannot be strongly accurate, the state can switch between *tempFail* and *ok* indefinitely. As soon as the state is set to *permFail*, however, the entity cannot recover from that state.

If the state is *tempFail* or *permFail*, the actor triggers the event *crash(Peer)* to the *Listener*, which represents the upper layer. If the state switches back to *ok*, the event *alive(Peer)* is triggered. It is up to the upper layer to decide what to do with the peer. In the case of Beernet, this is described in algorithm 21.

Algorithm 22 Fault stream for failure detection

```
fun {FailureDetector Listener}
   ...
   proc {Monitor monitor(Peer)}
      FaultStream = {GetFaultStream Peer}
   in
      for State in FaultStream do
         case State
         of tempFail then {Send Listener crash(Peer)}
         [] permFail then {Send Listener crash(Peer)}
         [] ok then {Send Listener alive(Peer)}
         end
      end
   end
   ...
end
```

7.6 General Architecture

Now that we have reviewed the fundamental concepts that allowed us to implements our components as event-driven actors, we review the general architecture of Beernet. In this section we use the words actors and component indifferently. We have mentioned that Beernet is globally organized as a set of layers providing higher level abstract with a bottom-up approach. Figure 7.1 gives the global picture of how actors are organized.

The bottom layer is the *Network* component. This actor is composed by four other actors. The most basic communication is provided by perfect point-to-point link (*Pp2p link*) that simply connects two ports. The *Peer-to-peer link* allows a simpler way of sending messages to a peer using its global representation, instead of extracting the port explicitly every time a message is to be sent. *Peer-to-peer link* uses *Pp2p link*. Network uses two failure detectors: one provided by the Mozart, and the other one implemented in Beernet itself. The *Mozart failure detector* is the one described in Algorithm 22, taking advantage of the fault-stream of every distributed entity. *Beernet failure detector* is built as a self-tunning failure detector that uses its own protocol to change the frequency and timeout values of the keep alive messages. Both failure detectors are eventually perfect, meaning that they are strongly complete, and eventually accurate.

The Relaxed-Ring component uses the Network component to exchange messages between directly connected peers, and to detect their failures. It has two main components: the *Relaxed-Ring maintenance* and the *Finger Table*. The relaxed-ring maintenance runs essentially the protocols we have described in Chapter 3. The finger table is in charge of efficiently routing messages that are not sent neither to the successor nor the predecessor of a node. The finger table actor can implement several of the routing strategies we discussed in Chapter 2, as long as it is consistent with the relaxed-ring topology. For Beernet, we have decided to implement fingers as they are described in Section 3.5, using PALTA strategy, which combines DKS [AEABH03] fingers with self-adaptable behaviour to improve efficiency. This actor is also in charge of monitoring the messages in order to provide correction-on-use of the routing table. Algorithm 23 shows that when a message arrives to the relaxed-ring maintenance, it first verify is the message is for the self node, for any of branches (backward) or for its successor. If none of the cases is valid, it delegates the event to the $FingerTable$ component.

The reliable message-sending layer is implemented on top of the relaxed-ring maintenance. This layer includes the basic services *Realiable Send*, *Multicast* and *Broadcast*. Each of them is running on its own actor, but they can collaborate if necessary, as the relaxed-ring maintenance collaborate with the finger table. The basic *DHT* with its *put* and *get* operations is implemented on top of the messaging services.

In Beernet, we have decided to implement the transactional layer, Trap-

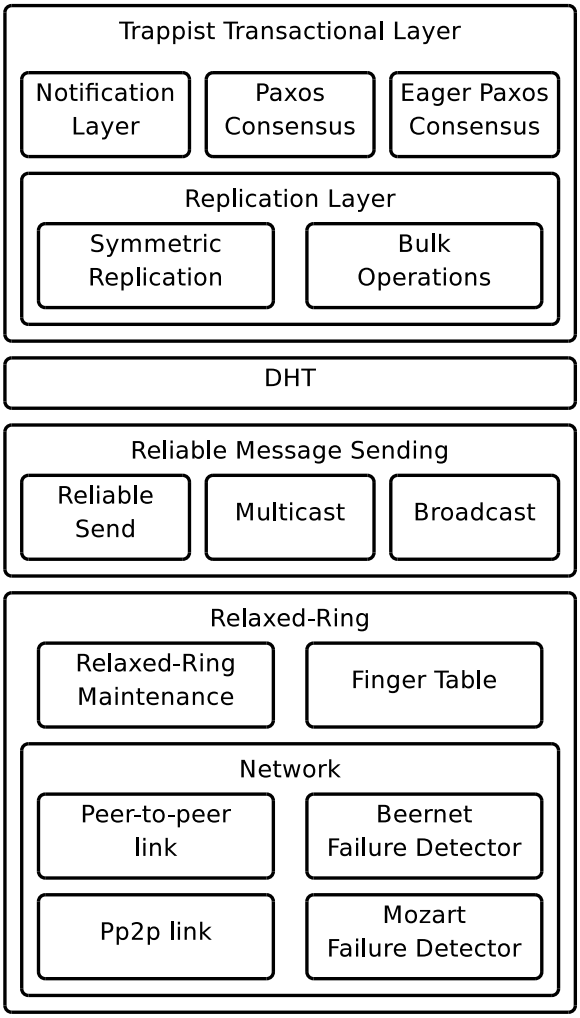


Figure 7.1: Beernet's actor architecture. Every component run on its own lightweight thread and they all communicate asynchronously through message passing.

Algorithm 23 Messages in the relaxed-ring maintenance component are routed to direct neighbours or delegated to the Finger Table

```
proc {Route Event}
  route(msg:Msg src:_ target:Target ...) = Event
in
  if {BelongsTo Target @Pred.id @SelfRef.id} then
    %% This message is for me
    {Send Self Msg}
  elseif {HasFeature Event last} andthen Event.last then
    %% Backward to the branches
    {Backward Event Target}
  elseif {BelongsTo Event.src.id @SelfRef.id @Succ.id} then
    %% I think my successor is the responsible
    {Send @Succ {Record.adjoinAt Event last true}}
  else
    %% Forward the message using the Finger table
    {Send @FingerTable Event}
  end
end
```

pist, having the replication layer as part of the Trappist component. This is a major difference with Scalaris [SSR08], because they present their architecture having replication and transaction as two independent layers. We claim that replication needs the transactional mechanism in order to restore replicas in case of failures. Let us suppose a set of six replicas i, j, k, l, m and n. where the majority i, j, k and l, holds the latest value of the replicated item. If k fails and it is replaced by k', how does k' knows from where to read in order to restore the replica? If it only reads from m or n, the majority is broken. If it reads from all replica set is doing unnecessary work, because it only needs to read from the majority. Having the knowledge of the protocol that is used to manage the replica is the best option to keep replica maintenance efficient. That is why in Beernet the replica maintenance also belongs to the transactional layer instead of being an independent component.

There are still some orthogonal components within the replica management that can be changed by equivalent ones. For instance, we have chosen to work with *Symmetric replication* instead of successor list replication, or leaf set replication. To reach the replicas, the transactional layer will use the *Bulk operations* [Gho06] which can be written for any replication strategy.

The Trappist layer includes three different protocols to provide transactional support: *Paxos Consensus*, *Eager Paxos Consensus* and *Two-Phase Commit*. The three of them are explained in detail in Chapter 6. Two-phase commit is not included in Figure 7.1 because its used is not recommended for building applications. We have implemented it for purely academic purposes. These protocols are enriched by a *Notification Layer* component that

contributes to develop more synchronous collaborative applications.

7.7 Discussion

One of the principles we respect in this paper is to *avoid shared-state concurrency*. We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for data-flow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives of lightweight threads and ports are also present in Erlang [Arm96], and they are not specific to object-oriented programming. Single-assignment variables also appear in E [MTSL05] and AmbientTalk [VMG+07] in the form of promises, and they are meant for synchronization of remote processes instead of lightweight threads.

The actor model presented here through programming patterns is further developed and supported by E and AmbientTalk. There is one important difference related to the use of lightweight threads. Since they are not supported by these two languages, there is basically only one actor running per process. The actor collaborates with a set of passive objects within the same process. Communication with local objects is done with synchronous method invocation. Communication with other actors, and therefore with remote references, is done with asynchronous message passing. This distinction reduces transparency for the programmer because it establishes two types of objects: local and distributed.

In Beernet, we organize the system in terms of actors only, making no distinction in the send operation between a local and a remote port. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an actor, and only one send operation.

As mentioned in the previous section, Kompics [AH08] is closely related because it is also a component framework conceived for the implementation of peer-to-peer networks. Instead of using actors for composition, it uses event-driven components which communicate through channels, analogous to events in [GR06].

7.8 Conclusion

We have presented examples in this chapter to highlight the importance of partial failure in distributed programming. The fact that failures cannot be avoided has a direct impact on the goal of transparent distribution which cannot be fully achieved. Therefore, it has also an impact on remote method invocation, the most common language abstraction to work with distributed objects. Because of partial failure, it is very difficult to make RMI work

correctly. In other words, RMI is considered harmful. Therefore, we have implemented Beernet where communication between processes and components is done with asynchronous message passing.

Even though full transparency in distributed programming cannot be achieved, it is important to provide some degree of transparency. We have shown how *port* references and the *send* operation can be used transparently. This is because send works asynchronously and because a broken distributed reference does not raise an exception in Mozart 1.4.0. Instead, a fault stream associated to every remote entity provides monitoring facilities.

We have also described the language abstractions we use to implement Beernet. We have chosen an actor model based on lightweight threads, ports, asynchronous message passing, single-assignment variables and lexical scoping. We have reviewed the general structured of Beernet and we have shown how components interact with each other.

Chapter 8

Applications

This chapter describe four applications designed and implemented using Beernet: Sindaca, DeTransDraw, a decentralized wiki and twitbeer. The first one, Sindaca, uses intensively the transactional DHT layer Trappist. DeTransDraw benefits from the Eager locking protocol in order to provide synchronized collaboration. The other two applications are designed and developed by other researchers and by students. We present them here to show the impact of the contribution of this dissertation.

8.1 Sindaca

This section presents the design and functionality of our community-driven recommendation system named Sindaca, which stands for Sharing Idols N Discussing About Common Addictions. The name spots the main functionality of this application which is making recommendations on music, videos, text and other cultural expressions. It is not designed for file sharing to avoid legal issues with copyright. It allows users to provide links to official sources of titles. Users get notifications about new suggestions, and they can vote on the suggestions to express their preferences. It is expected that users built communities based on their common taste. The system is implemented on top of Beernet [Pro09], presented in Chapter 7. The data of the system is symmetrically replicated on the network using the transactional layer for decentralized storage management, Trappist, presented in Chapter 6.

We have implemented a web interface to have access to Sindaca. All requests done through the web interface are transmitted to a peer in the network which triggers the corresponding operations on the peer-to-peer network. The results are transmitted back to the web server, which presents the information in HTML format as in any web page. Using a web interface to transmit information between the end user and the peer-to-peer network has been used previously in different projects. A very related one is the

peer-to-peer version of the Wikipedia using Scalaris [PRS07, SSR08]. We have extended this architecture with a notification layer which allows eager information updates. This layer is also used in the DeTransDraw application, as we will see in section 8.2. However, this eager notification feature is not provided on the web interface.

To generalize the similitudes and differences between Sindaca and the above mentioned applications, we can say the following: the Wikipedia on Scalaris uses *optimistic* transactions using the Paxos consensus algorithm. DeTransDraw uses *pessimistic eager-locking* transactions using Paxos consensus algorithm with a *notification layer*. Sindaca is a combination of theses strategies. It uses *optimistic* transactions with Paxos extended with the *notification layer*, both implemented in Trappist.

Sindaca is available for demo testing at url:

`http://beernet.info.ucl.ac.be/sindaca`

Use the following login information to enter the system:

- Username: fbrood
- Password: sindaca

Any modifications done by the tester user will be stored in the network, but they are not persistent to the reinitialization of the network. In case of problems during the test, please check contact information on the web page.

Figure 8.1 shows Sindaca's welcome page with the sign in form on the left of the page, together with the menu. The screenshot shows user *fbrood* logging in.

8.1.1 After sign-in and voting

If username and password are successfully provided, the user is taken to the profile page where information about the recommendations stored in the system is displayed. Figure 8.2 is a screenshot of the web page displayed after user *fbrood* has signed in. There is a welcome message both on the menu and on the center of the content. What follows is a list of recommendations suggested by members of the Sindaca community. This recommendation could have been made by other members or by the user itself. The recommendation is composed by a title, the name of the artist, and a link where the title can be found. As mentioned before, Sindaca does not provide storage for content preventing legal issues with copyright.

The listed recommendations are only those that has not received a vote from the user. A radiobutton is provided to express the preference which goes from no good to good. We have actually chosen a scale from *No beer* to *Beer*. The votes are submitted to the network when the user press the `Vote` button. Once the voting submission is sent, a transaction is triggered to modify the item that stores the recommendation. There are more items involved in this transaction, but the details will be explained in section 8.1.3.

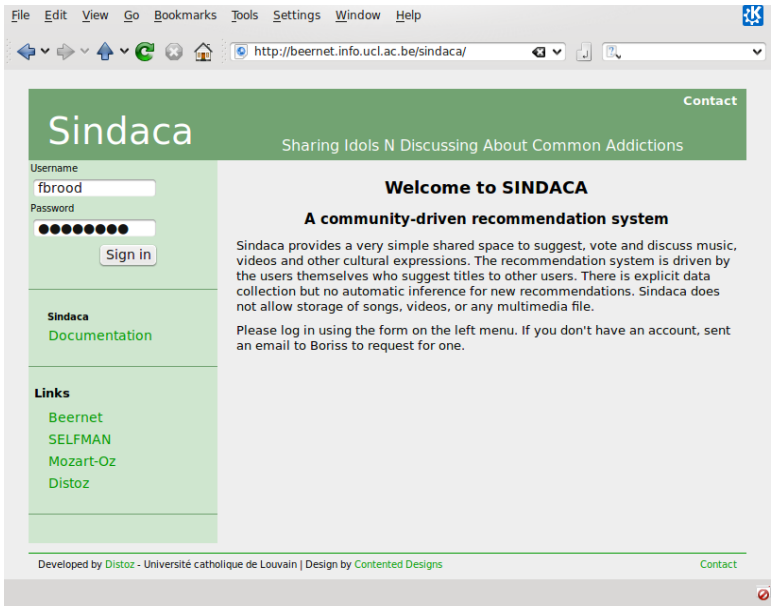


Figure 8.1: Sindaca's welcome page with sign in form.

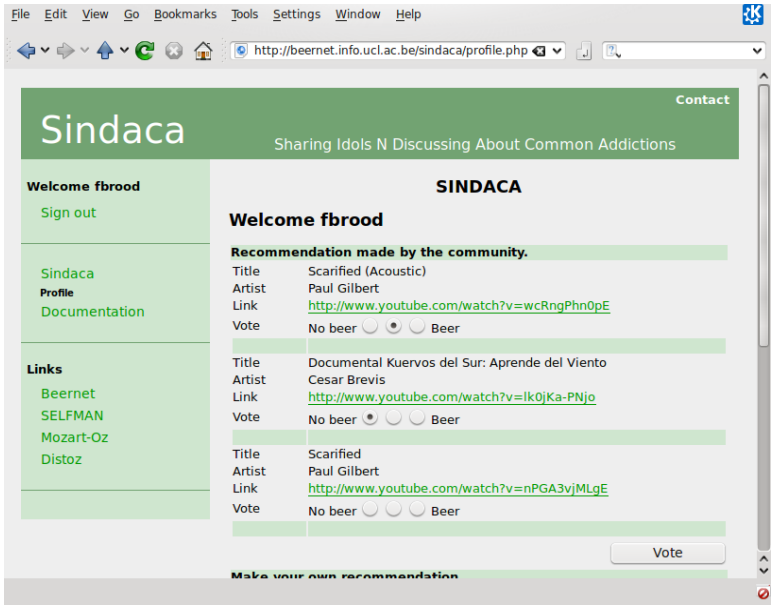


Figure 8.2: After sign in, users can vote for suggested recommendations.

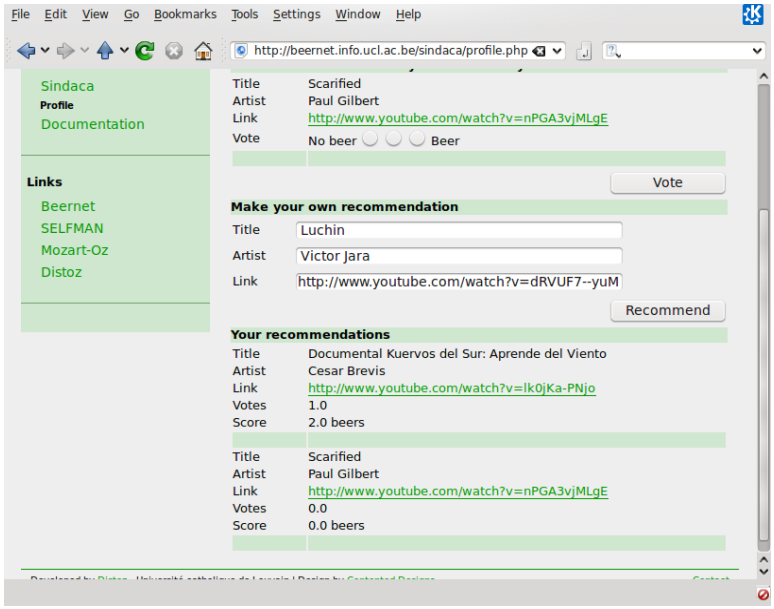


Figure 8.3: Adding a new recommendation.

8.1.2 Making a recommendation

The form to add a new recommendation is presented in the same page where the recommendations to be voted are displayed. The form can be seen in Figure 8.3 where the data for a new recommendation is already completed. The user must fill in the title, author, and link to the title. Once the data is submitted by clicking the button `Recommend`, a new item will be created in the network storing the recommendation. This item will be associated to the user that creates it.

Every user has a list of recommendations she has made. This list is displayed in the same profile page, below the form for adding new recommendations. Therefore, the full profile page displays from top to bottom: welcome message, list of recommendations to be voted, form to add a new recommendation, and the list of recommendations already made by the user. Figure 8.4 shows how the last list is presented. Apart from the above mentioned fields, namely title, artist and link, the information contains two other fields being part of the state of every recommendation: the amount of votes, and the average score of the title. The screenshot we display in Figure 8.4 was taken after the addition of the recommendation made in Figure 8.3. For that item we can observe that no vote is registered, and therefore there is no average score either.

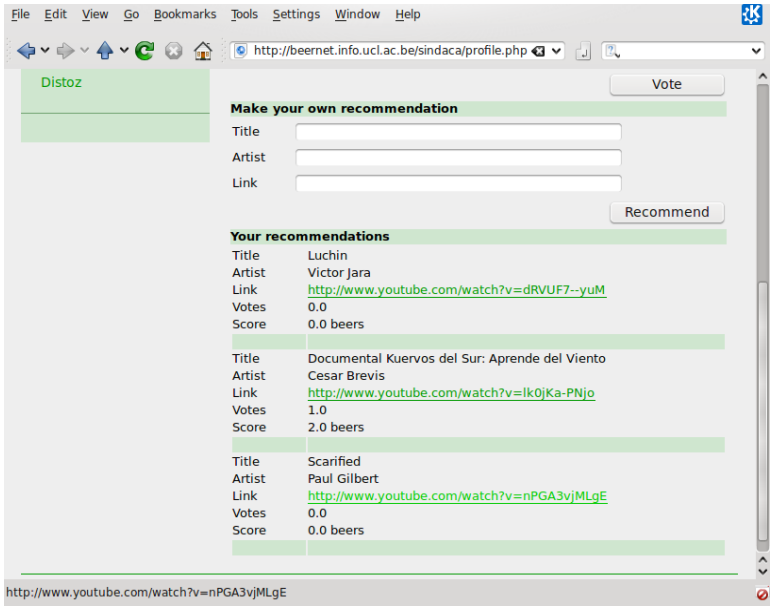


Figure 8.4: State of recommendations proposed by the user.

8.1.3 Design and Implementation

In the previous sections we described the functionalities of Sindaca, and we briefly introduced the effects of every action in the storage of the network. This section is dedicated to explain more about the details on the design and implementation of Sindaca.

First of all, it is important to remark that Sindaca is not implemented on top of a database supporting SQL queries. Sindaca is implemented on top of a transactional distributed hash table with symmetrically replicated state. Therefore, the basic unit for storage is the key-value pair, which is what it is called *item*. The information of every user is stored as one item. The *value* of such item is a record with the basic information: user's id, username and password. We have chosen a very minimal record to build the prototype, but the value can potentially store any data such as user's real name, contact information, age, description, etc. The *key* of the item is an *Oz name* [Moz08], which is unique and unforgeable, acting as a capability reference [MS03]. This strategy provides us certain level of security, because only programs that are able to map usernames with their capability can have access to the key, and therefore, access to the item. The username-capability mapping is only available to programs holding the corresponding capability to the mapping table.

