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

Deliverable reference number and title:	D.2.1a Report on basic computation model
Due date of deliverable:	July 15, 2007
Actual submission date:	July 15, 2007
Start date of project:	June 1, 2006
Duration:	36 months
Organisation name of lead contractor for this deliverable:	UCL
Revision:	1
Dissemination level:	PU

Contents

1 Executive summary

A self-managing system must adapt itself to many different kinds of environmental changes, of which the most important are faults, attacks, performance hotspots, configuration and software updates. All these changes interact, often in unexpected ways. We would like to build a self-managing system that behaves in the appropriate way. This is a design problem; we would like to build the system so that it is self-managing by design. Trying to make a large system self-managing after the fact is very difficult and prone to many unexpected errors. Our experience shows that adding self management requires explicit design decisions and often needs new algorithms. To make this kind of design tractable, we would like to know the rules and techniques to make systems self-managing by design. By analogy with object-oriented programming, we would like to find the rules we must follow so that the system has the desired structure and behavior. We find that the main principle is the pervasive use of feedback loops. Each part of the system must observe itself and correct deviations from correct behavior. This way of designing systems covers all forms of self management: fault tolerance, security, adaptability, reconfiguration, and performance tuning. Systems consist of a set of interacting feedback loops. We study how feedback loops interact and how we can obtain desired interactions without unpleasant surprises.

This report studies how to build software systems with feedback loops. We study several different kinds of systems, starting with biological systems and including software systems, to understand the main principles of designing with feedback loops. The area of control theory also studies design with feedback loops. We find that designing software systems with feedback is quite different from what is done in control theory. There are three main differences: nonlinearity, scale, and dynamicity. The first difference is that software systems are highly nonlinear: a single bit error can cause a catastrophic change in behavior. The second difference is the scale: a software system consists of a large number of interacting feedback loops. The third difference is the dynamicity: software systems change frequently: by adaptation, by reconfiguration (through software updates), and by human intervention.

We start our study of how to design such systems by looking at existing systems that are built in this way. We look at biological systems, which are highly dependent on feedback loops. We also look at software systems such as network transport protocols and structured overlay networks which use feedback loops to adapt themselves to their environments. We reconstruct the designs of these systems in terms of feedback loops. The structured overlay network is directly relevant to the project: we will use it as a foundation for the next stage.

This report concludes with several design rules for building systems with feedback loops. A first rule is to design systems based on convergence principles: each feedback loop should enforce a convergence principle. A second rule is to use the system instead of bypassing it. That is, consider a feedback loop as an encapsulated abstraction that provides a service, instead of a set of parts. A third rule is to use management instead of stigmergy to interact with a feedback loop. That is, control the loop directly instead of modifying the system that the loop observes. A fourth rule is to use local instead of distributed feedback loops. At this stage of the project, these rules remain mostly intuitive; we expect to put them on more solid theoretical grounding later in the project.

We have achieved some understanding of how to program with feedback loops. To make our methodology more concrete we need to make the design rules more precise. We also have to understand how the feedback loops fit within the large-scale behavior of the system. At large scales, systems undergo phase shifts when their behavior switches from one set of feedback loops to another. Finally, we need to understand systems with multiple conflicting goals. This is common in large-scale distributed systems with many participants. Different feedback loops become "antagonistic" and there must be a mechanism to avoid this or to resolve conflicts. We will investigate these issues in the next two years of the project.

2 Contractors contributing to the Deliverable

The following contractors have contributed to this deliverable:

- **UCL**. UCL is the main author of the three papers in the appendix. UCL explored the use of feedback loops in design, and in particular for structured overlay networks.

- **KTH**. KTH organized and gave the mini-course on reliable distributed programming. This course introduced the concurrent layered event-based architecture used for the definition of the self-managing algorithms in appendix C. KTH also developed the DKS structured overlay network [3] and an extension of DKS that handles network partitioning (see D1.1).

3 Results

We give a short overview of the main results of this deliverable with their motivations. For more details please see the three appendices to this report.

3.1 Overall structure of workpackage 2

This deliverable D2.1a is part of workpackage 2. The purpose of this workpackage is to construct the programming framework for self-managing systems. At month 12, this workpackage has three deliverables:

- D2.1a: Basic computation model. This deliverable gives the first answers to the question of how to program a self-managing system. During the first year, we explored programming with feedback loops and how they interact with a distributed system.
- D2.2a: Architectural framework specification. This deliverable specifies the component architecture: an event-driven component model.
- D2.3a: Formal operational semantics. This deliverable gives a formal foundation for the other two deliverables. In the first year, we have extended the Oz kernel language, which is a process calculus with many programmer concepts, with components and reflection. This work is based on the kell calculus. The result is called the Oz/K calculus.

These three deliverables study three aspects of the problem. The first studies how to program a self-managing system, the second studies the architecture of the system, and the third studies the framework for formally describing the system.

3.2 Programming with feedback loops

The Description of Work gives a vision of systems programmed as a set of interacting feedback loops. To make this vision concrete, we start by studying existing systems based on feedback loops (appendix A) and how to design systems with feedback loops (appendices A and B). The relaxed ring structure of appendix C (see also deliverable D1.1) handles ring maintenance in the case of imperfect failure detection (Internet-style failures). The paper reformulates the relaxed ring structure in terms of interacting feedback loops.

3.2.1 Definition of a feedback loop

A feedback loop consists of three elements that together interact with a subsystem: monitoring the relevant part of the subsystem's state, calculating a reaction, and implementing this reaction (see Fig. 1 in appendix A). We consider each of these three elements to be a concurrent component instance, interacting with other elements through asynchronous message passing. The three elements are usually designed explicitly. The subsystem that is observed is usually much larger and less well-known. It may contain parts of the external world and aggregated parts of the rest of the designed system.

3.2.2 Interacting feedback loops

A large system typically contains many feedback loops. These feedback loops are organized in a graph structure. The two main ways in which feedback loops interact are management and stigmergy:

- *Management.* This is when one feedback loop controls the other directly. The outer loop uses the inner loop as a service. For example, the inner loop can be a heating service using a thermostat. The outer loop can be a human that sets the temperature of the thermostat according to a particular policy.