The functionality of adding a new recommendation, shown in Figure 8.3, makes it clear that a recommendation belongs to a user. Therefore, every user item contains a list of capabilities which are references to recommen-

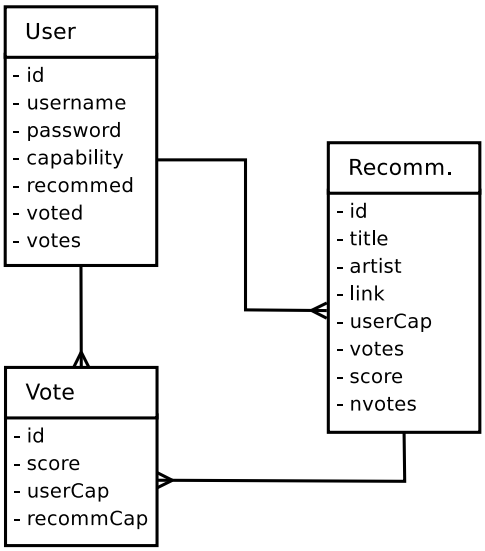


Figure 8.5: Sindaca's relational model.

dations. The functionality of voting also implies that every user item holds a list of capability references to votes. The relational model is described in Figure 8.5. We observe that a user can have multiple recommendations and multiple votes. What it is also stored in user's item is the list of recommendations already voted. That list will allows us to filter all other recommendations, presenting to the user only those she still have to vote.

From the relational model we can also observe that every recommendation has a list of votes associated to it. Every vote contains information about the score, the user who made the vote, and the voted recommendation. What it is not shown in the relational model is how to find all the items on the network. There are two other items which store the list of all user's keys and all recommendation keys. Every time a new user or recommendation is created, these global items are modified. There is no global item for votes, because votes are accessible through the users and the recommendations.

Creating a user Code 24 shows the transaction to create a user. First of all, it is necessary to read the list of users to verify that the new username is not already in use. This is done by reading the item under key `users`, and verifying if `Username` is a member of it. In such case, the transaction is aborted with the operation `{Obj abort}`. If the transaction continues, we read the item `nextUser` to get a user identifier. Then, we create a new item with the *capability key* `UserCap`. The value of the new item is a record with the fields we described above, and which follows the relational model on Figure 8.5. Afterwards, the value of the `nextUser` item is incremented, and the item with the list of users is also updated.

As we can observe, this transaction consists of six different access to the hast table, which include adding a new item, and modifying two existing ones. It is very important that all writes done to the hash table are committed, and therefore the cost of performing a transaction pays off. Even though the operation seems complicated from the point of view of distribution, we can observe that it looks only as a combination of read/write operations that could actually be local ones.

Algorithm 24 Creating a new user.

```
proc {CreateUser Obj}
     Users UserId UserCap
  in
     UserCap = {NewName}
     {Obj read(users Users)}
     if {IsMember Username Users} then
        UsernameInUse = true
        {Obj abort}
     else
        {Obj read(nextUser UserId)}
        {Obj write(UserCap user(username:Username
                        id:UserId
                        passwd:Passwd
                        cap:UserCap
                        recommed:nil
                        votes:nil
                        voted:nil))}
        {Obj write(nextUser UserId+1)}
        {Obj write(users {AddMember Users Username UserCap})}
        {Obj commit}
     end
  end
```

Committing a vote The most complex transaction is triggered with the voting functionality, and we will explain it using the implementation code as guide. The `PutVote` transaction that performs the needed read/write operations is shown in Code 25. First of all, it is necessary to know the capabilities to access the user that is voting, and the recommendation that is being voted. These are the variables `UserCap` and `RecommCap`, which belong to the scope of `PutVote`. Another variable that belongs to the scope is `Vote`, which contains the value of the vote. A new vote item is created with the correspondent vote capability `VoteCap`. This item connects the vote with the user and the recommendation. Then, the item of the voting user is modified in two fields: the list of voted recommendations is increased with the capability of the voted recommendation, and, the capability of the newly

created vote is added to the list of votes of the user. Three fields of the voted recommendation are affected. The created vote is added to its list of votes, and the vote counter is increased by one. The score average is recomputed taking the new vote into account. Here it is even clearer to observe the sequence of read/write operations performed in the transaction. This is because there is no condition to write the new values, as in Code 24. It is important to clarify that the external functions `Record.adjoin`, `NewScore` and `InsertRecomm`, do not perform any distributed operation. They are just for manipulating values and data structures.

Algorithm 25 Committing a vote on a recommendation

```
proc {PutVote Obj}
   User Recomm VoteCap
in
   VoteCap = {NewName}
   {Obj write(VoteCap vote(score:Vote
                recomm:RecommCap
                user:UserCap))}
   {Obj read(RecommCap Recomm)}
   {Obj write(RecommCap {Record.adjoin Recomm
                recomm(score:{NewScore Recomm Vote}
                      nvotes:Recomm.nvotes+1.0
                      votes:VoteCap|Recomm.votes)})}
   {Obj read(UserCap User)}
   {Obj write(UserCap {Record.adjoin User
                user(votes:VoteCap|User.votes
                   voted:{InsertRecomm Recomm.id
                               RecommCap
                               User.voted}
                )})}
   {Obj commit}
end
```

Create a recommendation We have seen already in the example of voting that each recommendation is stored in an item having a capability as key. This capability is also used in the users item to identify the recommendations associated to a user. Therefore, to create a new recommendation, the application needs to create a new item for it, increment the global counter of recommendations, and add it to the list of recommendations of the user who created it. We will not include the code sample of putting a recommendation because it follows the pattern described by the previous two examples, and it does not contribute anything new to the understanding of the ease of use of Beernet and its Trappist layer for transactional DHT.

Jalisco transactions The code samples presented in Codes 24 and 25 represent single transactions that will be given to a peer to run it on the network. The outcome of the transaction, either *abort* or *commit*, will be sent to a port where the application will decide the next step. When the transaction to create new users aborts because the username is already in use, the application will need to request the new user to choose a different username before attempting to run a new transaction. In the case of creating new recommendation and voting, getting abort as outcome of the transaction only means that there where some conflicting concurrent transactions that committed first. In such case, the transaction can be retried without any modification until it is committed. To simplify the process of retrying, we have implemented the procedure *Jalisco*, which comes from the Mexican expression *"Jalisco never loses" (Jalisco nunca pierde)*. This procedure will simply retry a transaction until it is committed. The code is shown in Code 26.

Algorithm 26 Jalisco transaction retries a transaction until it is committed

```
fun {Jalisco Trans}
   P S
   proc {InsistingLoop S}
      {ThePbeer executeTransaction(Trans P paxos)}
      case S
      of abort|T then
         {InsistingLoop T}
      [] commit|_ then
         commit
      end
   end
in
   {NewPort S P}
   {InsistingLoop S}
end
```

The function creates a port to receive the outcome of the transaction. The `InsistingLoop` executes the transaction on the peer `ThePbeer`, and it waits on the stream of the port to check the outcome of the transaction. If it is abort, it just continues with the loop. If it is commit, it simply return that the transaction has committed.

8.1.4 Configuration

The current version of Sindaca available for demo testing is configured with a peer-to-peer network of 42 nodes. All of the nodes are currently running on the server hosting Beernet's web page. The current state of development is a proof of concepts. We are planing to deploy the service on different machines.

Some initial information is stored in the network in order to bootstrap the network and run the test. This information includes the creation of user *fbrood*, which is available for testing. We have successfully run the service for several weeks allowing other users in our department to test the functionalities of the system by providing their own recommendations and voting on the recommendations of other users. The data provided by the testers is not persistent to the failure of the system. Unfortunately, if the network needs to be restarted, the data provided by the testers is lost.

To transmit the information from the web interface to the network, we have a running Mozart [Moz08] process that listens to the Apache [The09a]-PHP [The09c] service which is reading web requests. This Mozart process connects to a peer in the network in order to trigger the corresponding transaction. The implementation of the peer-to-peer network is done with P2PS/Beernet rev396 [Pro09] or later, which is available for downloading on Beernet's web site, under the download section.

8.2 DeTransDraw

DeTransDraw is a decentralized collaborative vector-based graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

8.2.1 TransDraw

DeTransDraw is a redesign of TransDraw [Gro98], a collaborative drawing tool based on a client-server architecture. We first describe TransDraw to identify its advantages and weakness, and then we explain how DeTransDraw can overcome the problems of its centralized predecessor. TransDraw has a shared drawing area where all users has access to all figures of the drawing. The main contribution of TransDraw is the introduction of transactions to manage conflict resolution between users, and to reduce the problems of network latency. The goal is that a user can manipulate the figures immediately, without waiting for the confirmation of a distributed operation. The transaction manager will solve the conflicts afterwards. A transaction is done in two steps: getting the lock and committing. When the user starts modifying one or more figures it request the corresponding locks. When the user finishes the updates, it performs a commit of the transaction with the new value. However, it is possible that the user loses its modification is another user concurrently modify the figure first.

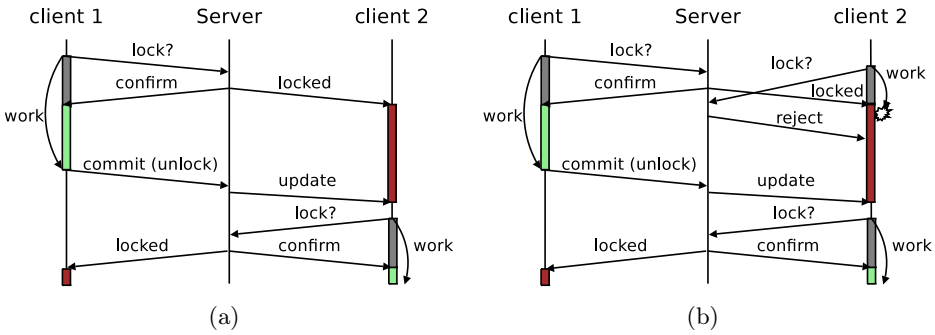


Figure 8.6: Transdraw coordination protocol. In (a) client 1 contacts the server to get the lock and update a figure. Client 2 does another update afterwards. In (b) client 2 get a rejection to its first attempt of acquiring the lock. When figure's lock is released, the client succeeds getting the lock.

Figure 8.6(a) describes the protocol where successful modifications done by two users. Users are represented by `client 1` and `client 2`, which are connected to the `Server`. The server stores the full state of the drawing, and manage the locks of every figure. The server is also the transaction manager of all transactions triggered by the users. The protocol shows that `client 1` starts to work on an object of the drawing and it request its lock. Since the lock is not already taken, the `Server` grants it with a `confirm` message. We can see the bar representing the work of `client 1`. While the bar is gray, it is modifying the figure without knowing of the lock will be granted. The *work-bar* becomes green when confirmation arrives, and it is finished when modifications are done, which results in triggering a `commit` message to the server, including the new value of the drawing object. Note that this message also means that locks are released. Two notifications are sent from `Server` to `client 2`: `locked` and `update`. The locked message prevent `client 2` from modifying that particular object. This is what the red bar represents. Was the new value is committed, the update also reaches `client 2` allowing it for requesting the lock, as it is done as its next step. This new lock is now informed to `client 1`.

Figure 8.6(b) shows the same protocol, with the addition of a failed request from `client 2` to acquire the same locked obtained by `client 1`. What we observe is that `client 2` begins to work on the drawing object resulting on requesting the lock. Its *work-bar* turns from gray to red when the notification of the locked granted to `client 1` arrives. If the notification would have not been sent, the `rejection` notification would have had the same effect. In this case we see that `client 2` has lost some of its work. Note that it is better to be notified earlier, because in other case the transaction would have failed at the end, when all the work was done. This is the main difference between asynchronous and synchronous collaboration.

Asynchronous communication can take an optimistic approach, whereas synchronous collaboration work better with a pessimistic approach, because it is more likely to have a conflict with another user. TransDraw actually merges optimistic and pessimistic approaches. It is optimistic because users can start working immediately even if they do not get the lock. This is essentially to minimize problems associated with latency. And it is pessimistic because it first tries to get the locks, and then it tries to commit. This is to provide a better collaboration between synchronous participants.

8.2.2 TransDraw weakness

Due to its centralized architecture, TransDraw's main weakness is its single point of failure. The server holds the whole state of the application. Other problems are congestion and scalability. Both of them having the same source we just mentioned. The server is the only transaction manager, and therefore, it is a bottle-neck for all the traffic between users. It is a single point of congestion and it does not scale beyond the capacity of the server.

A disputable issue concerns distributed locks. We will not discuss in details how these problems are solved, but we will briefly describe some possibilities to overcome the issues with locks. First of all, the application has decided to keep strong consistency on the state of the drawing, so it is very difficult to come up with a lock-free design. For the case that a client holds a lock for too lock, there is a protocol to explicitly request the client to release the lock. In case the client fails without releasing the lock, the transaction manager can release the lock based on failure detection or time-leasing, to prevent problems with false suspicions of failures. In conclusion, there are workarounds to minimize the problems with distributed locks, but the main issue with TransDraw is scalability and fault-tolerance. This is why a decentralized approach appears as the way to go.

8.2.3 Decentralized TransDraw

We have already discussed the main features and weakness of TransDraw. The aim of DeTransDraw is to provide the same functionality but it removes the server from the design, building the application on top of a peer-to-peer network, making the system fully decentralized. Each transaction runs with its own transaction manager, and the state of the application is spread across the network being symmetrically replicated. This provides not only load balancing but also scalable and fault tolerance.

DeTransDraw is implemented on top of Beernet, and it uses the eager paxos consensus algorithm provided by the transaction layer Trappist. Since Beernet provides a DHT, the drawing information has to be stored in form of items. Each drawing object is an item where its identifier is the key, and the value corresponds to the position, shape, colour, and other properties

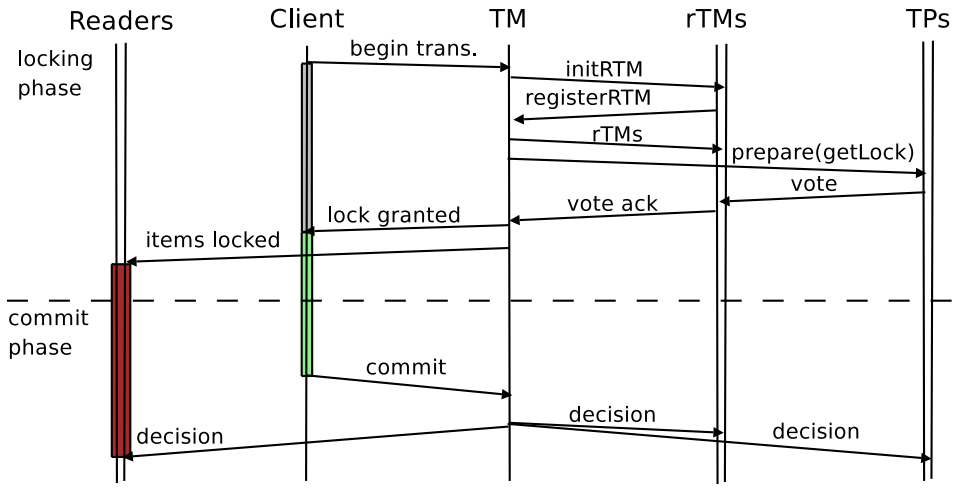


Figure 8.7: DeTransDraw coordination protocol. It combines optimistic and pessimistic approach, using Trappist's eager locking Paxos and the notification layer to propagate the information to the registered readers.

of the figure. The application has been implemented in our research group, being Jérémie Melchior the main developer.

Figure 8.7 shows the protocol we described for TransDraw in Section 8.2.1, but the client contacts a transaction manager (TM) instead of a server. In other words, the server is replaced by the peer-to-peer network. The protocol is an instance of Eager Paxos consensus algorithm, as it is described in Section 6.3, combined with the notification layer that communicates with the readers. In this case, the readers are all the other users of DeTransDraw. We can observe that the `client` performs the same operations as in the protocol of Figure 8.6 but with some different names. Requesting the lock is actually `begin transaction`. Confirmation of acquiring the locks is `locked granted`. The `commit` message is the same in both cases. As we mentioned already, the TM is different for every transaction, and the set of replicated TMs is chosen with the same strategy as symmetric replication. The key to generate the replica set is the one of the TM. The transaction participants (TPs) are all the peers storing a replica of the drawing objects involved in the transaction. Therefore, two concurrent transactions modifying disjoint sets of drawing objects could have completely different sets of TM, rTMs and TPs.

We discuss now the graphical user interface of DeTransDraw. Figure 8.8 shows the drawing editor being run by a client. The editor consists of three parts: the canvas, which is the shared drawing area, the toolbar, and the status bar. The state of these last two parts are different on every user depending on their actions. In the toolbar, button `SEL` stands for the selection of an object. Multiple object selection is done by holding the `Shift` key while

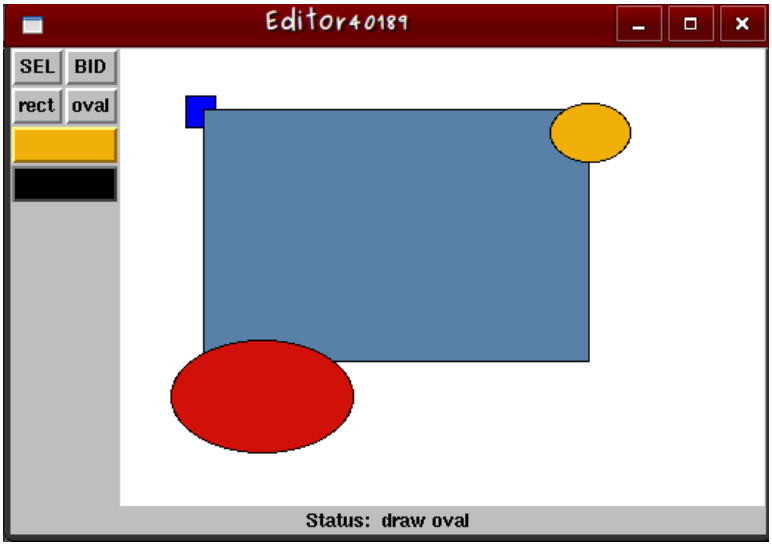


Figure 8.8: DeTransDraw graphical user interface.

selecting the objects. The buttons `rect` and `oval` allows the user to draw rectangles and ovals. These are the only figures provided on the first version of DeTransDraw. The two colored buttons represent, from top to bottom, the color of the object and its border. The status bar notifies the user of the action he is currently doing. In the case of the example, the user has clicked on the `oval` button, so it can draw a yellow oval with black border, as it is described by the coloured buttons. If the user is in selection mode, he is able to select either rectangles or ovals. A selected object appears with eight dots surrounding the object, as it will shown on Figure 8.9.

Figure 8.9 shows how the action of selecting drawing objects change the state of the network. The figure shows four application windows. The window at the top left corner is a screenshot of PEPINO [GMV07], an application that monitors the network and shows it state. In this case, the network is composed by 17 peers. The other three windows are instances of DeTransDraw which are connected to the network. Looking at the tool bars, we can deduce that the user at the top right corner draw the yellow oval, the user at the bottom left draw the blue square, and the highlighted user at the bottom right corner draw the large blue-gray rectangle. This highlighted user has selected the two ovals acquiring the correspondent locks. We observe in PEPINO some peers in blue, and some other in cyan. The peers in blue are the transaction participants which are currently locked. They are the replicas storing the state of the two ovals. Peers in cyan are the replicated transaction manager, being the peer in green the transaction manager for this operation. The other users do not see the modification of the position of the figures, because the other user has not committed yet its

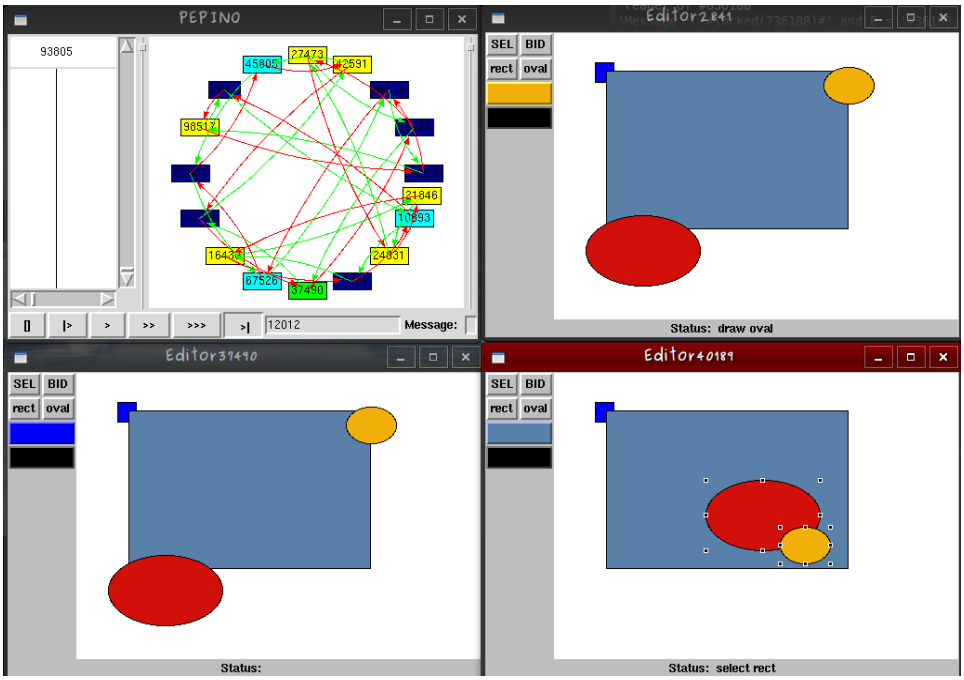


Figure 8.9: Locking phase in Detransdraw. The user with highlighted window has selected two figures to move them on the drawing. Blue peers on the ring show where are the locked replicas.

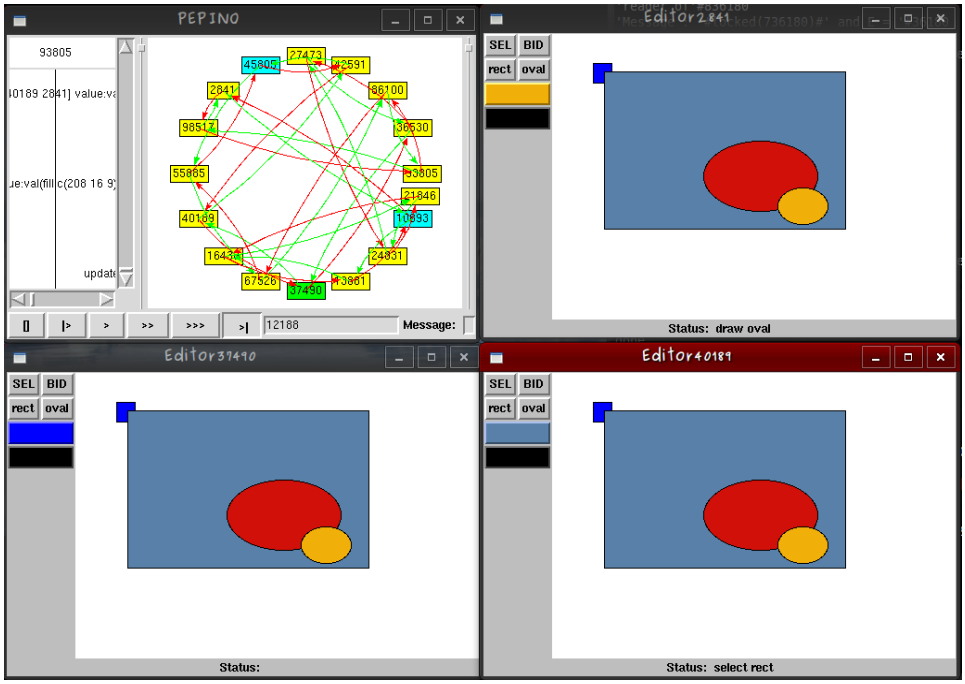


Figure 8.10: Commit phase in DeTransDraw. The user commits the changes, the new state is propagated to the other users, and locks are released.

modification.

We observe in Figure 8.10 that locks are released, and the new state of the ovals is replicated. All three instances of DeTransDraw observe the new state and the small black dots of selection disappear from the ovals that were modified. Looking at the network, there are no more blue peers, meaning that locks are released, but there still remains the information about the transaction manager and some of its replicas.

The software still needs more development to become a real drawing tool, but it is well advanced as a proof of concepts concerning its decentralized behaviour. It provides the same advantages as TransDraw minimizing the impact of network latency, allowing collaborative work with conflict resolution achieved with transactional protocols. It does not have any single point of congestion or failure, because every transaction has its own transaction manager, with a set of replicated transaction managers symmetrically distributed through the network. State is also decentralized on the DHT, having each item symmetrically replicated. Each transaction guarantees atomic updates of the majority of the replicas.

Instructions to download, install and run DeTransDraw can be found on the web site `http://beernet.info.ucl.ac.be/detransadraw`. There is also a work in progress version for Android mobile devices that soon will

be availale. Currently, mobile phones can only connect to existing peer-to-peer networks, run the GUI client and use an existing peer to transmit the messages to the network. The mobile phone is no yet a peer on its own because of performance issues. The development of the graphics has to be done in Java, and therefore, interacting Java with Mozart-Oz makes the application a bit slow. These implememtation problems are only associated to the graphical part, and they do not represent a problem at the level of peer-to-peer protocols.

8.3 Other applications

We present in this section two other applications implemented by other people using Beernet and the relaxed-ring. One is a student project, and the other one is an attempt to decentralize Twitter so as to not depend on the availability of their servers. These applications help us to show the impact of the contribution of this work.

8.3.1 Decentralized Wikipedia

Wikipedia [Wik09] is an online encyclopedia written collaboratively by volunteers, reaching currently more than 13 million articles. A large community of users constantly updates the articles and create new ones. Such system can certainly benefit from scalable storage and atomic commit, being a good case study for self-organizing peer-to-peer networks with transactional DHT. A fully decentralized Wikipedia [PRS07] was successfully built with Scalaris [SSR08], which is based on Chord$^{\#}$ [SSR07] using a transactional layer implementing Paxos consensus algorithm [MH07]. We presented the main characteristics of Chord$^{\#}$ in Chapter 2, and we described in detail Paxos consensus algorithm in Section 6.2. The real Wikipedia runs on a server farm with a fix amount of nodes, with a centrally-managed database. The decentralized version allows the network to add more nodes to the system when more storage capacity is needed. The stored items are symmetrically replicated, and each transaction runs its own instance of a transaction manager, preventing the system from having a single point of congestion.

To validate our implementation of the atomic transactional DHT using Paxos consensus algorithm, which is part of Trappist, running on top of the Relaxed-Ring, we decided to give the task of implementing a decentralized Wikipedia to the students of the course "Languages and Algorithms for Distributed Applications" [Van09], given at the Université catholique de Louvain, as a course for engineering and master students. The students had two weeks to develop their program having access to Beernet's API for building their peer-to-peer network, and for using the transactional layer to store and retrieve data from the network.

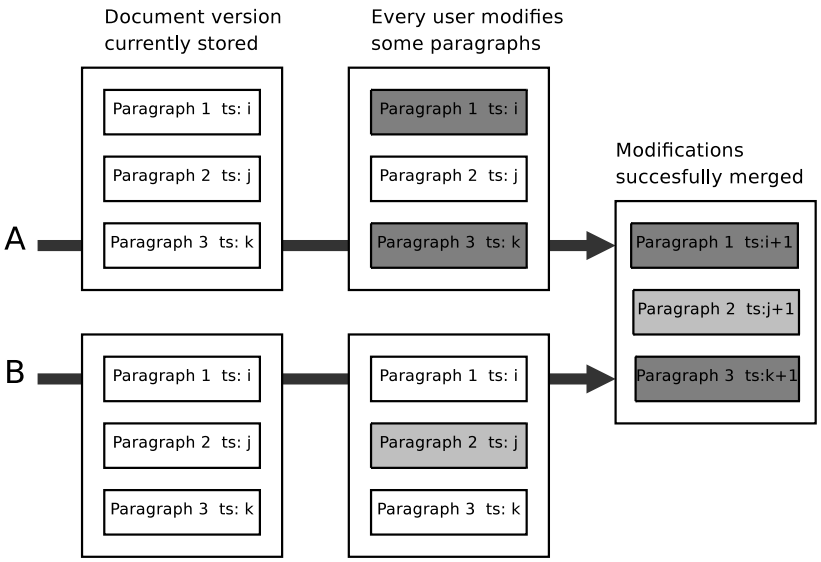


Figure 8.11: Users A and B modify different paragraphs of the same document. Both can successfully commit their changes because there are no conflicts.

To store data in a DHT, the information has to be stored as items with a key-value pair. A paragraph in an article was the granularity used to organize the information of the wiki. Articles where stored as a list of paragraphs. Using articles as the minimal granularity would have not been convenient because users never update more than one article at the time. Therefore, the transactional layer would have been used to update only an item at the time, being useful only for managing replica consistency. Furthermore, such granularity would not allow concurrent user to work on the same article. Figure 8.11 depicts how using paragraphs as the minimal granularity can be useful to allow concurrent users updating the same article. On the figure, both users get a copy of an article composed by three paragraphs. Each paragraph has its own version, marked as timestamps (ts). User A modifies paragraph 1 and 3, while user B modifies paragraph 2. When user A commits her changes, the transactional layer guarantees that both paragraph will be updated, or none of them will. This property is particularly interesting if we consider that the article could be source code of a program instead. Allowing only one change could introduce an error in the program. Continuing with the example, since modifications of users A and B do not conflict, both transactions commit successfully. Consequently, if user B would have also modified either paragraph 1 or 3, only one of the commits would have succeeded. It is up to the application to decide how resolve the conflict.

The code samples used in this section are taken and modified from one of the student projects, which was called WikiPi2Pedia, with permission of the

authors Alexandre Bultot and Laurent Herbin. Getting a copy of an article was divided into two transactions. The first one, wrapped inside the function `GetArticle`, return the list of keys representing the paragraphs associated to a given article, see Code 27. The title of the article is given as the key of the item. The variable `Node` represents the peer, and the operation performed is `executeTransaction(Trans Client paxos)` with the following parameters: `Trans` is a procedure receiving a transactional object as parameter, which actually the one over which the operation `read` is performed. The global variable `Client` its a port where the outcome of the transaction, either commit or abort, will be sent. The argument `paxos` is given to chose the protocol to be used for this transaction.

Algorithm 27 Getting the list of paragraphs keys from an article

```
fun {GetArticle Title}
   Value
   Trans = proc {$ Obj}
           {Obj read(Title Value)}
        end
in
   {Node executeTransaction(Trans Client paxos)}
   Value
end
```

The second steps for getting the text of an article is to retrieve the values of all the paragraphs. This is done in a similar way in Code 28, with the main difference that many items are read on this transaction. Every resulting `Value` from the `read` operation is added to the list of paragraphs, as it is shown in the following sample code. The operator '|' is used to put the `Value` at the head of the existing list of `Paragraphs`.

Reading an article is divided on these two steps to separate the issue of knowing is an article exist or not. If the article does not exist, a failed value will be the result of the transaction. The disadvantage is that the list of paragraphs can change in between these two steps, and therefore, the displayed article could miss some recently new paragraphs, or still display some deleted information. However, the risk that some other user makes these updates during the session of reading the article is even higher. So the disadvantage can be neglected.

Code 29 performs several transactions so as to update the article. The modifications are divided into two list of paragraphs, which are determined by the application: `ToCommit`, containing all paragraphs with modifications, and newly added paragraphs too; `ToDelete` are obviously the paragraph that will be deleted. These procedure implies several calls to `write` and `remove` on the transactional object. Calling `executeTransaction` on the `Node` guarantees that all of them will be committed, or the whole update fails.

Algorithm 28 Get the text from each paragraph

```
proc {GetPars ParIds}
   Paragraphs = {NewCell nil}
   Trans = proc {$ Obj}
          for K in ParIds do
            Value in
            {Obj read(K Value)}
            Paragraphs := Value|@Paragraphs
          end
        end
in
   {Node executeTransaction(Trans Client paxos)}
   @Paragraphs
end
```

This version is slightly simplified, because adding and removing items has also implications on the list of paragraphs of the article. The representation of such list is application dependent, so we will not include it on these code samples.

As we can see, reading an article and commit the correspondent updates is fairly simple using the transactional DHT API. As an average, the student projects were about 600 lines of code, including the graphical interface, and the code for bootstrapping the peer-to-peer network. The students were not asked to implemented an HTML interface. Instead, they could implement a simple GUI using the Mozart programming system [Moz08], to make it simpler to interact with Beernet. Figure 8.12 is a screenshot of another submitted project called WikipediOz's, with permission of the authors Quentin Pirmez and Laurent Pierson. The figure depicts how the GUI works, and opposite to Figure 8.11, it represents an example of a failed transaction due to a conflict on the edition.

The user running the window at the left of the image has modified paragraph 1 of the article entitled Patagonia[1]. The user running the window on the right has also modified paragraph 1 of the same article, in addition to modifications on paragraph 4. Even without reading the test, we can observe that paragraphs 1 and 4 are longer on the right side of the screenshot. By clicking on button `Save`, the commit transaction is triggered by the user on the right side, receiving a message *Commit successful* on green. The user on the left, executing the transaction afterwards, gets an error message *Commit failed* in red. To complete the description of the screenshot, at the bottom of the window there is a text field that allows searching for articles. The `Search` action performs the reading transactions. At the op of the window

[1]First four paragraphs taken from the Wikipedia `http://en.wikipedia.org/wiki/Patagonia`

Algorithm 29 Committing updates and removing paragraphs

```
proc {RobustCommit ToCommit ToDelete}
   Trans = proc {$ Obj}
           for UpdPar in ToCommit do
             {Obj write(UpdPar.id UpdPar.text)}
           end
           for DelPar in @ToDelete do
             {Obj remove(DelPar.id DelPar.text)}
           end
           {Obj commit}
         end
in
   {Node executeTransaction(Trans Client paxos)}
end
```

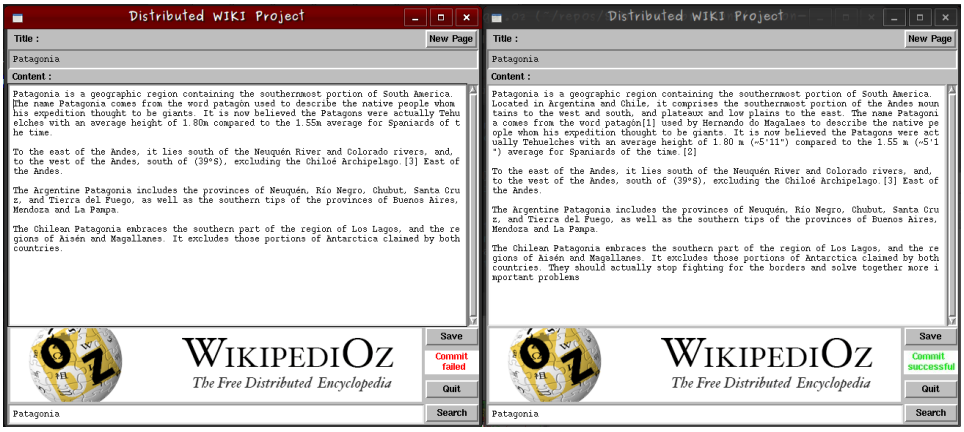


Figure 8.12: The user at the left modifies paragraph 1 of the article, but the commit fails because the user at the right just committed modifications on paragraphs 1 and 4. Note that the size of paragraphs 1 and 4 is larger at the right window.

Figure 8.13: Twitter's centralized architecture. If Twitter servers are down, the services disappears.

there is a button that allows to create new articles.

The feedback from the students helped us to improve our system, and it confirmed us that the provided API is suitable for other programmers to develop applications on top of our system. They said that all the complexity of building the network, routing messages, storing and retrieving data from the replicas, was well hidden behind the API. Unfortunately, they got the feeling that their student project did not let them test their skills on distributed programming for decentralized systems, because they were working on a higher level. This is of course positive for Beernet as programming framework, but we need to reconsider the project as an academic activity.

8.3.2 Twitbeer

Twitter [Twi09a] is a very popular service that allows its users to quickly communicate what they are doing to the rest of the users. The concept is known as *micro-blogging*, because messages can not be longer that 140 characters. Whenever a new message is posted, it is delivered to the author's subscribers, known as the *followers*. Therefore, these messages, known as *tweets*, are not really meant for establishing one-to-one communication, but to tell all your friends or followers what are you doing at any time. Author-follower relationship is mainly based on social connections, and thus, Twitter is also considered a social network.

Twitter has a well documented API [Twi09b], based on HTML, that has allowed the development of many applications and means to interact with Twitter services. Tweets can sent using Twitter's web form, via SMS using a regular mobile phone, using a supported instant messaging (IM) applications, or via third parties applications such as other social networks. Independent of the way the message is submitted, it has to arrive to Twitter's servers. Followers can received tweets from the people they are subscribed to via the web pages of Twitter, via SMS, reading RSS, or by other application that interface the services of Twitter. Figure 8.13 shows a very general architecture about how messages are sent and delivered. The diagram says nothing about how users are authenticated, and how it delivers the message to the followers. It only depicts the means of sending and receiving tweets,

spotting the main issue with the architecture: it is centralized. It highly relies on the servers of Twitter. If the servers are down, the whole twitter service is temporary unavailable.

Due to its centralized architecture, Twitter has suffered scalability problems preventing the service to accept and deliver any tweet returning the error message: "Too many tweets! Please wait a moment and try again." This is known as the "fail whale" error message, which has appeared correlated with popular technological events such as the "2008 Macworld Conference & Expo keynote address". In August 2009, Twitter was down for several hours due to a denial of service attack. These are real examples that evidence the problem of centralized architectures. The latest one inspired the development of Twitbeer, a twitter service running on top of the relaxed-ring as a peer-to-peer service. The idea is from Alfredo Cádiz, a researcher of the Université catholique de Louvain, which was disappointed of not being able to send tweets to his friends, and he was not able to follow them either. However, he was able to contact some of them via IM applications such as Jabber [Jab09], GTalk [Goo09b] and ICQ [Ame09]. They realized that they could bootstrap a peer-to-peer service with the same capabilities as Twitter, but without the need for a centralized server.

Twitbeer's architecture is depicted in Figure 8.14. All nodes in the diagram represent Twitter users. Some of the users are also connected to ICQ, and some others to Jabber. Not all connection lines are drawn to keep good visibility of the example. Several nodes form a Twitbeer ring that interface the original Twitter service. If the Twitter server goes down, the peer-to-peer service keeps running and users can submit tweets to their followers. Other peers that are not connected to the ring can establish contact using their respective IM application. Finally, all nodes become peers of the Twitbeer network. When the Twitter service is back, Twitbeer's log can be resubmitted to keep historical information. Of course, this last feature is limited by the possibilities offered by Twitter's API.

Twitbeer will not be implemented directly on top of Beernet. It will use its own implementation of the relaxed-ring, written in Objective-C [App09b], which is an object-oriented programming language that uses SmallTalk semantics for C programs. The election of the progrmming language is driven by the idea of running the relaxed-ring on the OS X iPhone [App09a].

8.4 Conclusion

We have presented four applications on this chapter that make use of Beernet, which implements the contributions of this thesis, being the Relaxed-Ring peer-to-peer network topology, and the transactional layer Trappist. Sindaca, a community driven recommendation system makes a extensive use of transactions exploiting mainly the Paxos consensus algorithm. Al-

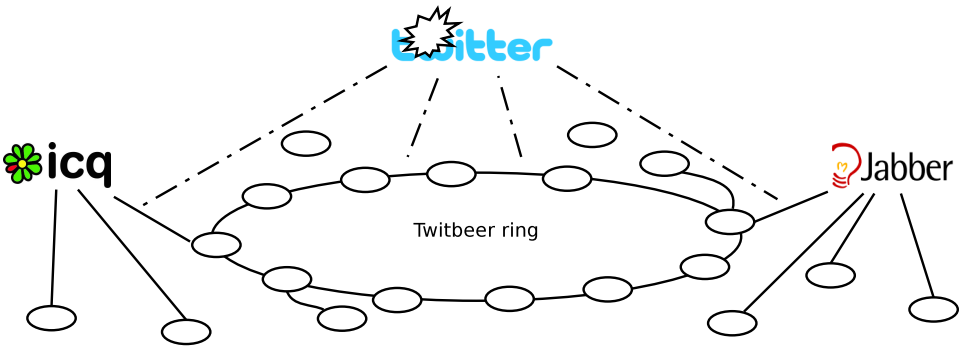


Figure 8.14: Twitbeer's decentralized architecture. Users connected to Twitter have other means to reach other. If Twitter fails, using other services they can contact with each other and keep on *twitting*.

most every transaction modify at least three series of items. All information is stored according to the DHT, and each item is symmetrically replicated. DeTransDraw is a redesign and implementation of centralized application for collaborative drawing. Since the collaborative work is done synchronously, it uses mainly the Eager Paxos protocol to notify all users about changes and potential changes on the shared area. The application is a combination of optimistic and pessimistic approach for transactions, and it does not suffer from the single points of failure or congestion. A version for Android mobile phones is in progress.

We have also described applications not written or design by the author to validate Beernet as a programming framework to build decentralize applications. A minimal version of a Wikipedia has been implemented by pre-graduate students in only two weeks. Students said that the API was simple to work with, avoiding the problems to deal with decentralized and replicated storage. Twitbeer is the design of a project that aims to be an alternative to the centralized server of Twitter when this one is down.