- *Stigmergy.* This is when two feedback loops observe the same subsystem. Each loop observes and acts upon the subsystem and therefore indirectly affects the other.

3.2.3 General architectural framework

We consider the overall design of a system to be a set of interacting feedback loops. We organize such a system as a set of concurrent components that communicate by means of asynchronous events [4]. The default behavior is that the components are independent. System design experience in many areas suggests that this is the correct default (see Section 5 in appendix A).

In a self-managing system, the system must be able to monitor and reconfigure itslef. This implies that the system is built as a set of interacting components where the components are first-class entities that can be manipulated (passed, installed, removed) at run-time. This is called higher-order component programming.

The formal model for the general architectural framework sketched here is the Oz/K process calculus, developed by INRIA and presented in deliverable ??. This process calculus is quite rich. We expect to define and implement a subset of this calculus in the next stages of the project.

3.2.4 Some design rules

When building software systems, it is not possible to define a formal model for the whole system and “solve” this model to obtain the system’s properties. This method is too inefficient: it does not lead to a good design without a lot of search. A more direct method is to design the system to be self-managing from the start. To achieve this, the design must be constrained by a set of design rules that will guarantee the right self-managing properties. One of the objectives of the SELFMAN project is to find such a design methodology, by analogy with the methodologies of object-oriented design (e.g., such as explained in [7]). In this first exploratory phase of the project, we have determined several rough design rules. They are used in the papers of the appendix. We list the rules briefly:

- A first rule is to design systems based on convergence principles: each feedback loop should enforce a convergence principle. This will guarantee stability within the domain of application of the convergence principle. An example is a the feedback loop that uses negative feedback based on a monotonicly-changing system parameter.

- A second rule is to use the system instead of bypassing it. That is, consider a feedback loop as an encapsulated abstraction that provides a service, instead of as a set of parts to be interacted with directly. A feedback loop provides a level of robustness. It is a bad idea to bypass this. In addition, bypassing this can lead to instability. For example, two feedback loops based on negative feedback (and hence stable in isolation), when interacting through stigmergy can result in an effective feedback loop using positive feedback (and hence unstable).

- A third rule is to use management instead of stigmergy to interact with a feedback loop. That is, control the loop directly instead of modifying the system that the loop observes. This rule is related to the preceeding one. Using stigmergy can make a system unstable even if the individual loops are stable.

- A fourth rule is to use local instead of distributed feedback loops. Distributed feedback loops are inherently unreliable: they will collapse if there are failures. The correct way is to use local feedback loops and to model the distributed system as the subsystem being observed. The local feedback loops therefore interact through stigmergy. To avoid instability, there should be a global convergence property for all local feedback loops.

At this stage of the project, these rules remain mostly intuitive; we expect to put them on more solid theoretical grounding later in the project (see Section 4.1 in the Future Work).

3.2.5 Interaction of feedback and distribution

Large systems are by nature distributed. In the SELFMAN project we are specifically looking at such systems, building on the results attained for structured overlay networks. We therefore have to address the issue of how feedback and distribution interact. This question has been addressed to some degree in the development of robust distributed algorithms. For example, in the area of self-stabilizing algorithms, there is a convergence property: after a temporary perturbation, the system will converge to a state that is part of a set of stable states.

3.3 The relaxed ring

Structured overlay networks maintain their efficient routing properties in the face of network problems. The run-time maintenance of structured overlay networks consists of two independent parts:

- Maintain ring connectivity. This part guarantees that a connected ring exists. Efficiency is not a concern for this part.

- Maintain finger tables. This part adds “fingers” (routing table entries) to improve the efficiency.

The relaxed ring solves the ring connectivity problem in the case of realistic failure models that occur on the Internet. For a more detailed explanation of the relaxed ring, see appendix C or deliverable D1.1. The relaxed ring algorithm is formulated as a set of interacting feedback loops that maintain several invariants. In this section we summarize the main properties of the relaxed ring.

3.3.1 Lookup consistency

The relaxed ring guarantees the following lookup consistency property:

> *Lookup consistency* implies that at any time there is only one responsible node for a particular key k, or the responsible node is temporarily not available.

This property relaxes the lookup consistency property of [3] by adding the possibility that the responsible node is temporarily not available. This form of lookup consistency can be correctly implemented in the case of a failure detector that can have incorrect suspicions of failure (i.e., Internet-style failure detection). This is not the case for the original definition of lookup consistency.

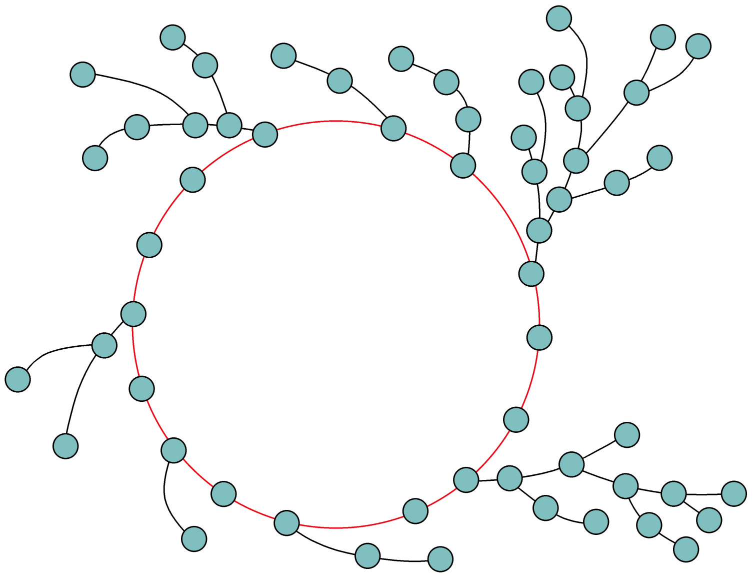

Figure 1: Example of a relaxed ring (perfect ring with bushy appendages)

3.3.2 Main invariant and join algorithm

The main invariant of the relaxed ring is that every peer node is in the same ring as its successor. This implies that it is sufficient for a node to have a connection with its successor to be considered inside the ring. Joining a relaxed ring is done in two steps:

- In the first step, a node sends a *join* message to its potential successor and receives a *join_ok* message as acknowledgement. At this point, the node is part of the relaxed ring and can be routed to. However, the node is part of one of the "bushes" sticking out of the inner, perfect ring (see example in Fig. 1).
- In the second step, the bushy extensions are collapsed node by node to form a perfect circular ring.

Both steps can be performed concurrently, and will in general be executing concurrently on a real system where nodes are continually joining and leaving. Because joining is separated into two independent steps, locking is not needed. This is an improvement over the join algorithm of [3], which requires cooperation between three nodes, two of which that need to be locked. The relaxed ring algorithm works in the case of an eventually perfect failure detector: a detector that may give false suspicions of failure (i.e., which correspond to temporary network problems on the Internet).

4 Future Work

In the first year of the project, we have mainly concentrated on looking at how systems are designed with feedback loops. In the next two years, we will continue this work in three complementary areas:

- Design rules for correct behavior of feedback loops.
- The large-scale structure of a system's behavior space.
- The layered security structure of a system.

4.1 Design rules for correct behavior of feedback loops

How can we be sure that a set of interacting feedback loops will work correctly? This is a nontrivial problem because the system is discrete and highly nonlinear. We have the following approach to solve this problem for practical system design. We are starting work to formalize feedback loops in terms of a process calculus (inspired by the Oz/K calculus of deliverable D2.3 and a version of an asynchronous probabilistic π calculus). We will then translate the formalization into recurrence equations that describe the behavior of the feedback loops. Certain forms of recurrence equations will guarantee convergent behavior. Reasoning backwards to the original system, this will give us design rules that if followed will guarantee convergent behavior. Ultimately, we will no longer need to formalize the system, but simply to follow the design rules. This work is being done by James Ortiz, a new Ph.D. student at UCL.

4.2 Large-scale behavior of a self-managing system

The large-scale structure of a system's behavior space consists of a set of domains, such that simple feedback behavior holds within each domain. When the feedback behavior is no longer possible, then the system changes from one domain to another. This happens, e.g., when the system is stressed strongly or when it has unstable behavior. The hotel lobby example of appendix A illustrates the case of unstable behavior. Another example is when a node fails in a structured overlay network: the feedback loops of the node collapse. Nevertheless, the rest of the system should continue to work.

4.3 Layered structure of a self-managing system

General self-managing systems are organized as sets of concurrent entities ("agents") that communicate through asynchronous message passing. An entity can be seen as an instance of a concurrent component or as an active object. Systems can be organized in layers, depending on how the agents are organized. We propose the following layered structure in decreasing order of adaptability and freedom:

- Open-ended market (a system where any agents can join). Arbitrary agent behavior is tolerated.

- Market (a system where each agent optimizes a local utility function). One designer per agent, the system is adaptable.

- Feedback loop architecture. One designer per system, the system is adaptable.

- Multiagent system with no particular structure. One designer per system, the system is not adaptable in general.

For more information on the general ideas underlying this structure, see [8]. The architecture described in appendix A covers the third element in this ranking, namely the feedback loop architecture. It holds when there is a single designer for the multiple agents.

4.3.1 Security infrastructure

Open-ended market systems are the top-most layer of this classification. They are secure because they tolerate arbitrary agent behavior. In order to make this work, the system provides a simple 'market infrastructure'. The basic idea is that the system enforces a conservation law using cryptographic protocols. For example, there can be a "currency" that is conserved. Services talk to each other and trade currency for results. The overall motor of the system is the external entities (like humans) connecting to it who want results. Basically, they ask for results in exchange for currency, and the propagation of currency inside the system drives its execution. Agents can be cooperative or malicious. The malicious agents will waste currency, but they will not endure because this will be detected (since they do not provide results!). This works on similar principles as Axelrod's Iterated Prisoner's Dilemma [1]. Good agents will flourish. It is similar to human markets but simplified. Secure systems built in this way are not "hacks" but are fundamentally correct and robust against malicious interference.

4.3.2 Collective intelligence

Collective Intelligence (COIN) is a design technique for multi-agent systems that allows to build systems that achieve a global goal with selfish agents that each are interesting only in maximizing a local utility function. The paper [10] explains the basics of COIN with a classic example, the El Farol bar problem. In this problem, agents are people who would like to take an evening off at a bar. If there are too few people at the bar, then the evening is a failure, and also if the bar is too crowded. Each agent remembers what happened the week before, and chooses one day to go in the next week. How can we maximize the global utility when each agent is thinking selfishly (and certainly not cooperating with the others!)? A naive algorithm that uses a utility based on how many people attended last time gives very bad results (a "Tragedy of the Commons").

It turns out that a good utility function is the “Wonderful Life” utility (so called through the Frank Capra movie). The value of the utility for agent w is the global utility minus the global utility where agent w is “disabled” (as if it did not exist). To make this computable locally, we use a simple reinforcement learning algorithm. Each agent has weights for each day of the week, and each week it changes one of the weights according to its experience in the bar that week. This is the “reward”. The utility is simply the sum of rewards over all the weeks. Picking a day for the next week is done according to a distribution that selects a day randomly according to the weights (technically, the algorithm implements a Boltzmann distribution and each weight is a Boltzmann energy).

The COIN techniques look like a good starting point for designing a security architecture for SELFMAN. The idea is that the SELFMAN infrastructure enforces local utility functions designed with COIN techniques. For example, the “reward” can be a “currency” designed to obey a conservation law and designed according to the Wonderful Life utility. This means that selfish agents (which will be the most numerous agents) will by design help the system achieve its global goals.

In SELFMAN we organized a mini-course on COIN which was held at partner ZIB on Feb. 15-16, 2007. It was taught by Mohamed El-Beltagy of Optomatica [2]. This course covered COIN and related work in the areas of game theory, agoric systems, and the theory of collectives.