Chapter 9

Conclusions

We have started this dissertation discussing about the complexity of building dynamic distributed systems. We have identified important issues in the foundations of distributed programming such as partial failure and non-transitive connectivity. We spot the problems of centralized systems which cannot scale and are very fragile because of their single point of failure. The thesis of this dissertation is that one way of dealing with the complexity of such systems is to build them self-managing and decentralized, and that global state of the system must be replicated across the network.

We review existing solutions with similar goals to our thesis in Chapter 2, where we identify the advantages and disadvantages of each of them, and their degree of self-management. In Chapters 3 and 4, 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. These claims are evaluated and validated in Chapter 5. We consider the performance degradation 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. The Relaxed-Ring is enhanced with a self-adaptable finger table that is able

to scale up and down, building a more efficient routing table according to the size of the network.

The basic DHT provided by the Relaxed-Ring has been improved with a replication layer built op top of it. The layer is built using symmetric replication as the strategy to place the items in the network. To guarantee the consistency and coherence of the replicas, a transactional layer called Trappist is in charge of providing atomic updates of the items, with the guarantee that the majority of the replicas store the latest value. Trappist implements three different transactional protocols, which are described in Chapter 6. This layer is part of the whole implementation of Beernet, which provides a self-managing peer-to-peer network with transactional replicated DHT. The fundamental concepts used on the implementation of Beernet and its architecture are described in detail in Chapter 7.

To validate the ideas presented in this dissertation, we show in Chapter 8 a set of applications built on top of Beernet. They take advantages of the different transactional protocols to provide synchronous and asynchronous collaborative tools. Two of the applications we present in the chapter are design and implemented by third parties, validating Beernet as programming framework.

Bibliography

[AAG+05] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Seif Haridi, and Manfred Hauswirth. The essence of p2p: A reference architecture for overlay networks. In Germano Caronni, Nathalie Weiler, Marcel Waldvogel, and Nahid Shahmehri, editors, *Peer-to-Peer Computing*, pages 11–20. IEEE Computer Society, 2005.

[AB95] MySQL AB. MySQL: The world's most popular open source database. `http://www.mysql.com`, 1995.

[AEABH03] 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.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009.

[AH02] Karl Aberer and Manfred Hauswirth. An overview on peer-to-peer information systems. In *WDAS-2002 Proceedings*. Carleton Scientific, 2002.

[AH08] Cosmin Arad and Seif Haridi. Practical protocol composition, encapsulation and sharing in kompics. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:266–271, 2008.

[Ama09] Amazon. Amazon web services. `http://aws.amazon.com`, 2009.

[Ame09] America Online. ICQ.com - community, people search and messaging service! `http://www.icq.com`, 2009.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[App09a] Apple Inc. iPhone Dev Center - Apple Developer Connection. `http://developer.apple.com/iphone/`, 2009.

[App09b] Apple Inc. The Objective-C 2.0 Programming Language. 2009.

[Arm96] J. Armstrong. Erlang –- a survey of the language and its industrial applications. In *INAP'96 –- The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[Aud01] Audiogalaxy. The audiogalaxy satellite. `http://www.audiogalaxy.com/satellite/`, 2001.

[BCvR09] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[BL03] Chonggang Wang Bo and Bo Li. Peer-to-peer overlay networks: A survey. Technical report, 2003.

[Bre00] 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.

[Cho04] Chord Developers. The Chord/DHash project. `http://pdos.csail.mit.edu/chord/`, 2004.

[CM04] 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, pages 1060–1067. Springer, 2004.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[CMV+08] Alfredo Cádiz, Boris Mejías, Jorge Vallejos, Kim Mens, Peter Van Roy, and Wolfgang De Meuter. PALTA: Peer-to-peer

AdaptabLe Topology for Ambient intelligence. In M. Cecilia Bastarrica and Mauricio Solar, editors, *SCCC*, pages 100–109. IEEE Computer Society, 2008.

[Col07] Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, dec 2007.

[CV06] 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.

[DBK+01] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David R. Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *HotOS*, pages 81–86. IEEE Computer Society, 2001.

[DGM02] Neil Daswani and Hector Garcia-Molina. Query-flood dos attacks in gnutella. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 181–192, New York, NY, USA, 2002. ACM.

[Dis09] Distributed Systems Architecture Research Group at Universidad Complutense de Madrid. Opennebula. `http://www.opennebula.org`, 2009.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[Dou02] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[DVM+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'hondt, and Wolfgang De Meuter. *Ambient-Oriented Programming in AmbientTalk*. 2006.

[EAH05] Sameh El-Ansary and Seif Haridi. An overview of structured overlay networks. In *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*. 2005.

[Emu04] Emule. The emule file-sharing application homepage. `http://www.emule-project.org`, 2004.

[FFME04] Michael Freedman, Eric Freudenthal, David Mazières, and David Mazi Eres. Democratizing content publication with coral. In *In NSDI*, pages 239–252, 2004.

[FI03] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03*, pages 118–128, 2003.

[FLRS05] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and dhts. In *WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems*, pages 55–60, Berkeley, CA, USA, 2005. USENIX Association.

[Fre03] FreeNet Community. The freenet project. `http://freenetproject.org`, 2003.

[GDA06] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. Oscar: Small-world overlay for realistic key distributions. 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 247–258. Springer, 2006.

[GGG+03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM.

[GGV05] Donatien Grolaux, Kevin Glynn, and Peter Van Roy. Lecture notes in computer science. In Peter Van Roy, editor, *MOZ*, volume 3389, pages 149–160. Springer, 2005.

[Gho06] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH –- Royal Institute of Technology, Stockholm, Sweden, dec 2006.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[Gly05] Kevin Glynn. Extending the oz language for peer-to-peer computing. Technical report, Université catholique de Louvain, Belgium, March 2005.

[Gly07] Kevin Glynn. P2pkit: A services based architecture for deploying robust peer-to-peer applications. `http://p2pkit.info.ucl.ac.be/index.html`, 2007.

[GMV07] Donatien Grolaux, Boris Mejías, and Peter Van Roy. PEPINO: PEer-to-Peer network INspectOr. In Hauswirth et al. [HWW+07], pages 247–248.

[Gnu03] Gnutella. Gnutella. `http://www.gnutella.com`, 2003.

[GOH04] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. A novel replication scheme for load-balancing and increased security. Technical Report T2004:11, Swedish Institute of Computer Science (SICS), June 2004.

[Goo09a] Google Inc. Google app engine. `http://code.google.com/appengine/`, 2009.

[Goo09b] Google Inc. Google talk - chat online and make free internet calls. `http://www.google.com/talk/`, 2009.

[GR06] Rachid Guerraoui and Louis Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, Berlin, Germany, 2006.

[Gro98] Donatien Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA, 2002. ACM.

[HWW+07] Manfred Hauswirth, Adam Wierzbicki, Klaus Wehrle, Alberto Montresor, and Nahid Shahmehri, editors. *Seventh IEEE International Conference on Peer-to-Peer Computing (P2P 2007), September 2-5, 2007, Galway, Ireland.* IEEE Computer Society, 2007.

[HWY08] Felix Halim, Yongzheng Wu, and Roland H. C. Yap. Security issues in small world network routing. In *SASO '08: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 493–494, Washington, DC, USA, 2008. IEEE Computer Society.

[IF04] Adriana Iamnitchi and Ian Foster. A peer-to-peer approach to resource location in grid environments. pages 413–429, 2004.

[Jab09] Jabber community. Jabber: IM service based on XMPP. `http://jabber.org`, 2009.

[KA08] Supriya Krishnamurthy and John Ardelius. An analytical framework for the performance evaluation of proximity-aware structured overlays. Technical report, Swedish Institute of Computer Science (SICS), Sweden, 2008.

[KMV06] Erik Klintskog, Boris Mejías, and Peter Van Roy. Efficient distributed objects by freedom of choice. In *Revival of Dynamic Languages Workshop, ECOOP'06*, July 2006.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[LML05] Paul Lin, Alexander MacArthur, and John Leaney. Defining autonomic computing: A software engineering perspective. pages 88–97, Brisbane, Australia, March 31 - April 1 2005. IEEE Computer Society.

[LMP04] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

[LMP06] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.

[Mar02] Evangelos P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[Mat04] MathWorld. The butterfly graph. `http://mathworld.wolfram.com`, 2004.

[MCGV05] Valentin Mesaros, Raphael Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Technical report, March 2005.

[MCPV05] Boris Mejías, Raphaël Collet, Konstantin Popov, and Peter Van Roy. Improving transparency of a distributed programming system. In *"Integrated Research in Grid Computing" CoreGRID Workshop*, November 2005.

[MCV05] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for mozart. In Peter Van Roy, editor, *MOZ*, volume 3389, pages 125–136. Springer, 2005.

[Mej09] Boris Mejías. Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT. Doctoral Symposium at the XtreemOS Summer School 2009, Wadham College, University of Oxford, Oxford, UK, September 2009.

[MGV07] Boris Mejías, Donatien Grolaux, and Peter Van Roy. Improving the peer-to-peer ring for building fault-tolerant grids. In *CoreGRID Workshop on Grid-* and P2P-**, july 2007.

[MH07] Monika Moser and Seif Haridi. Coregrid series. In *Proceedings of the CoreGRID Symposium*. Springer, 2007.

[MHV08] Boris Mejías, Mikael Högqvist, and Peter Van Roy. Visualizing transactional algorithms for DHTs. In Klaus Wehrle, Wolfgang Kellerer, Sandeep K. Singhal, and Ralf Steinmetz, editors, *The Eighth IEEE International Conference on Peer-to-Peer Computing*, pages 79–80. IEEE Computer Society, 2008.

[Mic09] Microsoft Corporation. Azure service platform. `http://www.microsoft.com/azure/`, 2009.

[Mil05] Drazen Milicic. *Software quality models and philosophies*, chapter 1, page 100. Blekinge Institute of Technology, June 2005.

[MJV06] Boris Mejías, Yves Jaradin, and Peter Van Roy. Improving robustness in P2PS and a generic belief propagation service for P2PKit. Technical report, Department of Computing Science and Engineering, Université catholique de Louvain, December 2006.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. pages 31–44, 2002.

[MMR95] Martin Müller, Tobias Müller, and Peter Van Roy. Multiparadigm programming in oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 dec 1995. A Workshop in Association with ILPS'95.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. pages 183–192, 2002.

[Moz08] Mozart Consortium. The mozart-oz programming system. `http://www.mozart-oz.org`, 2008.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In Vijay Saraswat, editor, *ASIAN'03*. Springer Verlag, December 2003.

[MTSL05] Mark S. Miller, E. Dean Tribble, Jonathan Shapiro, and Hewlett Packard Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[MV07] 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.

[MV08] Boris Mejías and Peter Van Roy. The relaxed-ring: a fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, 2008.

[Nap99] Napster, Inc. Napster. `http://www.napster.com`, 1999.

[Ope01] OpenNap Community. Open source napster server. `http://opennap.sourceforge.net`, 2001.

[Ove04] Overnet. The overnet file-sharing application homepage. `http://www.overnet.com`, 2004.

[Par09] XtreemOS Partners. XtreemOS: Enabling Linux for the grid. `http://www.xtreemos.org`, 2009.

[PGES05] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. 2005.

[Pos09] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. `http://www.postgresql.org/`, 2009.

[Pro08] Programming Languages and Distributed Computing Research Group, UCLouvain. P2ps: A peer-to-peer networking library

for mozart-oz. `http://gforge.info.ucl.ac.be/projects/p2ps/`, 2008.

[Pro09] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network. `http://beernet.info.ucl.ac.be`, 2009.

[PRS07] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. pages 256–267. 2007.

[RD01a] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329, 2001.

[Res09] Reservoir Consortium. Reservoir: Resources and services virtualization without barriers. `http://www.reservoir-fp7.eu`, 2009.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[RFI02] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system. *IEEE Internet Computing Journal*, 6:2002, 2002.

[RGK+05] 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.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *In Proceedings of the USENIX Annual Technical Conference*, 2004.

[SAZ+02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure, 2002.

[SGH07] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Handling network partitions and mergers in structured overlay networks. In Hauswirth et al. [HWW+07], pages 132–139.

[SGH08] 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.

[SMK+01] 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.

[Smo95] Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, chapter Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[SMS+08] 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, jun 2008.

[SSR07] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A structured overlay for multi-dimensional range queries. In *Euro-Par 2007*, 2007.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[ST05] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[The03] The PlanetLab Consortium. Planetlab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`, 2003.

[The09a] The Apache Software Foundation. Apache http server. `http://www.apache.org`, 2009.

[The09b] The Globus Alliance. Nimbus Open Source IaaS Cloud Computing Software. `http://workspace.globus.org/`, 2009.

[The09c] The PHP Group. PHP: Hypertext Preprocessor. `http://www.php.net`, 2009.

[TT03] Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(4):96–94, 2003.

[TTF+06] Paolo Trunfio, Domenico Talia, Paraskevi Fragopoulou, Charis Papadakis, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Seif Haridi. Peer-to-peer models for resource discovery on grids. In *Proc. of the 2nd CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, Paris, France, January 2006.

[TTH+07] Paolo Trunfio, Domenico Talia, Seif Haridi, Paraskevi Fragopoulou, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Harris Papadakis. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems*, 23(7):864–878, August 2007.

[TTZH06a] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Högqvist. A dht-based peer-to-peer framework for resource discovery in grids. Technical Report TR-0048, June 2006.

[TTZH06b] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Högqvist. A peer-to-peer framework for resource discovery in large-scale grids. In *Proc. of the 2nd CoreGRID Integration Workshop*, pages 249–260, Krakow, Poland, October 2006.

[TV01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Twi09a] Twitter, Inc. Twitter. `http://twitter.com`, 2009.

[Twi09b] Twitter, Inc. Twitter API documentation. `http://apiwiki.twitter.com`, 2009.

[Van06] Peter Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[Van09] Peter Van Roy. Languages and algorithms for distributed applications. `http://www.info.ucl.ac.be/Enseignement/Cours/SINF2345/`, 2009.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[VMG+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.

[Wik09] Wikimedia Foundation. Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Wikipedia`, 2009.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1994.

[ZHS+03] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.

[Zip29] G. K. Zipf. Relative frequency as a determinant of phonetic change. In *Harvard Studies in Classical Philology*, volume 15, pages 1–95, 1929.

A.15 Beernet: RMI-free peer-to-peer networks

Beernet: RMI-free peer-to-peer networks

Boris Mejías, Alfredo Cádiz, Peter Van Roy
Département d'ingénierie informatique
Université catholique de Louvain, Belgium
{firstname.lastname}@uclouvain.be

ABSTRACT

The key issue in distributed programming is partial failure: how to handle failures of part of the system. This unavoidable property causes uncertainty because we cannot know whether a remote object is ever going to reply to a message. It is also the reason why RMI/RPC is difficult to use. In this paper we describe the most convenient object-oriented mechanism we have found to develop peer-to-peer applications effectively, namely by using active objects that communicate via asynchronous message passing and fault streams for failure handling. We show that this works better than the usual approach of using RMI to communicate and distributed exceptions for failure handling. We define our peers as lightweight actors and we use them to build a highly dynamic peer-to-peer network that deals well with partial failure and non-transitive connectivity. We give many code examples to show the simplicity and naturalness of our approach.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Languages, Design

Keywords

actors, message-passing, distributed-programming

1. INTRODUCTION

The goal of distributed computing is to achieve the collaboration of a set of different processes. A process is an abstraction of an entity that can perform computations. This entity can be a computer, a processor in a computer, or a thread of execution in a processor. In order to achieve the collaboration of processes, there are several programming paradigms aiming to help developers to build distributed systems. One definition of a distributed system is given by Tanenbaum and van Steen [13]:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system"

This definition suggests using *distribution transparency*, where all the effort of distributed programming is moved to the construction of a middleware that supports the distribution of the programming language entities. But network and computer failures cause unexpected errors to appear at higher abstraction levels, which breaks transparency and complicates programming.

There are four main concerns that make distributed programming harder than a sequential program running in a single process. They are clearly described by Waldo et al. [16], and they involve *latency*, *memory access*, *concurrency* and *partial failure*. Latency is not a critical problem because it does not change the semantics of performing an operation on a local or a distributed entity. It just makes things go a bit slower. Memory access is solved by using a virtual machine that abstracts the access, and then it does not change the operational semantics either. A more difficult problem is concurrency. The middleware has to guarantee exclusive access to the state in order to avoid race conditions. There are different techniques such as data-flow, monitors or locks, that makes possible the synchronization between processes achieving a coherent state. Given that, and even though it is not trivial to write concurrent programs correctly, it is not a critical problem either. What really breaks transparency is partial failure. Basically, distribution transparency works as long as there is no failure.

A partial failure occurs when one component of the distributed system fails and the others continue working. The failure can involve a process or a link connecting processes, and the detection of such a failure is a very difficult task. In distributed environments such as the Internet, it is impossible to build a perfect failure detector because when a process p stops responding, another process p' cannot distinguish if the problem is caused by a failure on the link connecting process p or the crash of the process p itself. This explanation might be trivial, but it is usually forgotten. Failures are a reality on distributed systems. Another definition of a distributed system is given by Leslie Lamport:

> "A distributed system is one in which the fail-

DO21 '09 Genova, Italy

ure of a computer you did not even know it existed can render your own computer unsuable"

Even though this definition does not describe the possibilities of a distributed system, it makes explicit why distributed computing is special.

The classical view of distributed computing sees partial failure as an error. For instance, a remote method invocation (RMI) on a failed object raises an *exception*. This approach actually goes against distribution transparency, because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another less fundamental issue but still relevant, is that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives.

New trends in distributed computing, such as ambient intelligence and peer-to-peer networks, see partial failure as an *inherent characteristic* of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. We believe that this approach leads to more realistic language abstractions to build distributed systems.

In this paper we discuss the design decisions we have made to build our peer-to-peer network Beernet [11]. Our approach is based on asynchronous message passing to communicate between processes, and on actors [1] as components to organize every process, providing encapsulated state and avoiding shared-state concurrency. We discard the use of RPC or RMI between class-based objects, and we avoid raising exceptions due to broken distributed entities. We have made these decisions based on the needs of building Beernet. We believe that a peer-to-peer network is an interesting case study for distributed systems, because it is highly dynamic with respect to connectivity between peers and because it does not have a central point of control. Both these properties make the management of the system more complex.

Our model is influenced by the programming languages Oz [10] and Erlang [3], and by the algorithms of the book "Introduction to Reliable Distributed Programming" [7], which presents an event-driven model that we describe in the following section. Section 3 presents incrementally the language abstractions we have used in our approach. Section 4 summarizes the related work discussed along the paper. Section 5 finishes by recapitulating our main conclusions.

2. EVENT-DRIVEN COMPONENTS

The algorithms for reliable distributed programming presented in [7] are designed in terms of components that communicate through events. Every component has its own state, which is encapsulated, and every event is handled in a mutually exclusive way. The model avoids shared-state concurrency because the state of a component is modified by only one event at the time.

Every component provides a specific functionality such as point-to-point communication, failure detection, best effort broadcast, and so forth. Components are organized in layers where the level of the abstraction is organized bottom-up. A higher-level abstraction requests a functionality from a more basic component by triggering an event (sending a request). Once the request is resolved, an indication event is sent back to the abstraction (sending back a reply). Algorithm 1 is taken from the book, where only the syntax has been slightly modified. It implements a best-effort broadcast using a more basic component, (*pp2p*), which provides a perfect point-to-point link to communicate with other processes.

Algorithm 1 Best Effort Broadcast

```
1: upon event ⟨ bebBroadcast | m ⟩ do
2:     for all p in other_peers do
3:         trigger ⟨ pp2pSend | p, m ⟩
4:     end
5: end

6: upon event ⟨ pp2pDeliver | p, m ⟩ do
7:     trigger ⟨ bebDeliver | p, m ⟩
8: end
```

The best-effort broadcast (*beb*) component handles two events: *bebBroadcast* as requested from the upper layer, and *pp2pDeliver* as an indication coming from the lower layer. Every time a component requests *beb* to broadcast a message m, *beb* traverses its list of other peers, triggering the *pp2pSend* event to send the message m to every peer p. Every p is a remote reference, but it is the *pp2p* component which takes care of the distributed communication. At every receiving peer, the *pp2p* component triggers *pp2pDeliver* upon the reception of a message. When *beb* handles this event, it triggers *bebDeliver* to the upper layer, as seen in Algorithm 1. It is important to mention that *beb* does not have to wait for *pp2p* every time it triggers *pp2pSend*, and that *pp2p* does not wait for *beb* or any other component when it triggers *pp2pDeliver*. This asynchronous communication between components means that each component can potentially run in its own independent thread.

Using layers of components allows programmers to deal with issues concerning distribution only at the lowest layers. For instance, the component *beb* is conceived only with the goal of providing a broadcast primitive. The problem of communicating with a remote processes through a point to point communication channel is solved in *pp2p*. If a process p crashes while the message is being sent, it does not affect the code of *beb*, thus improving the transparency of the component. There is no need to use something like

try $\langle send\ m\ to\ p \rangle$ **catch** $\langle failure \rangle$

It is the responsibility of *pp2p* to deal with the failure of p. It is also possible that *pp2p* triggers the detection of the crash of p to the higher level, and then it is up to *beb* to do something with it, for instance, removing p from the list of other peers to contact. In such a case, the failure of p is considered as part of the normal behaviour of the system, and not as an exception. Even though the code for the maintenance of *other_peers* set is not given [7], we can deduce it from the implementation of the other components. In Algorithm 2 the *register* event is a request made from the upper layer, and *crash* is an indication coming from the *pp2p* layer.

Even though we advocate defining algorithms using event-driven components using the approach of [7], there are some important drawbacks to consider. To compose layers, it is necessary to create a channel, connect the components using the channel, and subscribe them to the events they will handle. We find this approach a bit over sophisticated. It

Algorithm 2 Best Effort Broadcast extended

```
1: upon event ⟨ bebRegister | p ⟩ do
2:     other_peers := other_peers ∪ {p}
3: end

4: upon event ⟨ crash | p ⟩ do
5:     other_peers := other_peers \ {p}
6: end
```

could be simplified by talking directly to a component and using a default listener only when necessary. A related problem concerns the naming convention of events. The name reflects the component implementing the behaviour, making the code less *composable*. For instance, if we want to use a fair-loss point-to-point link (*flp2p*) instead of *pp2p*, we would have to change the *beb* code by replacing *pp2pSend* by *flp2pSend*, and instead of handling *pp2pDeliver* we would have to handle *flp2pDeliver*.

Since the architecture considers components and channels, an alternative and equivalent approach would be to use objects with explicit triggering of events as method invocation, instead of using anonymous channels. Using objects as collaborators, they could be replaced without problems as long as they implement the same interface. In such an approach, both *flp2p* and *pp2p* would handle the event *send* and trigger *deliver*.

The other problem of [7] is that there is no explanation of how to transfer a message from one process to the other. The more basic component *flp2p* is only specified in terms of the properties it holds, but it is not implemented. There is no language abstraction to send a message to a remote entity.

3. BEERNET

Beernet [11] is a library implemented in Mozart/Oz [10] that provides an API to build peer-to-peer applications. Mozart is an implementation of the Oz language, which is a multi-paradigm programming language supporting functional, concurrent, object-oriented, logic and constraint programming paradigms [15], and offering support for distributed programming with a high degree of transparency. Thanks to the multi-paradigm support of Oz, we were able use more convenient language abstractions for distribution and local computing while building Beernet. In this section we discuss the basic language abstractions that we considered appropriate and necessary to implement event-driven components, and which abstractions allowed us to improve the approach towards an event-driven actor model.

The peer-to-peer network built by Beernet uses the relaxed-ring network topology [8]. It provides a distributed hash table (DHT) with replicated storage using distributed transactions to guarantee data consistency. A peer-to-peer network is a very interesting case study of a distributed system because it is very dynamic. Peers are constantly joining and leaving the network, either as graceful leaves or due to failures. It does not use a central point of control and it can be run without relying on an existing routing infrastructure because it provides its own structured overlay network for message routing. Because of the context, during the rest of the paper we use the term peer as equivalent to the previously used process.

3.1 Threads and data-flow variables

One of the strengths of the Oz language is its concurrency model which is easily extended to distribution. The kernel language is based on procedural statements and single-assignment variables. When a variable is declared, it has no value yet, and when it is bound to a value, it cannot change the value. Attempting to perform an operation that needs the value of such a variable will wait if the variable has no value yet. In a single-threaded program, that situation will block forever. In a multi-threaded program, such a variable is very useful to synchronize threads. We call it a data-flow variable. Oz provides lightweight threads running inside one operating system process with a fair thread scheduler.

The code in algorithm 3 shows a very simple example of data-flow synchronization. First, we declare variables *Foo* and *Bar* in the main thread of execution. Then, a new thread is created to bind variable *Bar* depending on the value of *Foo*. Since the value of *Foo* is unknown, the '+' operation waits. A second thread is created which binds variable *Foo* to an integer. At this point, the first thread can continue its execution because the value of *Foo* is known.

Algorithm 3 Threads and data-flow synchronization

```
1: declare Foo Bar
2: thread Bar = Foo + 1 end
3: thread Foo = 42 end
```

This synchronization mechanism does not need any lock, monitor, or semaphore, because there is no explicit state, and therefore, no risk for race conditions. The values of *Foo* and *Bar* will be the same for all possible execution orders of the threads. Single-assignment variables are also used in languages such as E [9] and AmbientTalk [6, 14], where they are called promises or futures. They are combined with the *when* operator as one of the mechanisms for synchronization.

The execution of a concurrent program working only with single-assignment variables is completely deterministic. While this is an advantage for correctness (race conditions are impossible), it is too restrictive for general-purpose distributed programming. For instance, it is impossible to implement a server talking to two different clients. To overcome this limitation, Oz introduces Ports, which are described in the following section.

3.2 Ports and asynchronous send

A port is a language entity that receives messages and serializes them into an output stream. After creating a port, one variable is bound to the identity of the port. That variable is used to send asynchronous messages to the port. A second variable is bound to the stream of the port, and it is used to read the messages sent to the port. The stream is just like a list in Lisp or Scheme, a concatenation of a head with a tail, where the tail is another list. The list terminates in an unbound single-assignment variable. Whenever a message is sent to the port, this variable is bound to a dotted pair containing the message and a fresh variable.

Algorithm 4 combines ports with threads. First we declare variables P and S. Then, variable P is bound to a port having S as its receiving stream. A thread is created with a *for-loop* that traverses the whole stream S. If there is no value on the stream, the *for-loop* simply waits. As soon as a message arrives on the stream, it is shown on the output console. A second thread is created to traverse a list

of beers (*BeerList*, declared somewhere else), and to send every beer as a message to port P. This is a like a barman communicating with a client. Everybody who knows P can send a message to it, as in the third thread, where the list of sandwiches is being traversed and sent to the same port. Beers will appear on the stream in the same order they are sent. Beers and sandwiches will be merged in the stream of the port depending on the order of arrival, so the order is not deterministic between them.

Algorithm 4 Port and asynchronous message passing

```
 1: declare P S
 2: P = {NewPort S}
 3: thread
 4:     for Msg in S do {Show Msg} end
 5: end
 6: thread
 7:     for Beer in BeerList do {Send P Beer} end
 8: end
 9: thread
10:     for Sdwch in SandwichList do {Send P Sdwch} end
11: end
```

The send operation is completely asynchronous. It does not have to wait until the message appears on the stream in order to continue with the next instruction. The actual message send could therefore take an arbitrary finite time, making it suitable for distributed communication where latency is an issue. With the introduction of ports, it is already possible to build a multi-agent system running in a single process where every agent runs on its own lightweight thread. The non-determinism introduced with ports allows us to work with explicit state, and there is no restriction on the communication between agents.

3.3 Going distributed

Event though full distribution transparency is impossible to achieve because of partial failures, there is some degree of transparency that is feasible and useful. Ports and asynchronous message passing as they are described in the previous section can be used transparently in a distributed system. The semantics of `{Send P Msg}` is exactly the same if P is a port in the same process or in a remote peer. In both cases the operation returns immediately without waiting until the message is handled by the port. If there is a need for synchronization, the message can contain an unbound variable as a future. Then, the sending peer waits for the variable to get a value, which happens when the receiving peer binds the variable. This implies that the variable, and whatever is contained in the message, is transparently sent to the other peer. Variable binding must therefore be transparent.

Algorithm 5 does a ping-pong between two different peers. Code lines from 1 to 5 represent peer A who sends a ping message to peer B. The message contains an unbound variable *Ack*, which is bound by peer B to the value `pong`. Binding variable *Ack* resumes the *Wait* operator at peer A. Peer B, from lines 6 to 10, makes a pattern matching of every received message with pattern `ping(A)`. If that is the case, it binds A to `pong` and continues with the next message. The pattern matching is useful to implement a method dispatcher as we will see in the next section.

This sort of transparency is not difficult to achieve, except

Algorithm 5 Ping-Pong

```
 1: % at Peer A
 2: declare Ack
 3: {Send PeerB ping(Ack)}
 4: {Wait Ack}
 5: {Show "message received"}

 6: % at Peer B
 7: for Msg in Stream do
 8:     case Msg of ping(A) then
 9:         A = pong
10:     end
11: end
```

when a partial failure occurs. An older release of Mozart, version 1.3.0, takes the classical approach to deal with partial failures: it raises an exception whenever an operation is attempted on a broken distributed reference. Most programming languages take the same approach. This approach has two important disadvantages. First, it is cumbersome because it is necessary to add **try** ... **catch** instructions whenever an operation is attempted on a remote entity. More fundamentally, exceptions break transparency when reusing code meant for local ports. If a *distribution* exception is raised, it will not be caught because the code was not expecting that sort of exception.

AmbientTalk [6, 14] adopts a better approach. In ambient-oriented programming, failures due to temporary disconnections are a very common thing, therefore, no exception is raised if a message is sent to a disconnected remote reference. The message is kept until the connection is restored and the message is resent. Otherwise if the connection cannot be fixed after a certain time, it will be garbage collected. Failures are also a common thing in peer-to-peer networks. The normal behaviour of a peer is to leave the network after some time. Therefore, a partial failure should not be considered as an exceptional situation.

A more recent Mozart release, version 1.4.0, does not raise exceptions when distributed references are broken. It simply suspends the operation until the connection is reestablished or the entity is killed. If the operation needs the value of the entity, for instance in a binding, the thread blocks its execution. If a send operation is performed on a broken port, because of its asynchrony, it still returns immediately, but the actual sending of the message is suspended until the connection is reestablished. This failure handling model [5] is based on a *fault stream* that is attached to every distributed entity. An entity can be in three states, *ok*, *tempFail*, or *permFail*. Once it reaches the permanent failure state, it cannot come back to *ok*, so the entity can be killed. If the entity is in temporary failure for too long, it can be explicitly killed by the application and forced to *permFail*. To monitor an entity's fault stream, the idea is to do it in a different thread that does not block and that can take actions over the thread blocking on a failed entity.

3.4 Event-driven Actors

The actor model [1] provides a nice way of organizing concurrent programming, benefiting from encapsulation and polymorphism in analogous fashion to object-oriented programming. We extend the previous language abstractions with Oz *cells* which are containers for mutable state. State

is modified with operator ':=', and it can be read with operator '@'. We do not need to add new language abstractions in order to build our event-driven actors. Without language support, actors are a programming pattern in Oz as is shown in Algorithm 6. Having ports, the cell is not strictly necessary but we use it to facilitate state manipulation. Every actor runs in its own lightweight thread and communicates asynchronously with other actors through ports. Encapsulation of state is achieved with lexical scoping, and exclusive access to state to avoid race conditions is guaranteed by handling only one event/message at a time.

Algorithm 6 is a working implementation of Algorithms 1 and 2 using the language abstractions we have described in this section. It is written in Oz without syntactic support for actors but the semantics are equivalent. The function *NewBestEffortBroadcast* creates a closure containing the state of the actor and its behaviour. The state includes a list of *OtherPeers* and another actor implementing perfect point-to-point communication, which is named *ComLayer* to make explicit that it could be replaced by any actor that understands event *send*, and not only *pp2p*.

The behaviour is implemented as a set of procedures where the signature of the event is specified in each procedure's argument. For instance, the declaration on code line 9 reads that procedure *Receive* implements the behaviour to handle **upon event** *deliver(Src Msg)*. The variable *Listener* represents the actor in the upper layer.

Algorithm 6 Beernet Best Effort Broadcast

```
 1: fun {NewBestEffortBroadcast Listener}
 2:     OtherPeers ComLayer
 3:     SelfPort SelfStream
 4:     proc {Broadcast broadcast(Msg)}
 5:         for Peer in OtherPeers do
 6:             {Send ComLayer send(Peer Msg)}
 7:         end
 8:     end
 9:     proc {Receive deliver(Src Msg)}
10:         {Send Listener Msg}
11:     end
12:     proc {Add register(Peer)}
13:         OtherPeers := Peer | @OtherPeers
14:     end
15:     proc {Crash crash(Peer)}
16:         OtherPeers := {Remove Peer @OtherPeers}
17:     end
18: in
19:     OtherPeers = {NewCell nil}
20:     ComLayer = {NewPP2Point SelfPort}
21:     SelfPort = {NewPort SelfStream}
22:     thread
23:         for M in SelfStream do
24:             case M.label
25:             of broadcast then {Broadcast M}
26:             [ ] deliver then {Receive M}
27:             [ ] register then {Add M}
28:             [ ] crash then {Crash M}
29:             end
30:         end
31:     end
32:     SelfPort
33: end
```

Variable *SelfPort* is bound to the port that will receive all messages coming from other actors. A thread is launched to traverse the *SelfStream*. For every message that arrives on the stream, pattern matching checks the label of the message in order to invoke the corresponding procedure. This part of the code represents the method dispatching of the actor. In the Beernet implementation, the creation of the port and the method dispatching are modularized to avoid code duplication, thus reducing the code size of every actor.

The book [7] contains complementary material including a Java implementation of the *beb* component. Discarding comments and import lines, the implementation takes 67 lines of code, with the component infrastructure already abstracted. It is worth mentioning that a large number of lines are dedicated to catch exceptions. Equivalent functionality within the Beernet actor model takes only 33 lines.

3.5 Peer-to-peer

The architecture of Beernet is based on layers that abstract the different concepts involved in the construction of the peer-to-peer network. A closely related work is the Kompics component framework [2], which follows the component-channel approach of [7] using a similar architecture. The main difference with Beernet is that instead of having components that communicate through channels, we decided to use event-driven actors.

Beernet is built on top of the relaxed-ring [8], a structured overlay network providing a distributed hash table (DHT) as in Chord [12]. In such a network peers are organized into a ring. Hash keys goes from 0 to $N-1$ forming a circular address space. Every peer joins the network with an identifier. The identifier is used to find the correct predecessor and successor in the ring. When peer q joins in between peers p and s, it means that $p < q < s$ following the ring clockwise. Peer s accepts q as predecessor because it has a better key than p. Another reason to be a better predecessor, is that the current predecessor is detected to have crashed. Hence, the maintenance of the ring involves *join* and *crash* events, and it must be handled locally by every peer in a decentralized way.

In order to keep the ring up to date, Chord performs a periodic stabilization that consists in verifying each successor's predecessor. From the viewpoint of the peer performing the stabilization, if the predecessor of my successor has an identifier between my successor and myself, it means that it is a better successor for me and my successor pointer must be updated. Then, I notify my successor. Algorithm 7 is taken from Chord [12]. Only the syntax is adapted. The big problem with this algorithm is seen in line 2. Asking for successor's predecessor is done using RMI. This means that the whole execution of the component waits until the RMI is resolved. There is no conflict resolution if *successor* is dead or dies while the RMI is taking place. If there is a partial failure, the algorithm is broken.

An improved version of the stabilization protocol is given in Algorithm 8 using event-driven actors. The representation of a peer is a data structure having *Peer.id* as the integer identifying the peer, and *Peer.port* as the remote reference, being actually an Oz port. The '.' is not an operator over an actor or an object. It is just an access to a local data structure. The '...' in the algorithm hide the state declaration and the method dispatcher loop. The '<' operator defines the order in the circular address space. We use it here for

Algorithm 7 Chord's periodic stabilization

```
1: upon event ⟨ stabilize | ⟩ do
2:     x := successor.predecessor
3:     if x ∈ (self, successor) then
4:         successor := x
5:     end
6:     successor.notify(self)
7: end
8: upon event ⟨ notify | src ⟩ do
9:     if predecessor is nil or src ∈ (predecessor, self) then
10:        predecessor := src
11:    end
12: end
```

simplicity without changing the semantics of the algorithm.

Stabilization starts by sending a message to the successor with an unbound variable X to examine its predecessor. The peer then launches a thread to wait for the variable to have a value, and once the binding is resolved, it sends a message to itself to verify the value of the predecessor. This pattern is equivalent to the *when* abstraction in E [9] and AmbientTalk [14]. By launching the thread, the peer can continue handling other events without having to wait for the answer of the remote peer. If the remote peer crashes, the *Wait* will simply block forever without affecting the rest of the computation. When the *Wait* continues, the peer sends a message to itself in order to serialize the access to the state with the handling of other messages. Otherwise there would be a race condition.

Algorithm 8 Chord's improved periodic stabilization

```
1: fun {NewChordPeer Listener}
2:     ...
3:     proc {Stab stabilize}
4:         X in
5:         {Send Succ.port getPredecessor(X)}
6:         thread
7:             {Wait X}
8:             {Send Self.port verifySucc(X)}
9:         end
10:    end
11:    proc {Verify verifySucc(X)}
12:        if Self.id < X.id < Succ.id then
13:            Succ := X
14:        end
15:        {Send Succ.port notify(Self))}
16:    end
17:    proc {GetPred getPredecessor(X)}
18:        X = Pred
19:    end
20:    proc {Notify notify(Src)}
21:        if Pred == nil
22:        orelse Pred.id < Src.id < Self.id then
23:            Pred := Src
24:        end
25:    end
26:    ...
27: end
```

Beernet uses a different strategy for ring maintenance. Instead of running a periodic stabilization, it uses a strategy called *correction-on-change*. Peers react immediately when they suspect another peer to have failed. The failed peer is removed from the routing table, and if it happens to be the successor, the peer must contact the next peer in order to fix the ring. To contact the next successor, every peer manages a successor list, which is constantly updated every time a new peer join or if there is a failure.

Algorithm 9 presents part of a *PBeer* actor, which is a Beernet peer. Failure recovery works as follows: when peer P fails, a low-level actor running a failure detector triggers the $crash(P)$ event to the upper layer, where *PBeer* handles it. *PBeer* adds the crashed peer to the crashed set and removes it from its successor list. If the crashed peer is the current successor, then the first node from the successor list is chosen as the new successor. A *notify* message is sent to the new successor. When a node is notified by its new predecessor, it behaves as a Chord node, but in addition, it replies with the *updSL* message containing its successor list. In this way, the successor list is constantly being maintained.

Algorithm 9 Beernet's failure recovery

```
1: fun {NewPBeer Listener}
2:     ...
3:     proc {Crash crash(Peer)}
4:         Crashed := Peer | @Crashed
5:         SuccList := {Remove Peer @SuccList}
6:         if P == @Succ then
7:             Succ := {GetFirst SuccList}
8:             {Send Succ.port notify(Self)}
9:         end
10:    end
11:    proc {Notify notify(Src)}
12:        if {Member Pred @Crashed}
13:        orelse Pred.id < Src.id < Self.id then
14:            Pred := Src
15:        end
16:        {Send Src.port updSL(Self @SuccList)}
17:    end
18:    ...
19: end
```

3.6 Fault streams for failure handling

As described at the end of subsection 3.3, we use a *fault stream* associated to every distributed entity in order to handle failures. An operation performed on a broken entity does not raise any exception, but it blocks until the failure is fixed or the thread is garbage collected. This blocking behaviour is compatible with asynchronous communication with remote entities. In the fault stream model, presented by Collet et al [5, 4], the idea is that the status of a remote entity is monitored in a different thread. The monitoring thread can take decisions about the broken entity, in order to terminate the blocking thread. For instance, there are language abstractions to kill a broken entity so it can be garbage collected.

Algorithm 10 describes how we use the fault stream in the implementation of Beernet. There is an actor in charge of monitoring distributed entities called *FailureDetector*. Upon event *monitor(Peer)*, the actor uses the system operation *GetFaultStream* in order to get access to the status of the remote peer. The fault stream is updated automatically by the Mozart system, which sends heartbeat messages to the remote entity in order to determine its state. When the state

changes, the new state appears on the fault stream. If the connection is working, the state is set to *ok*. If the remote entity does not acknowledge a heartbeat, it is suspected of having failed, and therefore, the state is set to *tempFail*. Since Internet failure detectors cannot be strongly accurate, the state can switch between *tempFail* and *ok* indefinitely. As soon as the state is set to *permFail*, however, the entity cannot recover from that state.

If the state is *tempFail* or *permFail*, the actor triggers the event *crash(Peer)* to the *Listener*, which represents the upper layer. If the state switches back to *ok*, the event *alive(Peer)* is triggered. It is up to the upper layer to decide what to do with the peer. In the case of Beernet, this is described in algorithm 9.

Algorithm 10 Fault stream for failure detection

```
 1: fun {FailureDetector Listener}
 2:    ...
 3:    proc {Monitor monitor(Peer)}
 4:       FaultStream = {GetFaultStream Peer}
 5:    in
 6:       for State in FaultStream do
 7:          case State
 8:          of tempFail then {Send Listener crash(Peer)}
 9:          [] permFail then {Send Listener crash(Peer)}
10:          [] ok then {Send Listener alive(Peer)}
11:          end
12:       end
13:    end
14:    ...
15: end
```

4. DISCUSSION AND RELATED WORK

One of the principles we respect in this paper is to *avoid shared-state concurrency*. We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for data-flow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives of lightweight threads and ports are also present in Erlang [3], and they are not specific to object-oriented programming. Single-assignment variables also appear in E [9] and AmbientTalk [14] in the form of promises, and they are meant for synchronization of remote processes instead of lightweight threads.