4.3.3 The “grey goo” problem in Second Life

The “grey goo” problem in Second Life is related to this security layer. The MMORPG [1] application *Second Life* [6] allows people to trade objects and services in a virtual world. They have recently been having problems with "grey goo": self-replicating objects that use up resources (CPU, memory, and network). There was a discussion on the `e-lang` mailing list in Jan. 2007 about this problem and how to solve it. One way to solve it is to use a conservation law: pay for resources. Then the grey goo cannot replicate. This is the solution we could do in Selfman (the “agoric solution”). It seems that the Second Life developers implemented another solution: dynamic rate-limiting of object creation along with monitoring and alerting tools that let the sysops identify and kill off the goo that manages to exploit

[1]Massive Multiplayer Online Role-Playing Game

the system.

The Second Life experience is relevant for the SELFMAN project. Their experience is a useful datapoint. They are setting up a real economy, in some sense, and they have problems with pyramid schemes and other scams. For more information on the practical side of issues, see the Second Life Anti-Griefing Guild (`community.livejournal.com/slagg/`). "Griefers" in Second Life are like Byzantine nodes. Griefers are so-called because they create grief. They do their best to interrupt proceedings in virtual worlds often for no other reason than because it is possible.

5 Papers and publications

Three papers were published this year for this deliverable:

- *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS '06), Springer ENTCS 182, Sept. 2006.

  Abstract: Most software is fragile: even the slightest error, such as changing a single bit, can make it crash. As software complexity has increased, development techniques have kept pace to manage this fragility. But today there is a new challenge. Complexity is increasing rapidly as a result of two factors: the increasing use of distributed systems as a result of the sufficient reliability and bandwidth of the Internet, and the increasing scale of these systems as a result of the addition of many new computers to the Internet (e.g., mobile phones and other devices). To manage this new complexity, we propose an approach based on self- managing systems: systems that can maintain useful functionality despite changes in their environment. The paper motivates this approach and gives some ideas on how to build general self-managing software systems. An important part of the approach is to build systems as hierarchies of interacting feedback loops. We give examples of these systems and we deduce some of their design rules. The SELFMAN project is elaborating these ideas into a programming methodology and an implementation.

- *Implementing Self-Adaptability in Context-Aware Systems*, Workshop on Multiparadigm Programming with Object-Oriented Languages, part of ECOOP 2007, July 31, 2007. The appendix contains an extended abstract. The full paper will be part of the MPOOL workshop and will be available in time for the project review.

  Abstract: Context-awareness is the property that defines the ability of a computing system to dynamically adapt to its context of use [5]. Systems that feature this property should be able to monitor their context, to reason about the changes in this context and to perform a corresponding adaptation. Programming these three activities can become cumbersome as they are tangled and scattered all over in the system programs. We propose to model context-aware systems using feedback loops [9]. A feedback loop is an element of system theory that has been previously proposed for modelling self-managing systems. A context-aware system modelled as a feedback loop ensures that the activities of monitoring, reasoning and adapting to the context are modularised in independent components. In this work, we take advantage of such modularisation to explore different programming paradigms for each component of the loop.

- *A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks*, XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Nov. 2007. An early version of this paper was presented at the CoreGRID workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, FORTH, Heraklion, Greece, June 2007. This paper explains the relaxed ring work (see also deliverable D1.1) in the context of a feedback loop architecture. We intend to submit this paper to a major conference after adding simulation and experimentation results.

Abstract: There is no doubt about the increase in popularity of decentralised systems over the classical client-server architecture in distributed applications. These systems are developed mainly as peer-to-peer networks where it is possible to observe many strategies to organise the peers. The most popular one for structured networks is the ring topology. Despite many advantages offered by this topology, the maintenance of the ring is very costly, being difficult to guarantee lookup consistency and fault tolerance at all time. By increasing self-management in the system we are able to deal with these issues. We model ring maintenance as a self-organising and self-healing system using feedback loops. As a result, we introduce a novel relaxed-ring topology that is able to provide fault-tolerance with realistic assumptions concerning failure detection. Limitations related to failure handling are clearly identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.

These papers are included as appendices to this deliverable.

References

[1] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, 1984.

[2] Mohamed El-Beltagy. An introduction to COllective INtelligence (COIN), 2007. See `www.natural-computation.com/selfman`.

[3] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[4] Rachid Guerraoui and Louis Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[5] IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.

[6] Linden Research, Inc. Second Life, 2007. See `http://en.wikipedia.org/wiki/Second_Life`.

[7] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall PTR, 1997.

[8] Mark S. Miller and K. Eric Drexler. The agoric papers. In *The Ecology of Computation*, 1988.

[9] Peter Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[10] David Wolpert, Kevin R. Wheeler, and Kagan Tumer. General principles of learning-based multi-agent systems. In *Agents*, pages 77–83, 1999.

A Self Management and the Future of Software Design

Self Management and the Future of Software Design

Peter Van Roy[1] ,[2]

Department of Computing Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Abstract

Most software is fragile: even the slightest error, such as changing a single bit, can make it crash. As software complexity has increased, development techniques have kept pace to manage this fragility. But today there is a new challenge. Complexity is increasing rapidly as a result of two factors: the increasing use of distributed systems as a result of the sufficient reliability and bandwidth of the Internet, and the increasing scale of these systems as a result of the addition of many new computers to the Internet (e.g., mobile phones and other devices). To manage this new complexity, we propose an approach based on *self-managing systems*: systems that can maintain useful functionality despite changes in their environment. The paper motivates this approach and gives some ideas on how to build general self-managing software systems. An important part of the approach is to build systems as hierarchies of interacting feedback loops. We give examples of these systems and we deduce some of their design rules. The SELFMAN project is elaborating these ideas into a programming methodology and an implementation.

Keywords: Software development, self management, general system theory, distributed system, feedback, software component, complexity, concurrency, asynchronous, autonomic computing, overlay network

1 Introduction

Software is fragile and highly nonlinear: even a minor error can have catastrophic effects. Major disasters have occurred due to minor errors such as omitted commas in Fortran programs or changed bits because of alpha rays [11]. So far, this has not unduly hampered the quantity of software being developed. As software complexity has increased, software development techniques have kept pace. This situation is analogous to the Red Queen's behavior in Alice [10]: we are running as fast as we can in order to stay in the same place. Software development is now facing a new challenge: complexity is increasing quickly because of two reasons. First, the reliability and bandwidth of the Internet infrastructure has reached a point where it is

[1] This paper is intended to stimulate discussion; all comments are welcome! This work is funded by the European Union in the SELFMAN project (contract 34084), EVERGROW project (contract 001935), and CoreGRID network of excellence (contract 004265). We thank Luis Quesada, Boriss Mejias, Raphaël Collet, Yves Jaradin, Kevin Glynn, and Seif Haridi for comments that helped improve this paper.

[2] Email: pvr@info.ucl.ac.be

feasible to build large distributed applications. Examples of such applications include a wide variety of file-sharing programs (Napster, Gnutella, Morpheus, Freenet, Bit Torrent, etc.), collaborative tools (Skype and other messenger tools), Massive Multiplayer Online Role Playing Games (MMORPGs) (World of Warcraft, Dungeons & Dragons, etc.) and research testbeds (SETI@home [25], PlanetLab [12], etc.). Technologies for building such applications now exist, e.g., Web services and Grid software. The second reason is the increase in the number of small devices connected to the Internet. For example, mobile phones are now full-fledged computing nodes with Internet connectivity, and protocols such as Zigbee, Bluetooth, and Wifi facilitate network connectivity among small devices.

How can we address the problem of programming large-scale distributed systems? Such systems have new properties that greatly increase the complexity of programming: scale (large numbers of independent nodes), partial failure (part of the system fails), security (multiple security domains), resource management (resources are localized), performance (harnessing multiple nodes or spreading load), and global behavior (emergent behavior of the system as a whole). Each of these properties has been studied in isolation. For example, the area of distributed algorithms has solutions for handling partial failure in many cases. But the properties have not been looked at together. The purpose of this paper is to give some ideas how this can be done.

Global behavior is particularly relevant for large systems. They must be designed carefully, otherwise the system will not behave well when stressed. Ideally, it should converge rapidly to its desired behavior and stay there despite changes in the system's environment. But it may instead collapse, oscillate, or show chaotic behavior. Such erratic behavior has been observed for power grids and has resulted in large-scale power outages [15]. One reason for this is because the power grid's behavior was designed for a situation close to equilibrium; it was not studied far from equilibrium.

2 Self-managing systems

To build large-scale distributed systems with good behavior, we need a framework in which to think about them. What should such a framework look like? To reduce the complexity of the system, it should be able to manage its own problems as much as possible. This leads us to propose self-managing systems as a suitable framework. A self-managing system is one that can maintain its functionality despite changes in its environment, in a general sense.

Self-managing systems have recently been brought to the forefront because of IBM's Autonomic Computing initiative [19]. When computer systems become large then the cost of managing them becomes prohibitive. The initiative aims to reduce this cost by removing humans from the management loop. The role of humans is then to manage the policy and not to maintain the mechanisms. This greatly reduces the need for manual intervention.

Another area that is building self-managing systems is structured overlay networks [1]. This research is inspired by the popular protocols of peer-to-peer networks. Many of the applications mentioned in the introduction are based on these peer-to-peer networks. Unlike peer-to-peer networks based on random neighbor communication, structured overlay networks provide both guarantees (information is guaranteed to be found if it exists) and efficiency (broadcast does not flood the network as it does in, e.g., random neighbor networks such as the one used in Gnutella). Structured overlay networks provide primitive self-managing behavior: they reorganize themselves to maintain their functionality in reaction to environmental changes such as failures and overloads. Structured overlay networks have led to robust software that is being used in various areas, such as the construction of robust distributed communication networks and robust storage services that continue to provide service despite high node turnover (node "churn").

These two research areas, autonomic systems and structured overlay networks, have attracted attention once again to self-managing systems. But self-managing systems are actually a very old idea. The beginning of the area as a discipline can be dated to the definition by Norbert Wiener of cybernetics in the 1940's [29] and by Ludwig von Bertalanffy of general system theory in the 1960's [5]. The basic idea of system theory is to study the concept of a *system*, its properties and design. There are various ways to define the concept of a system [24]. For this paper, we define a system recursively as a set of components (called subsystems) connected together to form a coherent whole. The main problem is to understand the relationship between the system and its subsystems: can we predict a system's behavior and can we design a system with a desired behavior.

System theory is still very much in its early stages. Recent research results have not been systematized in a textbook and the ideas have not been applied to computer science in a systematic way. W. Ross Ashby wrote an introductory textbook in 1956 that is still worth reading today [4]. Gerald M. Weinberg wrote an introduction in 1975 explaining how to use system theory to improve general thinking processes [28]. In the area of computer systems, textbooks exist only for specialized subfields such as distributed algorithms [21]. We consider that it is high time to apply system theory to software construction. This paper gives examples of realistic systems to motivate this goal and to explore how to build software according to system theory.

3 Designing self-managing systems

How does one design a self-managing software system? We do not yet have a general set of design techniques, but we can talk about several important aspects: feedback loops, global properties, and a general architectural framework. It turns out that designing with feedback loops is fundamental. Feedback loops are currently being used for the autonomous management of computing clusters, for example they are being used in J2EE clusters [6] and Grid systems [2]. But feedback loops are much more generally applicable in system design. We give examples of systems built with feedback loops to see what they can teach us for the general case. The paper by Andrzejak *et al* [2] gives a broad introduction to the different disciplines that can be useful when designing adaptive systems with feedback loops. The present paper is narrower: it restricts itself to the architectural questions of how the loops are organized and how they interact with each other and with distributed programming.