The actor model presented here through programming patterns is further developed and supported by E and AmbientTalk. There is one important difference related to the use of lightweight threads. Since they are not supported by these two languages, there is basically only one actor running per process. The actor collaborates with a set of passive objects within the same process. Communication with local objects is done with synchronous method invocation. Communication with other actors, and therefore with remote references, is done with asynchronous message passing. This distinction reduces transparency for the programmer because it establishes two types of objects: local and distributed.

In Beernet, we organize the system in terms of actors only, making no distinction in the send operation between a local and a remote port. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an actor, and only one send operation.

As mentioned in the previous section, Kompics [2] is closely related because it is also a component framework conceived for the implementation of peer-to-peer networks. Instead of using actors for composition, it uses event-driven components which communicate through channels, analogous to events in [7].

5. CONCLUSIONS

We have presented examples in this paper to highlight the importance of partial failure in distributed programming. The fact that failures cannot be avoided has a direct impact on the goal of transparent distribution which cannot be fully achieved. Therefore, it has also an impact on remote method invocation, the most common language abstraction to work with distributed objects. Because of partial failure, it is very difficult to make RMI work correctly. In other words, RMI is considered harmful. Our position is that communication within remote processes must be done with asynchronous message passing.

Even though full transparency cannot be achieved, it is important to provide some degree of transparency. We have shown how *port* references and the *send* operation can be used transparently. This is because send works asynchronously and because a broken distributed reference does not raise an exception in Mozart 1.4.0. Instead, a fault stream associated to every remote entity provides monitoring facilities.

We have also described the language abstractions we use to implement Beernet, a peer-to-peer network with a highly dynamic interaction between peers. In order to organize the behaviour of every peer, we have chosen an actor model based on lightweight threads, ports, asynchronous message passing, single-assignment variables and lexical scoping. These language abstractions are very suitable for implementing actors, and they can be used in other programming paradigms.

Acknowledgments

This work is supported by projects SELFMAN, VariBru, and MoVES. The authors would like to thank S. González and P. Hass for fruitful discussion and comments on this work.

6. REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[2] C. Arad and S. Haridi. Practical Protocol Composition, Encapsulation and Sharing in Kompics. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:266–271, 2008.

[3] J. Armstrong. Erlang –- a Survey of the Language and its Industrial Applications. In *INAP'96 –- The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[4] R. Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, Dec. 2007.

[5] R. Collet and P. Van Roy. Failure Handling in a Network-Transparent Distributed Programming Language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'hondt, and W. De Meuter. *Ambient-Oriented Programming in AmbientTalk*. 2006.

[7] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[8] B. Mejías and P. Van Roy. The Relaxed-Ring: A fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, September 2008.

[9] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in E as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[10] Mozart Community. The Mozart-Oz Programming System - *http://www.mozart-oz.org*, 2008.

[11] Programming Languages and Distributed Computing Research Group, UCL. Beernet: pbeer-to-pbeer network - *http://beernet.info.ucl.ac.be*.

[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.

[13] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[14] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.

[15] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[16] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1994.

A.16 From mini-clouds to Cloud Computing

From mini-clouds to Cloud Computing

Boris Mejías, Peter Van Roy
Université catholique de Louvain – Belgium
{boris.mejias|peter.vanroy}@uclouvain.be

Abstract

Cloud computing has many definitions with different views within industry and academia, but everybody agrees on that cloud computing is the way of making possible the dream of unlimited computing power with high availability. However, being a cloud computing provider seems to be reserved to very large companies that can achieve having a huge data center. The rest of the companies and institutions have to play the role of cloud users. We propose an architecture to organize a set of mini-clouds provided by different institutions, in order to provide a larger cloud that appears to its users as a single one. Such architecture requires self-managing behaviour in order to deal with the complexity of matching cloud users requests with the computing utility that mini-clouds of institutions can offer.

Index Terms: Self-management, Cloud-computing, System architecture.

1. Introduction

Cloud Computing is an active area in the IT industry for a couple of years already, calling immediately the attention of the research community thanks to its possibilities and challenges. Projects such as Reservoir [14], NanoDataCenters [10], XtreemOS [18] and OpenNebula [4] are just examples of the interest of the research community. However, defining cloud computing is not that simple, because there are many interpretations within industry and academia. Interpretations go from seeing cloud computing as ubiquitous computing, or that any application provided as a web service is said to be living in the cloud. Consistently with Berkeley's view of Cloud Computing [2], and sharing some of the conclusions of 2008 LADIS workshop [3], we see Cloud Computing as the combination of hardware and software that can provide the illusion of infinite computing power with high availability.

Large companies provide this illusion of infinite computing power by having real large data centers with software capable to provide access on demand to every machine on the data center. Industrial examples supporting Cloud Computing are Google AppEngine [6], Amazon Web Services [1] and Microsoft Azure [9]. This three companies follows the architecture described in [2] where the base of the whole system is such a large company being the cloud provider. Cloud users are actually smaller companies or institutions that use the cloud to become Software as a Service (SaaS) providers. The end user is actually a SaaS user, which is indifferent to the fact that a cloud is providing the computational power of the SaaS.

We assume such architecture for Cloud Computing to develop our proposal, which focuses on the interaction between the cloud user and the cloud provider. We delegate the interaction between SaaS provider and SaaS user to the design of the SaaS application itself.

When a SaaS provider decides to move its service to Cloud Computing, it has to deal with the API offered by the cloud provider. Since every cloud provider has its own API, it is not trivial to migrate from one cloud provider to another one. Another limitation we see in Cloud Computing as it is currently developed, is that only very large companies have the resources to become cloud providers. Middle to large companies with their own data center have not enough resources to indefinitely scale up independently. This is not a problem except when the service experience high peaks on their demand curve. Most of the time those data centers remain not far from idle. This is why Cloud Computing appears as a interesting solution.

Instead of focusing on companies, we consider academic institutions, such as universities, for our case study. These institutions usually have clusters and servers that most of the time are running at a low percentage of their full capacity, but they could run into problems when students generate a large demand of university's services. The same situation can occur to middle size companies. We consider that these institutions could provide a mini-cloud that alone is not enough, but that in combination with other mini-clouds can provide scalable resources on demand. We define a mini-cloud as a set of computers linked together within an institution, with the ability to provide services such as web pages and storage. A mini-cloud can‘ be one or more clus-

ters, or several computers within the same local area network.

We propose on this work an architecture that combines several mini-clouds to become a large decentralized cloud provider. The new cloud is interfaced to the SaaS provider with a manager layer that transparently handles its resource demands as if it were a single cloud provider. With respect to computing power, it is not the goal to provide the same amount of resources as a large cloud provider, which is prepared to host general-purpose cloud services to many companies using the same data center. The goal is to allow institutions to share their resources so as to use them when their demand is higher that what their mini-clouds are able to handle. As a consequence, the illusion of infinite computing power is given to a certain amount of institutions, instead of to an unbound amount of SaaS providers. The difficulty of the approach lies on the management interface, which has to be able to make the system *scale up and down*, depending on the demand of the system. We believe that scalability can be achieved in two ways: with a hierarchical and centralized structure being organized by a scheduler as in grid computing, or with a decentralized peer-to-peer network that can handle churn and scalable storage.

Scalability is certainly not the only challenge presented in cloud computing. Avoiding *data lock-in* is also very important to allow SaaS providers to migrate from one cloud provider to another at a low cost. To prevent data lock-in, and independently of the chosen strategy for organizing the nodes on the cloud, we identify the need of designing the management interface layer as a self-managing set of components that can follow a plan, but that it is also capable of self-adapting the plan according to the state of the service. Components can be reconfigured according to the adaptation plan or they can simply be replaced by other components.

In the next section we will present the general architecture to provide Cloud Computing with a set of mini-clouds. Section 3 discusses our strategy to design the interfacing layer, and we conclude in Section 4.

2. Cloud Computing with mini-clouds

The most general architecture to represent how Cloud Computing is provided by large companies, such as Google, Amazon or Microsoft, is analyzed in [2]. We can observe that architecture in Figure 1.a. The *cloud provider* is at the base of the architecture offering *utility computing* to the *cloud user*. Utility computing can be understood as a certain amount of resources during a certain amount of time, for instance, a web server running for one hour, or several Tera bytes of storage for a certain amount of days. The cloud user, which is actually a *SaaS provider*, has a predefined utility computing request, which can vary enormously depending on its users demands. At the top of the architecture we find the *SaaS user* which requests services from the SaaS provider. The service that the SaaS provider offers to its users is usually presented as a web *application*.

There are basically two things that are important to the SaaS provider: get more resources when users' demand increases more than what the current resources can handle, and release resources when the SaaS users are not demanding too much from the services. The objective is to maximize the quality of the service, and minimize the cost of utility computing demanded to the cloud provider. The system has to be able not only to scale up, but also to scale down. Idle resources are an unnecessary cost to pay.

The analysis made in [2] identifies 10 challenges on Cloud Computing. This proposal focuses on three of them: *data lock-in*, *scalable storage* and *scaling quickly*. We will discuss the challenges related to scalability in Section 3. Now we will see how a possible solution to the first challenge can help us to introduce mini-clouds in the architecture.

2.1. Abstracting the cloud provider

Data lock-in refers to the problem of SaaS providers of not being able to easily migrate from one cloud provider to another. This is because there is no common API for different cloud providers, and it is unlikely to expect the main corporations to agree on something like that. The Reservoir project [14] introduces a managing layer between the cloud provider and the cloud user. This layer has its own API to be used by the SaaS provider. The request for utility computing is managed in this layer which is in charge of using cloud provider's API. This adapted architecture is depicted in Figure 1.b. By abstracting the cloud provider, the application runs independently of the cloud provider behind the interface layer, reducing the problem of data lock-in. The issue of data lock-in cannot be entirely removed because it also depends on the functionality that the cloud provider can offer with its API. Even though, having a layer between cloud provider and user it is a great advantage.

2.2. Gathering mini-clouds

After abstracting the cloud provider behind the interface layer, we can replace the base of the architecture with whatever is able to provide a similar functionality of a cloud provider. Our proposal is to gather several mini-clouds from different institutions willing to collaborate in order to achieve a large amount of resources that can provide Cloud Computing to those institutions.

We consider the following scenario to motivate the possibilities of such system. Our university has a web service for students and the academic personnel to organize the ma-

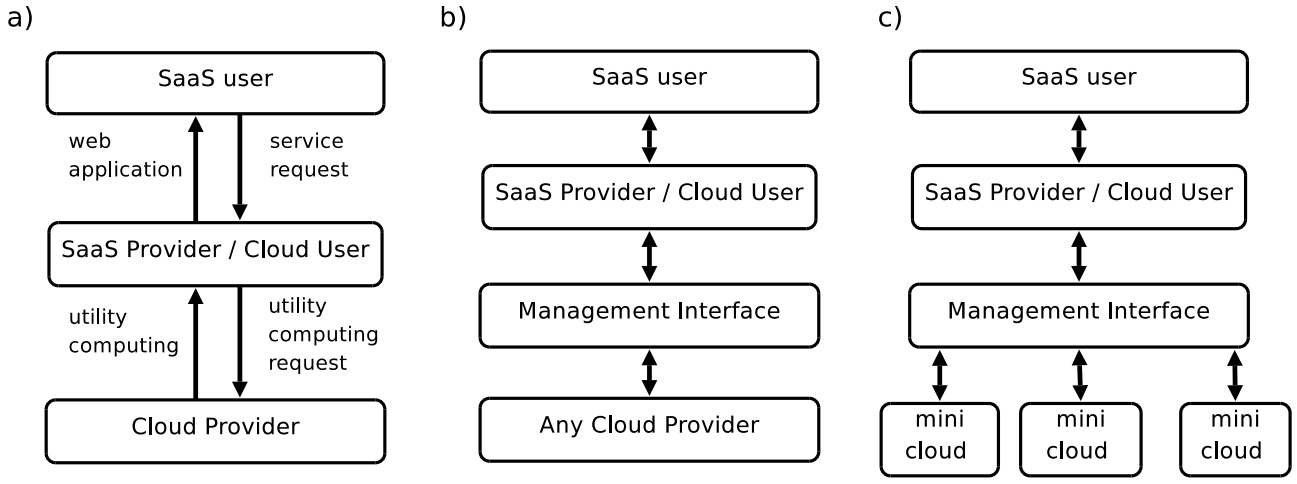


Figure 1. a) General Cloud Computing architecture with a single large cloud provider. b) Adding a managing layer that can interface any single large cloud provider. c) Replace the cloud provider with many mini-clouds.

terial and projects of every course. Having independent pages for every course has the inconvenience that each one of them has different layout scheme, navigation map, and different support for student collaboration. The web service instead, provide a platform to host every course with an equivalent scheme and functionality, so students can navigate and use it more efficiently. But having one single platform increases a lot the size of the system and the amount of users. We know that most of the time students make a light use of the service, but there are very identifiable peaks of use. For instance, there are more students visiting the courses at the beginning of the semester, but even more at the end, during the period of exams. There are small peaks when the deadline of a project is approaching and many students want to submit their files at same time. All these characteristics reflect that our scenario can be seen as regular web service that needs to optimize its resources to be able to handle peaks on demand, and to minimize the use of resources the rest of the time, when the system load is minimum. We assume that many universities have implemented their own platform to provide an equivalent service with equivalent characteristics. Since every university has already acquire the hardware to host these services, each of them can be considered a mini-cloud with limited capabilities to scale. Therefore, combining these mini-clouds to emulate a large cloud provider can increase the benefits of the everyone's infrastructure.

Figure 1.c depicts our proposal where the cloud provider is replaced by several mini-clouds. Comparing Figure 1.b with Figure 1.c we observe that it is indifferent to the SaaS provider what is providing the utility computing. Therefore, it is also possible to migrate not only from one cloud provider to a different one, but also from cloud provider to mini-clouds and vice-versa, without changing the SaaS application.

3. Self-managing interface

As we saw in the previous section, abstracting the cloud provider with an interface layer reduces the problem of data lock-in, and it allows us to introduce mini-clouds to behave as a cloud-provider. The new issue now is to deal with the higher complexity of designing the management layer with have to interface different APIs from different cloud providers or mini clouds. Using mini clouds raises also the issue of organizing distributed resources. Working with one single cloud provider is simpler because management can be done in a centralized manner, and the resources are usually in the same location, but our challenge is to organize the set of mini-clouds.

Even though the complexity is increased with the interface layer, it also gives other possibilities, specially with respect to scalability, which is part of the focus in our research. We have extensively studied structured overlay networks in the Selfman project [16], where peer-to-peer networks can scale well and quickly. Some of the networks developed in Selfman, Beernet [13, 8] and Scalaris [15, 12], provide not only self-organization of peers to deal with churn, but they also provide self-managing replicated storage. These networks are prepared to deal with unanticipated churn, because it cannot be known in advance when peers are going fail, join or leave the network.

Working with the cloud presents an important advantage with respect to churn. It is the cloud manager who decides when are the new nodes going to be aggregated, and when nodes can leave the network in order to released resources. Failures are obviously still unpredictable, but they can be more accurately detected, because the available resources are known in advance. Therefore, building a peer-to-peer network with cloud resources provides a self-organizing system with controlled churn, which can help to deal with the two of the challenges mentioned in [2]: scalable storage and scaling quickly.

3.1. Three-layer architecture

Due to the complexity of the interface layer, we identify the need for self-management in the design of it. For our proposal we use the three-layer architecture presented in [7], which is an adaptation of [5] applied to software design.

The architecture is depicted in Figure 2. At the bottom we find the *component control* layer, which communicates directly with cloud resources. This layer consists of components in charge of monitoring resources and triggering actions on them. This layer, and actually the whole architecture, is full of feedback loops [17] that constantly monitor the mini-clouds, analyze the information and decide on actions to affect the state of the cloud in order to achieve predefined goals. If we choose for a peer-to-peer architecture to organize the resources, peers are living on this layer.

The state of components running at the bottom of the architecture is reported to the *change management* layer. The interaction between these two layers can be seen as a meta feedback loop. Change management is constantly monitoring the component control to introduce changes whenever is needed. For instance, if a failure detector seems to trigger false suspicions too often, it could be reconfigured or replaced by another failure detector.

To analyze the top layer of the architecture we come back to our scenario of the web service provided by the university to administrate courses. Having logs of the web service it is possible to create a predefined plan of requesting and releasing resources from the cloud. This pre-planning would consider the schedule of the students on a daily basis, and it would take into account the yearly academic agenda to include exams periods on the demand of resources. The task of the *goal management* layer is to guarantee that the plan is going to be followed. Since pre-planning cannot be perfectly conceived, the layer must constantly monitor the system, being able to change the plan to deal with unexpected demand from the users.

Previous experience on adaptive planning systems gives us the intuition that the goal-management layer can be conceived with constraint programming. If the layer is to be

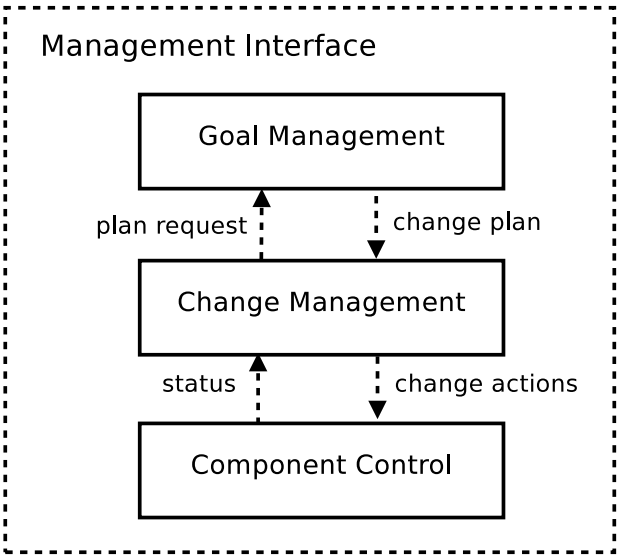


Figure 2. Three-layer architecture for a self-managing interface layer.

applied on a pay-as-you-go scheme, either for commercial cloud providers or as a strategy for fair use of mini-clouds, we can imagine many constraints such as the amount of *money* that can be spent, the maximum allowed delay to provide the service, or the resources that can be provided. Satisfying all these constraints is a constraint satisfaction problem (CSP). Trying to find an optimum way of satisfying such constraints is a constraint optimization problem (COP). Because our case study involves many entities that do not have a central point of control or global state, this lead us to a distributed constraint satisfaction problem (DCSP) and distributed constraint optimization problem (DCOP). We believe that DCSP and DCOP are the right paradigms to address the design of the goal-management layer on this three layer architecture. Furthermore, the system constantly changes the number of participants of the problem scaling up and down, and therefore, we can model it as Dynamic DCOP [11].

4. Conclusions

Research interest on cloud computing is constantly growing with different views within industry and academia. This work shares the view of a global architecture where a SaaS provider requests utility computing to a cloud provider. Such view presents many challenges from which we focus our proposal on three of them: reduce data lock-in, provide quick scalability and provide scalable storage. We start our proposal from the architecture that introduces a management interface layer between the SaaS provider

and the cloud provider, in order to make the application independent of the cloud provider. This abstraction allows as to replace the cloud provider with a set of mini-clouds that can be provided by different institutions for a global benefit.

To deal with the complexity of the middle layer we take inspiration from three-layer architecture to provide self-management. By combining these two ideas we believe that it is possible to get cloud computing out of mini-clouds. Since the resources of mini-clouds are distributed, we identify the need for decentralized management with self-organization, which can be provided by the inclusion of structured overlay networks. Our system becomes a scalable peer-to-peer network with controlled churn. We also propose the use of constraint programming solvers to achieve adaptable planning respecting the constraint on resource usage and quality of service.

5. Acknowledgments

The authors would like to thank Luis Quesada for helpful discussion on the architecture. This work has been supported by project SELFMAN.

References

[1] Amazon. Amazon web services. `http://aws.amazon.com`, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009.

[3] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[4] Distributed Systems Architecture Research Group at Universidad Complutense de Madrid. Opennebula. `http://www.opennebula.org`, 2009.

[5] E. Gat. On three-layer architectures. In *ARTIFICIAL INTELLIGENCE AND MOBILE ROBOTS*, pages 195–210. AAAI Press, 1997.

[6] Google Inc. Google app engine. `http://code.google.com/appengine/`, 2009.

[7] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[8] B. Mejías and P. V. Roy. The relaxed-ring: a fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, 2008.

[9] Microsoft Corporation. Azure service platform. `http://www.microsoft.com/azure/`, 2009.

[10] Nanodatacenters Partners. Nanodatacenters EU FP7 project media distribution. `http://www.nanodatacenters.eu`, 2009.

[11] A. Petcu and B. Faltings. Optimal solution stability in dynamic, distributed constraint optimization. In *IAT*, pages 321–327. IEEE Computer Society, 2007.