3.1 Feedback loops

The notion of a *feedback loop* is a basic element of system theory. A feedback loop consists of three elements that together interact with a subsystem (see Figure 1): an element that monitors the state of the subsystem, an element that calculates a corrective action, and an element that applies the corrective action to the subsystem. For the purposes of this paper, we consider these elements to be concurrent software agents that communicate by asynchronous message passing. The complete system can be described as a graph of interacting feedback loops. Feedback loops can interact in two main ways. The simplest interaction is where both loops affect interdependent system parameters, i.e., they interact through their environment. This is called stigmergy. A second form of interaction is where a loop manages another loop, i.e., the first loop continuously adapts the policy implemented by the second loop. In both cases, the system's global behavior depends on all the feedback loops taken together.

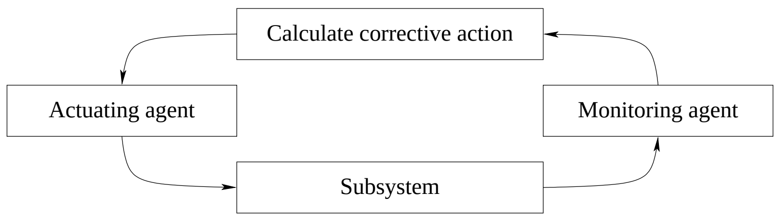

Fig. 1. Basic structure of a feedback loop

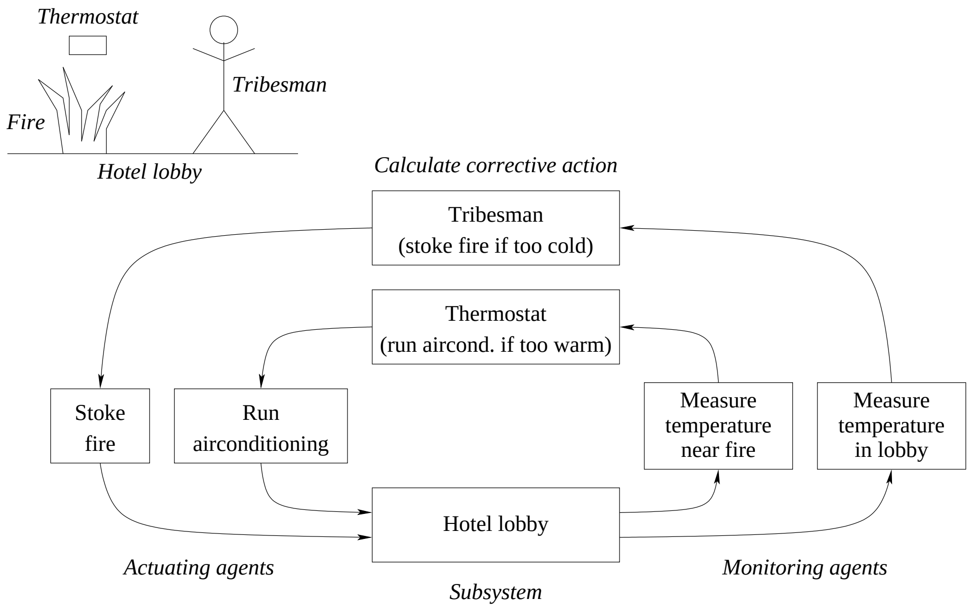

Fig. 2. Wiener's example of two feedback loops interacting through stigmergy

3.1.1 Two simple examples

The first example is taken from Wiener [29] and is shown in Figure 2. It consists of two interacting feedback loops with counterintuitive global behavior: in an air-conditioned hotel, a primitive tribesman attempts to warm himself by starting a fire. This causes the airconditioning to work harder, so the result is that the harder he stokes the fire, the lower the temperature becomes. In this example, the two loops affect system parameters that depend on each other, namely the temperature in different parts of the lobby. Each block in the figure is a concurrent agent continuously sending asynchronous messages to the other agents in the direction of the arrows. Even though each loop taken in isolation uses negative feedback and is stable,[3] the result of both loops is that the system becomes unstable, i.e., the temperature will continue to decrease (until the system reaches a boundary, and then its behavior will change again). We conclude that it is not enough to add a negative feedback loop to an existing system to ensure stability! The result may well be unstable because of the new loop's interaction with the system.

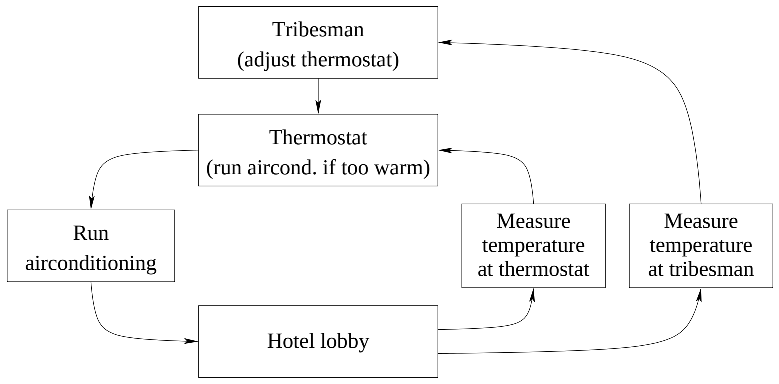

Fig. 3. Wiener's example modified to use management instead of stigmergy

The correct solution is given in Figure 3. Instead of starting a fire, the tribesman simply adjusts the thermostat. This maintains the stability of the airconditioning loop. This is an example of one loop managing another. This illustrates a design rule: to modify a system's behavior, the right way is to work with the system and not to try to bypass it.

The second example is shown in Figure 4. This shows a generic single-user application as a feedback loop structure. We give this example to illustrate that feedback loops are generally useful in programming and not just for contrived examples such as Figure 2. Feedback loops are omnipresent in software systems if one looks with the right mindset. The three elements of the loop in Figure 4 all run on a single computer, and the subsystem being managed is a human user. The monitoring and actuating agents are the computer's GUI interface. Remark that we consider the

[3] In negative feedback, an increase in the monitored value of a system parameter causes a corrective action that decreases the system parameter. In positive feedback, the corrective action increases the system parameter.

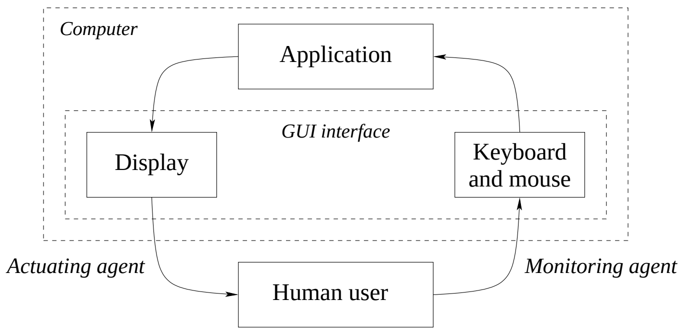

Fig. 4. A single-user application shown as a feedback loop

user and not the application to be the managed subsystem. This viewpoint is advantageous because it lets us extend the feedback loop structure in interesting ways. We can put a second loop around the first to monitor the application's behavior and apply corrections if something goes wrong. When the user runs two applications and passes information between them then we have two loops interacting through stigmergy. The rest of this paper gives more substantial examples of systems shown as feedback loop structures, including systems that were not originally conceived in this way.

3.1.2 Using program properties

Designing systems with feedback has been extensively studied in electronics, typically with building blocks such as operational amplifiers and phase-locked loops. These systems exploit the fact that there is a good (piecewise) linear approximation of the building blocks' behavior. This is a strong condition that can be exploited. But linearity is probably too strong a condition to impose on computer systems, which are highly nonlinear by default, e.g., changing a single bit can have major effects. It may be possible to use a weaker property than linearity that can be satisfied by computer systems and that gives a satisfactory design theory. The approach then is to choose first a property that facilitates reasoning about the program and its global behavior, and then to build a program that satisfies the property. This can greatly simplify program design. Note that one possible failure mode is that the property itself no longer holds.

One example property is monotonicity or strict monotonicity. In a strict monotonic system, when the input changes in one direction (e.g., increases, in a general sense), the output will also change in the same direction. Using monotonicity as the basic property is sufficient for designing systems with feedback. A negative feedback amplifier can be built using strict monotonicity. Another property weaker than linearity that may be useful is continuity, but continuity is in general not enough to guarantee stability. We note that two further properties that may be useful in a theory of feedback program design are determinism and confluence.

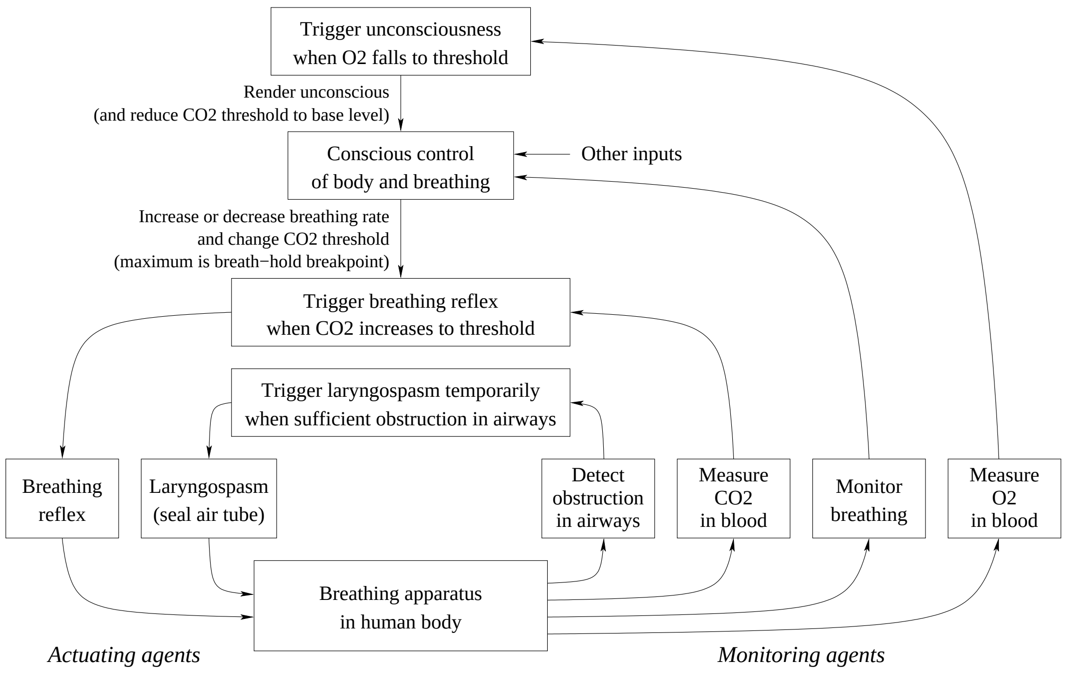
Trigger unconsciousness when O2 falls to threshold
Render unconscious (and reduce CO2 threshold to base level)
Conscious control of body and breathing
Other inputs
Increase or decrease breathing rate and change CO2 threshold (maximum is breath−hold breakpoint)
Trigger breathing reflex when CO2 increases to threshold
Trigger laryngospasm temporarily when sufficient obstruction in airways
Breathing reflex
Laryngospasm (seal air tube)
Detect obstruction in airways
Measure CO2 in blood
Monitor breathing
Measure O2 in blood
Breathing apparatus in human body
Actuating agents
Monitoring agents

Fig. 5. Feedback loop structure of the human respiratory system

3.2 System design with feedback loops: the human respiratory system

Let us give a detailed example of a practical design that uses feedback loops. Our example is taken from a biological system, namely the human body. Biological systems have to survive in natural environments, which can be particularly harsh. For that reason, we consider that studying biological systems is a useful way to get insight in how to design software for a more complex system. Our example is the human respiratory system. Figure 5 shows the different components of this system and how they interact. We derived this figure from a precise medical description of the system's behavior [31]. The figure is slightly simplified when compared to reality. We have left out interactions with the rest of the body. Nevertheless it is complete enough to give many insights. There are four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious). From the figure we can deduce what happens in many realistic cases. For example, when choking on a liquid or a piece of food, the larynx constricts and we temporarily cannot breath (this is called laryngospasm). We can hold our breath consciously: this increases the CO_2 threshold so that the breathing reflex is delayed. If you hold your breath as long as possible, then eventually the breath-hold threshold is reached and the breathing reflex happens anyway. A trained person can hold his or her breath long enough so that the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the normal 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 nonlinear and may run both with negative or positive feedback. It is the most complex of the four loops by far. We can justify why it is sandwiched in between two simpler loops. On the one 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 nested feedback loops that implement abstraction. On the other side, the outermost loop overrides the conscious control 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 (but not all) of the body's feedback loop structures. This very power means that it needs a check.