[12] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. pages 256–267. 2007.

[13] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network. `http://beernet.info.ucl.ac.be`, 2009.

[14] Reservoir Consortium. Reservoir: Resources and services virtualization without barriers. `http://www.reservoir-fp7.eu`, 2009.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[16] Selfman Partners. Self management for large-scale distributed systems based on structured overlay networks and components. http://www.ist-selfman.org, 2009.

[17] P. Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[18] XtreemOS Partners. XtreemOS: Enabling Linux for the grid. `http://www.xtreemos.org`, 2009.

A.17 Best Presentation Award: "Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT"

Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT*

Boris Mejías
Université catholique de Louvain – Belgium
boris.mejias@uclouvain.be

Abstract

The increment of network bandwidth and computing power has definitely made an impact on distributed systems which are becoming larger, more complex and therefore, difficult to manage. Although classical client-server architecture provides a simple management scheme with centralized control of the whole system, it does not scale because the server becomes a point of congestion and a single point of failure. If the server fails, the whole system collapses.

The key to deal with the complexity of large-scale distributed systems is to make it decentralized and self-managing. Peer-to-peer networks, and specially in their form of structured overlays, offer a fully decentralized architecture which is self-organizing and self-healing. These properties are very important to build systems that are more complex than file-sharing, which is currently the most common use of peer-to-peer. Despite the nice design of many existing structured overlay networks, many of them present problems when they are implemented in real-case scenarios. The problems arise due to basic issues in distributed computing such as partial failure, imperfect failure detection and non-transitive connectivity. This talk is about Beernet, a peer-to-peer network based on the relaxed-ring topology that provides cost-efficient ring maintenance without relying on transitive communication.

Fault-tolerant distributed hash tables requires some replication mechanism so as to deal with the failure of a peer without loosing data. Maintaining the replicas is not just costly but it is also difficult to guarantee their coherency. Beernet uses a transactional protocol based on Paxos consensus algorithm over symmetric replication that guarantees that at least the majority of the replicas is kept coherent. The transactional layer is adapted to provide synchronous and asynchronous collaboration between peers at the application level.

*This research is mainly funded by project SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

XtreemOS

Enabling Linux for the Grid

XtreemOS Summer School 2009

Best Presentation Award in the Doctoral Symposium

is hereby granted to

Boriss Mejias

By the presentation titled

"Beernet: a relaxed-ring approach for P2P networks with transactional replicated DHT"

Franz J. Hauck
Doctoral Symposium Coordinator

Alvaro Arenas
XtreemOS Summer School Chair

Oxford, September 10th, 2009

Information Society Technologies

A.18 Wiki credibility enhancement

Wiki Credibility Enhancement*

Felix Halim, Wu Yongzheng, Roland Yap
School of Computing
National University of Singapore
13 Computing Drive
Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

ABSTRACT

Wikipedia has been very successful as an open encyclopedia which is editable by anybody. However, the anonymous nature of Wikipedia means that readers may have less trust since there is no way of verifying the credibility of the authors or contributors. We propose to automatically transfer external information about the authors from outside Wikipedia to Wikipedia pages. This additional information is meant to enhance the credibility of the content. For example, it could be the education level, professional expertise or affiliation of the author. We do this while maintaining anonymity. In this paper, we present the design and architecture of such system together with a prototype.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Web-based services

General Terms

Security, Standardization, Verification

Keywords

Credibility, login, Wikipedia, OpenID, anonymity

1. INTRODUCTION

Wikipedia is perhaps one of the most successful efforts to create collaborative content. It is an encyclopedia covering a wide range of knowledge to exploit the "wisdom of the crowds" and to which anybody can contribute. Arguably, the success of Wikipedia is due to its open and self-policing nature. Anonymity is also a key feature – anybody can create an online persona with an account, or alternatively, the IP address is used.

One of the criticisms of Wikipedia is that the material is written by "anonymous strangers of unknown qualifications" [1]. Consider an entry on a technical subject, say a medical article, one might prefer an article written by a qualified physician. In this paper, we propose to enhance the credibility of the information contained in Wikipedia.

Consider the following scenario. The ACM maintains a comprehensive library of computer science publications with author information. If a contributor to a computer science Wikipedia article has credentials such as published in x ACM conferences and journals or affiliation being MIT, this information can increase the credibility of an author. We call such information, *credibility information*. In Wikipedia, authors are identified by login id or IP address, but as anybody can make one or more login ids, the login information of an author does not by itself lend credibility. Rather, we want to be able to make use of other information from credible and trusted sources outside Wikipedia to transfer credibility information into Wikipedia. In the ACM example, anonymity could be retained while asserting a statement like published papers in CACM.

Unlike Wikipedia, Google Knol [2] attempts to provide credibility information. In Knol, the credibility of the articles is based on the name of the author which can be certified by credential providers such as credit card companies or manually by phone. The verification mechanism is proprietary to Knol. Furthermore, it means that the author cannot be anonymous. Essentially, name verification tells one that a certain individual with a particular name as certified by Google contributed the article. However, the name by itself may not be very credible with the exception of well known authors. However, ambiguity still exists since several individuals could have the same name. For example, a Wikipedia author with pseudonym Essjay [3] claimed to be a (bogus) tenured professor who taught theology. Such an incident could also take place in Knol since a valid real name does not provide information about expertise or profession (i.e. professor of theology).

In this paper, we propose a simple extension to Wikipedia (and MediaWiki) which enhances the information in Wikipedia to make it more credible by automatically using credibility information from trusted third parties. Our extension maintains the open and anonymous nature of Wikipedia. We transfer information from trusted third parties and associate that securely with the text written by the author. We have implemented a prototype which utilizes the MediaWiki tag extension together with OpenID [4] as either an authentication or credibility provider although other credibility providers could also be used. Some scenarios where we can enhance Wiki:

*This work was supported by SELFMAN (contract: 034084)

WikiSym '09, October 25-27, 2009, Orlando, Florida, U.S.A.

- Verifying the author's name: A credit card provider such as Visa can certify that the author is a human and optionally his/her actual name. This gives a Knol-like flavor to Wikipedia. It can also help to make it more difficult for robots to edit Wikipedia.
- Verifying the anonymous membership with an organization: A provider like ACM can provide university or expertise credentials for an author without his/her personal identity.
- Restricting anonymous voting system: A credibility provider can be used to restrict the voting system in Wikipedia [5] to from certain voters without disclosing the name of the voters.
- Other services: can enhance wiki articles by giving information about the author while preserving the anonymity of the author.

2. DESIGN GOALS

Before discussing the design of the credibility enhancement for Wikipedia, we first give our design objectives:

- **Credibility:** The purpose of the credibility enhancement is to enable Wikipedia to show some external trusted information about the authors. Such information could be the authors' real name, professional affiliations, proof of identity, etc., essentially anything which can give additional credibility to the text in an article. This information has to be verified so that authors cannot easily provide false information. We also want to avoid an author stealing other author's identity to publish/edit pages.
- **Anonymity:** We want to preserve the capability of authors to be anonymous if they want to, i.e. we do not want Knol [2] which requires that the real names of users be verified. Furthermore, we want to ensure that users' private data is not stored in Wikipedia, so that even if Wikipedia is compromised, users' private data will not be exposed.

  There is a trade-off between credibility and anonymity. Authors sometimes want to be anonymous, but that means their statements/edits may be less credible. Less credible edits are more likely to be deleted by Wikipedia administrators. We give the author the freedom of balancing the trade-off and provide different levels of credibility information.
- **Ease of Use:** The enhancement should not make Wikipedia much harder to use, e.g. forcing authors to download and run some software on their local machine is inconvenient and should be avoided.

We remark that the credibility information in our proposal is independent of reputation. We preserve reputation [6] on any edits, and, reputation can be linked to the author's credibility as well.

3. PROTOCOL DESIGN

The credibility extension involves four components including the author which work together as shown in Fig. 1 *C1-4*. *C1* is the Wikipedia web server with our credibility extension installed. *C2* is the credibility proxy. We suggest it

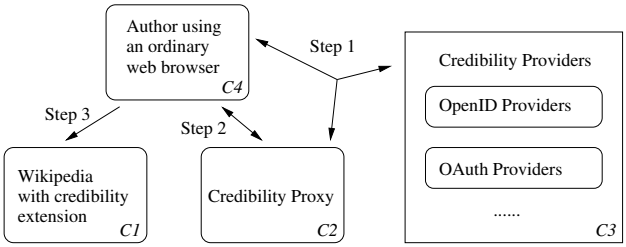


Figure 1: Components and work flow of the credibility extension.

be run in a different host to prevent author's credential being compromised in case *C1* is compromised. The Wikipedia web server stores a certificate of the proxy so that Wikipedia can verify signatures generated by the proxy. Note that it is possible to have more than one proxy, but we use one for the illustration purpose.

C3 is one or more credibility providers. The credibility providers give credible information specified by the author to the credibility proxy. They communicate with the credibility proxy using the respective supported protocol. For example, the OpenID protocol needs three-way communication among the author, OpenID server, and credibility proxy. *C4* is the Wikipedia author using an ordinary web browser.

There are three main steps to get a credible edit in a Wikipedia page:

- **Step 1: acquiring author information**
  In the case of OpenID [4] or OAuth [7] protocol, this step involves three-way authentication. After this step, the credibility proxy should have the author's information. This step can be performed multiple times to get information from multiple providers.
- **Step 2: sign**
  The author selects the appropriate author information (see Fig. 2) to be passed to Wikipedia and enters the text to be published in Wikipedia. The credibility proxy signs the author's information together with the text and generates the signed text, see the screen shot in Fig. 2.
- **Step 3: edit page**
  The author pastes the signed text to Wikipedia (shown in Fig. 3). Note that the author does not have to login to Wikipedia in order to use the credibility extension. The signed text can be published elsewhere on the web and someone else can enter the signed text into Wikipedia. It can also be copied between pages.

When the edited page is viewed, the credibility extension verifies that the edit has been signed correctly using the credibility proxy's certificate. If the edit is verified, the author's information will be displayed — this can be done in various ways, e.g. as in Fig. 4. Our credibility extension is compatible with caching which is important for Wikipedia performance, the signed text does not have to be verified every time it is viewed.

The trust relationships among the four components are:

- Wikipedia trusts the credibility proxy to sign the correct information. Wikipedia also trusts that the proxy's key is not compromised.

- The authors trust the credibility proxy to only release information which they authorize. Note that the information can be filtered by the credibility providers before it is given to the proxy, so the ideal case is that the proxy *only* knows the information to be signed and released. However, some information such as the user ID in the OpenID server and user's IP address are always known to the proxy.

- The credibility proxy does *not* have to trust the credibility providers because the providers' name will be shown together with the signed text. We leave the Wikipedia readers to decide whether to trust the providers or not but Wikipedia could choose to trust predefined providers so as to be able to conveniently display them in the Wikipedia article.

- The authors implicitly trust the credibility providers which are chosen by them.

4. CREDIBLE WIKI PROTOTYPE

We describe a credible wiki prototype to illustrate our ideas. It consists of a credibility proxy and a MediaWiki extension. Our prototype employs the OpenID 2.0 framework [8] to communicate between the credibility proxy and third party credibility providers to share information about the particular user. However, other protocols could also be used. The proxy anonymizes the user information selected by the user and signs it along with the text. Wikipedia only needs a lightweight extension to check the signature of the text sent by the proxy. If the signature matches, it will be published along with the assigned credibility information. Otherwise no special credibility will be given to the text.

4.1 The Credibility Providers

Credibility providers are the source of the additional information for the authors to enhance the credibility of their edits. Recently, `http://www.myid.is` provides a service to certify a digital identity online which is similar to what Knol uses for author name verification. One can imagine a variety of credibility providers to provide a variety of information which could include public and private organizations. The information would be some property associated with the author such as professional association, real name verification, geographic location or country, etc.

The credibility provider must have a protocol to share information to the credibility proxy or any other consumer. We observed that OpenID [4, 8] and OAuth [7] are the two most promising open protocols to be used widely for managing the online identity and sharing information.

OpenID provides a decentralized open standard for user authentication and access control. The user only needs to setup one digital identity on an OpenID provider to gain access to other systems. We take advantage of an OpenID provider not for login but as a way of transferring information about a digital identity, so we use an OpenID provider as a credibility provider.

Our examples with our prototype use a free OpenID provider (`myopenid.com`) as the credibility provider. Since there is no particular trust associated with `myopenid.com`, the information in the examples is only illustrative.

4.2 The Credibility Proxy

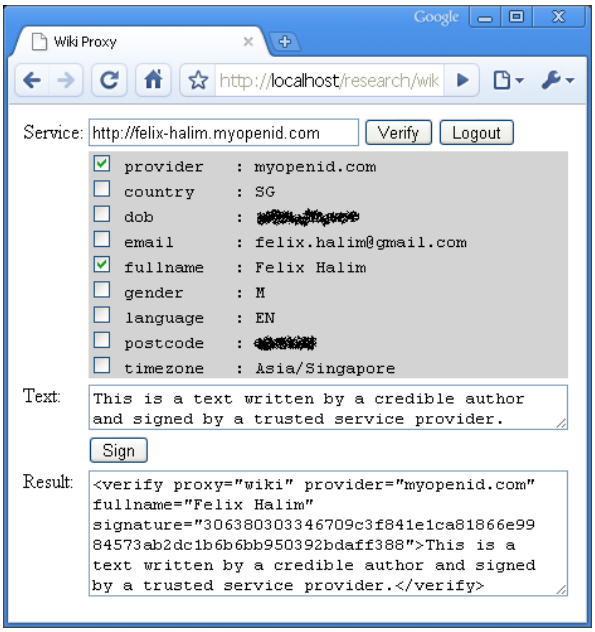


Figure 2: A Proxy for Wikipedia.

Fig. 2 shows our prototype. The service field is filled with the URL of the credibility provider. The gray area is the user information retrieved from the OpenID credibility provider. The *text* area is the text that will be signed by the proxy together with selected user information. The example chooses to include the provider and the full name to be signed with the text. The *result* area is the ready to use wiki text that can be inserted anywhere in a wiki page.

We allow the author to select which information from credibility providers to be attached. This information should be thought of as credibility attributes to be attached to the edit. Wikipedia could have a policy to require certain attributes from trusted credibility providers in order to achieve a certain category of credibility. For example, to get a credibility of a "scientist", the author has to include information such as: institution, position, and perhaps information about publications (as in the ACM example in Sec. 1).

4.3 Wikipedia Extensions

MediaWiki is the software behind Wikipedia. MediaWiki can be extended using extensions such as tag extensions, parser functions, special pages, or template extensions. We implement our credible wiki using a tag extension which we call the *verifier extension*.

4.3.1 Wiki Verifier Extension

The text signed by the credibility proxy can be put inside any page in Wikipedia (as well as outside Wikipedia since verifying the signature can be easily done with the certificate of the credibility provider). We created a verifier extension tag to check that the text and additional attributes inside the tag have been signed by the proxy.

There are three mandatory items and several optional attributes within the verifier tag extension:

- **proxy**: the name or the public key of the proxy. Wikipedia will be able to verify the signed text by having

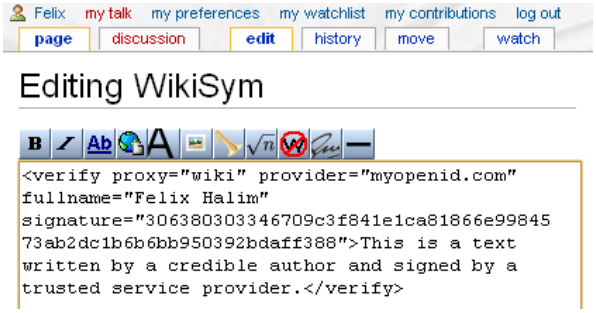


Figure 3: Verifier tag extension for Wiki.

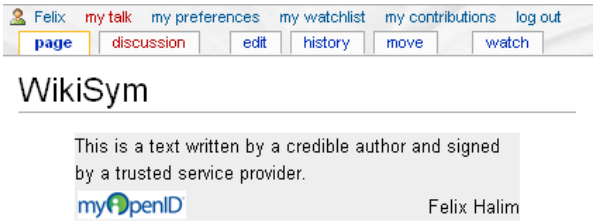


Figure 4: The end result in Wikipedia page.

a list of trusted proxies and their certificates.

- **signature**: the signature of the text inside the tag. The signature should match with the digested text decrypted using the public key of the proxy.
- **text**: the text to add or edit.
- **optional attributes**: such as provider, full name, country, email, etc. can be included as the attribute of the verifier tag. Wikipedia then can use the additional information to display the text.

Fig. 3 shows an example of a verifier extension tag. The content of the signature attribute contains the signed digest of the information in the verify tag. If any of the text or attribute values are changed, the verify tag will treat the text content as regular text rather than as credible text.

Credible text in a Wikipedia page should be presented in a way which can show its credibility information. While there are many ways of doing the presentation, Fig. 4 shows displaying credible text by graying the background. The displayed paragraph with grayed background provides the "context" for the author when editing a paragraph in Wiki. The idea of a context is to make it harder to abuse the credible text (i.e. placing the text in different paragraph or articles that have different context to get different meanings from the same text).

The display of the text can be improved further with more credibility information (other than 8 fields in Fig. 2). For example, a badge-like display can be used to annotate the text with particular properties to be associated with user information matching a Wikipedia credibility category, e.g. "computer scientist".

The presentation of credible text, shown in Fig. 4, changes the flow of text, thus it may not be scalable when there are many small edits, where each sentence in a paragraph is edited by a different author over time. More sophisticated GUIs can be added to present credible text without changing the flow, for example using JavaScript to highlight any credible text upon mouse-over. The credibility provider logo and other credibility information can be listed after the main text which is similar to how citations are handled in Wikipedia.

4.3.2 Wiki Poll Extension

The MediaWiki poll extension [5] can also benefit from the credibility extension. Currently, the poll extension stores the IP address and Wikipedia user name pair as the poll account to vote for the poll. If the user does not have a Wikipedia account then only the IP address will be used to vote. The poll doesn't allow duplicate votes for each poll account.

Credibility allows recording additional information about the pool participants. Alternatively, we may want to restrict the participants of the poll by only accepting, for example, users from a particular country. This can be done by requiring a "country" field from a trusted credibility provider (other information could be hidden).

5. DISCUSSION

Wikipedia accumulates information through the efforts of anonymous contributors and volunteers. While this is democratic, it has a weakness that the information may be perceived as being less credible (regardless of whether or not it is actually so). Normally, Wikipedia uses external citations to add credibility to the information entered. However the citation may either not be available or not easily accessible (confidential). The text might also simply be just words of wisdom from an expert author but it is hard to convince the readers that the text they are reading has a certain quality as it may lack sufficient citation.

Well known authors usually have credibility information outside Wikipedia. Our enhancement allows them to transfer the rich information about the author available from the third party credential provider to Wikipedia. Our enhancement can be seen as a complement to the citation mechanism. It is important to note that, in the process of transferring the author information, we can maintain the anonymity of the authors which is consistent with the philosophy of Wikipedia and serves to protect the authors.

Our credibility mechanism can be used to enhance any reputation mechanism. It may be also used by administrators to manage edits.

6. REFERENCES

[1] P. Denning, J, Horning, D. Parnas and L. Weinstein, "Wikipedia Risks", Comm. of the ACM, 48(12), 2005.
[2] "Knol", `http://knol.google.com/k`.
[3] "Essjay Controversy", `http://en.wikipedia.org/wiki/Essjay_controversy`.
[4] `http://openid.net/`.
[5] `http://www.mediawiki.org/wiki/Extension:Poll`.
[6] B.T. Adler, K. Chatterjee, L. de Alfaro, M. Faella, I. Pye and V. Raman, "Assigning Trust to Wikipedia Content", WikiSym, 2008.
[7] `http://oauth.net/`.
[8] D. Recordon and D. Reed, "OpenID 2.0: A Platform for User-Centric Identity Management", Digital Identity Management, 2006.

A.19 Routing in the Watts and Strogatz Small World Networks Revisited

Routing in the Watts and Strogatz Small World Networks Revisited

Felix Halim, Yongzheng Wu and Roland H.C. Yap
School of Computing
National University of Singapore
13 Computing Drive, Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

I. Introduction

The study of small-world networks (SWN) has become popular with the growth of the World Wide Web (or simply Web) and more recently with the rising growth of social networking sites such as Facebook. The idea of a small world can be simply described as between any two people there is only a short chain linking them through their acquaintances. SWNs was first studied in the pioneering work of Stanley Milgram [3] who showed experiments forwarding letters that the length of the chain was between five and six. This is also popularly known as "six degrees of separation".

SWNs have been applied in many contexts ranging from social science, biological science to mathematics and computer science. SWNs are also particularly relevant to the web. In the context of this paper, we are interested in SWNs applied to problems in networking, namely, routing in networks, particularly in peer-to-peer systems.

The small world phenomena suggest that a graph modeling a small world should have a small diameter. Furthermore, SWNs are not just a random graph but they have structure. A well studied model of a SWN is the one proposed by Watts and Strogatz [2]. We will abbreviate the Watts and Strogatz model as WS-SWN. WS-SWN has the virtue of simplicity, while capturing structure in various networks and possessing low diameter and has been shown to capture the structure real small-world networks.

SWNs have been used in large scale self-management systems for example as an alternative to structured overlay networks because of their routability and flexibility. They have been used to balance load [6], reduce maintenance cost of the network [9], and efficiently disseminating data [8], [7]. When the SWN is used as an overlay network, navigability becomes an important issue.

It is usual to consider a network to be *navigable* if there is a routing algorithm where the (expected) routing length has a polylogarithmic bound (in the number of nodes). In addition, the routing algorithm should not need to know global information about the whole graph (a decentralized routing algorithm).

The most popular SWN models are those based on the Kleinberg model because of their navigability. However, it may not be the best model to represent a self managing social network since a social network typically has more localized links, i.e. a high clustering coefficient. WS-SWN gives a better fit to small networks arising from social networks as it has parameters to adjust the clustering coefficient and the average shortest path length.

In this position paper, we revisit the question of *navigability* in WS-SWN. Firstly, the WS-SWN makes it easy to construct SWNs with different amounts of clustering making it a useful model for real SWNs. Secondly, WS-SWN is not navigable using pure greedy routing. Our initial experiments suggest that by considering other routing algorithms, WS-SWN may be navigable. It is at least more navigable than might be expected.

II. Background

Watts and Strogatz proposed a method to generate a graph with small world-like properties: large mean *clustering coefficient* and small mean shortest path length. The clustering coefficient measures the level of clustering among the neighborhood of a node. The larger the coefficient, the higher the clustering. Let k_i be the degree of node i, and e_i be the number of edges among the nodes directly connected to node i. The clustering coefficient C_i of node i is defined as $C_i = 2e_i/k_i(k_i - 1)$. Since e_i is never larger than $k_i(k_i - 1)/2$, the clustering coefficient is a fraction between 0 and 1 (when the neighborhood forms a clique, we get the maximum clustering coefficient of 1).

We briefly describe WS-SWN here. Given three parameters: n, k and p, where $1 \ll ln(n) \ll k \ll n$ and $0 \leq p \leq 1$, the algorithm constructs an undirected graph of n nodes with average degree k. First, construct a ring of n nodes, each connected to k neighbors, $k/2$ on each side. Next, for each edge, rewire it with probability p. Rewiring is done by replacing the edge with an edge uniformly randomly distributed in the graph.

Kleinberg proposed another SWN model. We give a 1-D example. First, construct a ring of n nodes, each connected to its two neighbors. These edges are called base connections. Next, for each node, randomly add q edges to another node where the length of the edge follows power-law distribution so that there are more shorter edges than longer edges. These edges are called long links.

Kleinberg shows that this network model is navigable using greedy routing, and the expected routing distance (number of hops) is $O(log^2 n)$. The greedy routing algorithm simply

chooses the neighbor which has the smallest lattice distance to the destination as the next hop. Because of the base connection, a node with smaller lattice distance can always be found and the algorithm always terminates. The Kleinberg result also shows that WS-SWN is not navigable using greedy routing.

III. Improving Local Routing Algorithms on WS-SWN

We will consider algorithms which are not strictly greedy routing but can still be considered as local routing algorithms which are greedy-routing like. The motivation is to get better navigability in WS-SWN. We investigate a NoN-Greedy [4] routing strategy. In NoN-Greedy routing, each node knows the link information of its neighbors (1-lookahead). With more (but still local) information, the idea is that the routing algorithm can be better guided towards the target and avoid wrong decisions which make lead to backtracking.

Fig. 1 illustrates the problem with greedy routing on WS-SWN. The graph generated using the WS-SWN procedure has $p = 0.1$, $n = 1000$, and $k = 10$. The source node is the green square with one circle. The target node is the blue square with two concentric circles. As the routing starts from the source, it quickly approaches the target node region in few hops. However, greedy routing has difficulty routing to the target node although it's very close. This is because only some particular "gateway" nodes can lead to the target node, and it is hard for the greedy routing to find them. In this example, greedy routing fails after taking more than 100 hops.

The insight in this paper is that an improved routing algorithm which still works as a local algorithm makes it easier to find the "gateway". Fig. 2 shows exactly same graph as in Fig. 1 but with a different route. Each node has link information contained in its neighborhood (1-lookahead). The cyan squares denotes nodes which are one step away from the target (the neighborhood of the target) while the magenta squares denotes 2 steps away from the target (the 1-lookahead neighborhood of the target). Here, we can see that if the routing lands on one of the cyan or magenta nodes, it can reach the target immediately if the additional neighborhood information is available. In the figure we see that the last 2 hops to the target are guided by the 1-lookahead by first visiting the magenta square which then guides the routing to the cyan square and then directly to the target node.

1-lookahead dramatically increases the likelihood of the greedy routing to find the target node. Without 1-lookahead guidance, the route took more than 100 hops. Using 1-lookahead, the routing completes in 13 hops. As we shall see later, adding more edges also helps in widen the gates thus making the greedy routing less likely to be stuck in a dead-end.

IV. Preliminary Results

We now present some preliminary routing experiments on WS-SWN graphs. All the experiments are run using 10^5 nodes and $p = 0.1$.

A. The Effect of Lookahead on WS-SWN

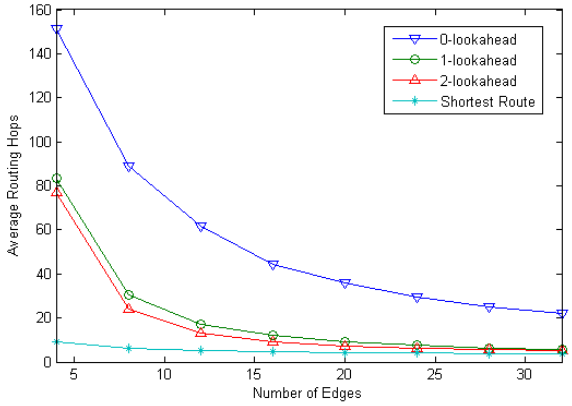


Fig. 3. Average Routing Hops vs. Number of Edges for Lookahead Variants

In our experiments, we also include vanilla greedy routing which is equivalent to 0-lookahead routing. Fig. 3 shows the improvement achieved by having a 0,1, and 2 lookahead strategy versus the average number of edges per node (average degree of each node). The shortest-route shows the average shortest path lengths to route between any two nodes. 1-lookahead is a greedy routing strategy that can see the neighbor's friends (or links), thus, it has more vision than 0-lookahead. 2-lookahead has another level of vision over 1-lookahead. As we can see, employing the lookahead reduces the bottleneck effect of greedy routing on WS-SWN (which tends to circle around the target node). The gain of having 1-lookahead is far greater than having 2-lookahead. Since 2-lookahead is expensive (in space), it may not be practical in real world situations. As such, we may want to limit ourselves to 1-lookahead. This experiment shows that the increase of the number of edges gives the biggest advantage when it approaches $O(log(n))$ number of edges. Beyond that, the benefit becomes less. We see in this experiment that 1-lookahead can approach optimal routing when the number of edges is more than $log(n)$.

We observed that WS-SWN may not be as "un-navigable" as thought. By having more (local) information than just what a greedy routing uses (such as 1-lookahead), it is possible to improve the routing performance further. Our position is this paper is to ask whether there exist better distributed algorithms that can achieve good routing performance for WS-SWN or even a random network without having the cost of 1 or 2 lookahead to make them more navigable.

B. Kleinberg versus WS-SWN

The Kleinberg model which based on power-law edge distribution has been shown to be navigable and the WS-SWN is generally described as being non-navigable [1].

Fig. 4 shows the average routing hops versus the number of edges of a Kleinberg-1D SWN and WS-SWN. We allow up to 1-lookahead in both models. The poor routing performance

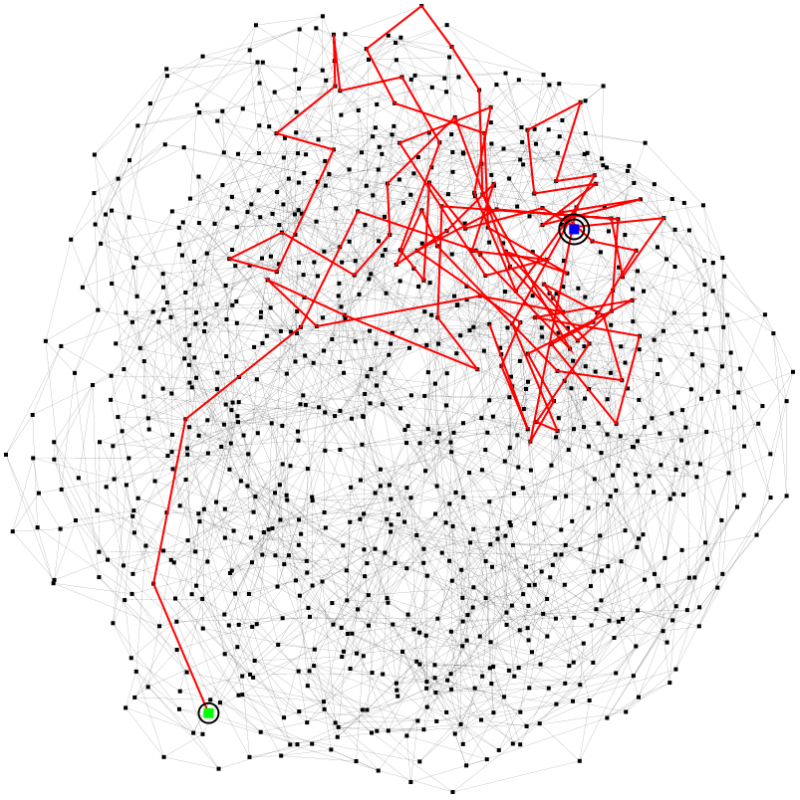


Fig. 1. Routing is harder when it's closer to the target

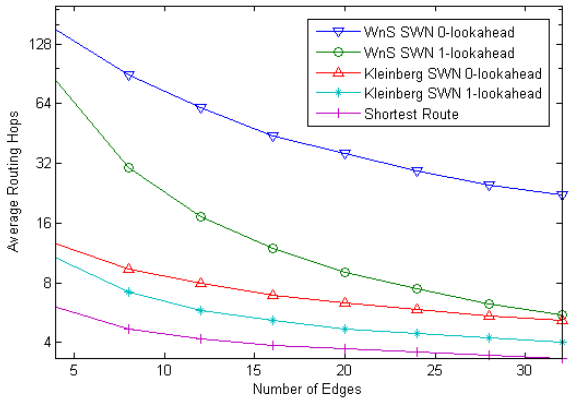


Fig. 4. Kleinberg versus WS-SWN

of greedy (0-lookahead) on WS-SWN is readily apparent. But as the number of edges increases 1-lookahead gets close to Kleinberg with greedy routing. We see also that the Kleinberg SWN doesn't benefit much from 1-lookahead [5].

In terms of the graph structure, a WS-SWN graph is more clustered than a Kleinberg graph. WS-SWN employs a parameter p which determines the clustering coefficient and the diameter of the graph [2]. The p values that give high clustering coefficient and low diameter is from 0.01 to 0.5. A Kleinberg SWN typically has a clustering coefficient of 0.01. In this case, WS-SWN model more closely resembles real world social networks which typically has high clustering coefficient (e.g. consider the students in the same course, they would form a few cliques). We conjecture that a good routing strategy for WS-SWN model might give clues how to design routing algorithms for real social networks.

V. Conclusion

Routing in SWNs have mainly been studied in the context of Kleinberg-like SWNs because of their navigability. If one wants to employ real world SWNs such as social networks for self-managing overlay networks, we believe that models like WS-SWN may be more suitable because the resulting graphs from the WS-SWN construction have properties closer to real world SWNs. Furthermore, WS-SWN provides a parameter p to adjust the clustering coefficient and the diameter of the graph.

The drawback is that WS-SWN is not navigable using greedy routing. We demonstrate some preliminary experiments which suggest that WS-SWN may be more navigable than previously thought. Using additional routing information (such as 1-lookahead), routing performance in WS-SWN seems to approach greedy routing performance in the Kleinberg SWN model. This naturally leads to the question of what other

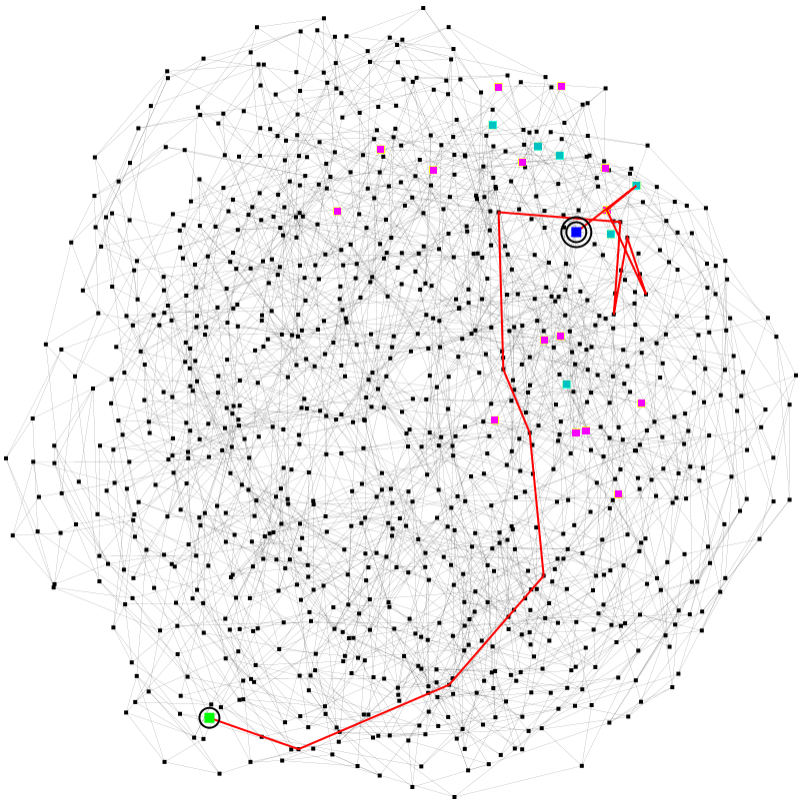


Fig. 2. Improving the routing

strategies can be used to improve the routing further in WS-SWN and actual instances of real world SWNs. This is the subject of further work.

ACKNOWLEDGEMENTS

This work was supported by SELFMAN (contract: 034084)

REFERENCES

[1] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective", *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 163–170, 2000.

[2] D. Watts and S. Strogatz, "Collective Dynamics of small-world networks", *Nature* 393,440, 1998.

[3] J. Travers and and S. Milgram, "An Experimental Study of the Small World Problem", *Sociometry*, Vol. 32, No. 4, 425–443, 1969.

[4] M. Naor and U. Wieder, "Know thy Neighbor's Neighbor: Better Routing for Skip-Graphs and Small Worlds", *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, Vol. 3279, 269–277, 2004.

[5] C. Martel and V. Nguyen, "Analyzing Kleinberg's (and other) Small-world Models", *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, 179–188, 2004.

[6] Sarunas Girdzijauskas, "Oscar: Small-world overlay for realistic key distributions", *Proceedings of the 4th International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 247–258, 2006.

[7] G.S. Manku, M. Bawa and P. Raghavan, "Symphony: distributed hashing in a small world", *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 127–140, 2003.

[8] Sarunas Girdzijauskas, Gregory Chockler, Roie Melamed and Yoav Tock, "Gravity: An Interest-Aware Publish/Subscribe System Based on Structured Overlays", *The 2nd International Conference on Distributed Event-Based Systems*, 2008.

[9] Sarunas Girdzijauskas, Wojciech Galuba, Vasilios Darlagiannis, Anwitaman Datta and Karl Aberer, "Fuzzynet: Zero-maintenance Ringless Overlay", *Technical report*, 2008.

Bibliography

[1] Artur Andrzejak. Generic self-healing via rejuvenation: Challenges, status quo, and solutions. In *Workshop on Architectures and Languages for Self-Managing Distributed Systems (SELFMANSASO)*, San Francisco, California, USA, September 2009.

[2] Denise Anthony, Sean W. Smith, and Tim Williamson. The Quality of Open Source Production: Zealots and Good Samaritans in the Case of Wikipedia. Technical Report TR2007-606, Dartmouth College, Computer Science, 2007.

[3] Cosmin Arad, Jim Dowling, and Seif Haridi. Building and evaluating P2P systems using the Kompics component framework. In *Proceedings of the Ninth IEEE International Conference on Peer-to-Peer Computing (P2P'09)*, pages 93–94, September 2009.

[4] O. Babaoglu, M. Jelasity, A.M. Kermarrec, A. Montresor, and M. van Steen. Managing clouds: a case for a fresh look at large unreliable dynamic networks. *Operating Systems Review*, 40(3), 2006.

[5] Facebook. Facebook statistics. `http://www.facebook.com/press/info.php?statistics`, 2009.

[6] Felix Halim, Yongzheng Wu, and Roland H.C. Yap. Routing in the watts and strogatz small world networks revisited. In *Workshop on Architectures and languages for self-managing distributed systems (SELFMAN@SASO09) at the Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2009.

[7] Felix Halim, Wu Yongzheng, and Roland H.C. Yap. Wiki credibility enhancement. In *Fifth International Symposium on Wikis and Open Collaboration (WikiSym)*, 2009.

[8] Mikael Högkvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schütt. Using global information for load balancing in DHTs. In *SASO SELFMAN Workshop*, October 2008.

[9] Mikael Högqvist and Nico Kruber. Active/Passive Load Balancing with Informed Node Placement in DHTs. In *Proceedings of the International Workshop on Self-Organizing Systems (IWSOS 2009)*. Springer, December 2009.

[10] Mikael Högqvist and Stefan Plantikow. Towards Explicit Data Placement in Scalable Key/Valaue-stores. In *Workshop on Architectures and Languages for Self-Managing Distributed Systems (SELFMAN@SASO09)*, September 2009.

[11] 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.

[12] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. *32nd ACM Symposium on Theory of Computing*, 2000.

[13] Nico Kruber. Dht load balancing with estimated global information. Master's thesis, Humboldt-Universität zu Berlin, 2009. licensed under the Creative Commons Attribution-Share Alike 3.0 Germany License (see http://creativecommons.org/licenses/by-sa/3.0/de/).

[14] Boris Mejías. Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT. Doctoral Symposium at the XtreemOS Summer School 2009, Wadham College, University of Oxford, Oxford, September 2009.

[15] Boris Mejías. Beernet: The relaxed beer-to-beer network. Université catholique de Louvain, 2009. Available at `beernet.info.ucl.ac.be`.

[16] Boris Mejías, Alfredo Cádiz, and Peter Van Roy. Beernet: RMI-free peer-to-peer networks. In *DO21 '09: Proceedings of the 1st International Workshop on Distributed Objects for the 21st Century*, pages 1–8, New York, NY, USA, 2009. ACM.

[17] Boris Mejías and Peter Van Roy. The relaxed-ring: A fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, September 2008.

[18] Boris Mejías and Peter Van Roy. From mini-clouds to Cloud Computing. In *Workshop on Architectures and Languages for Self-Managing Distributed Systems (SELFMAN@SASO09)*, September 2009.

[19] Moni Naor and Udi Wieder. Know thy neighbor's neighbor: Better routing for skip-graphs and small worlds. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[20] The Open Handset Alliance. Android, 2007. Available at `www.android.com`.

[21] Programming Languages and Distributed Computing Research Group, UCLouvain. P2ps: A peer-to-peer networking library for mozart-oz. `http://gforge.info.ucl.ac.be/projects/p2ps/`, 2008.

[22] Oskar Sandberg. Distributed routing in small-world networks. In *Eighth Workshop on Algorithm Engineering and Experiments (ALENEX06)*, 2006.

[23] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. Technical Report ZR-09-28, Zuse Institute Berlin, September 2009.

[24] Thorsten Schütt, Alexander Reinefeld, Florian Schintke, and Christian Hennig. Self-adaptation in large-scale systems: A study on structured overlays across multiple datacenters. In *Architectures and Languages for Self-Managing Distributed Systems (SELFMAN@SASO09)*, September 2009.

[25] Thorsten Schütt, Alexander Reinefeld, Florian Schintke, and Marie Hoffmann. Gossip-based Topology Inference for Efficient Overlay Mapping on Data Centers. In *P2P*, 2009.

[26] Bongwon Suh, Gregorio Convertino, Ed H. Chi, and Peter Pirolli. The singularity is not near: Slowing growth of wikipedia. In *The Fifth International Symposium on Wikis and Open Collaboration (WikiSym)*, 2009.

[27] Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[28] Jeffrey Travers, Stanley Milgram, Jeffrey Travers, and Stanley Milgram. An experimental study of the small world problem. *Sociometry*, 32:425–443, 1969.

[29] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature: Macmillan Publishers Ltd*, 1998.