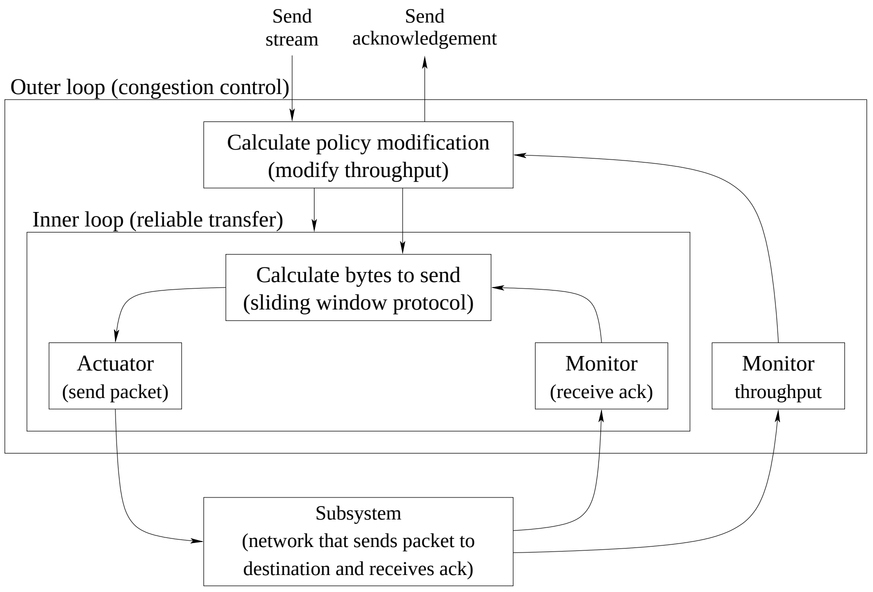
Send stream
Send acknowledgement
Outer loop (congestion control)
Calculate policy modification (modify throughput)
Inner loop (reliable transfer)
Calculate bytes to send (sliding window protocol)
Actuator (send packet)
Monitor (receive ack)
Monitor throughput
Subsystem (network that sends packet to destination and receives ack)

Fig. 6. An example programming pattern with two nested feedback loops

3.3 A new way of designing programs

The style of system design illustrated in the last section can be applied to programming. Programming then consists of building hierarchies of interacting feedback loops. Let us give a simplified example with two nested feedback loops that implements a reliable byte stream transfer protocol with congestion control (this is a variant of the TCP protocol). The protocol sends a byte stream from a source to a destination node. Figure 6 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 which packets have arrived successfully. The inner loop manages a sliding window: the actuator sends packets so that the sliding window can advance. The sliding window can be seen as a case of negative feedback using monotonic control. The outer loop does congestion control: it monitors the throughput of the system and acts by either changing the policy of the inner loop or by changing the inner loop itself. If the buffered send stream grows too big or the rate of acknowledgements decreases, then it modifies how the inner loop works, for example by reducing the rate of send acknowledgements or the rate of sending. If the transfer stops then the outer loop may terminate the inner loop and abort the transfer.

This structure is a special case of a multi-agent system. Each block in Figure 6 is a single agent acting concurrently with the others and sending messages asynchronously to the others. Each of the two feedback loops implements one task according to a given policy. The policy of the inner loop is determined by the outer loop. Because the system is distributed over two nodes, part of the design consists in situating each agent on a node.

The example of Figure 6 has just two nested feedback loops. In a real system, there will typically be more nested feedback loops. In particular, the outermost loop determines the main interface between the system and its environment.

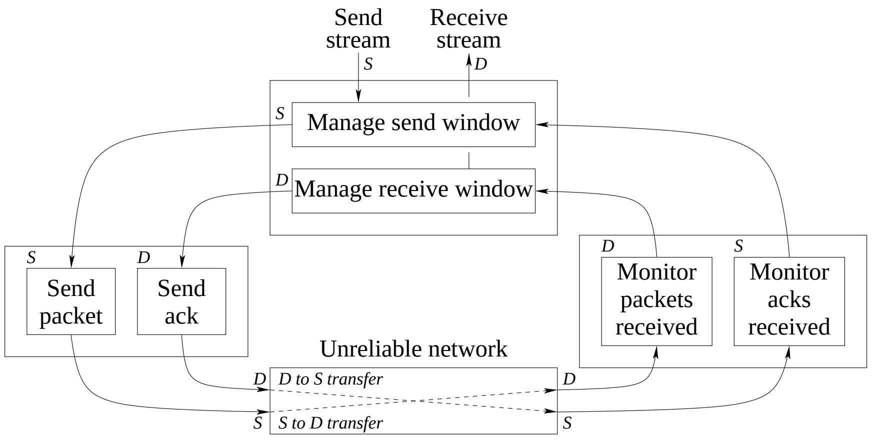

Fig. 7. Inner loop of the reliable byte stream protocol showing distribution

3.4 Interaction between feedback loops and distribution

The protocol of Figure 6 runs on a distributed system consisting of two nodes. Figure 6 only shows what happens at the source node. Figure 7 gives a more complete depiction of the inner loop of Figure 6 that shows the execution on both nodes. In Figure 7, each component is annotated with S or D depending on whether it executes on the source or destination node. This protocol can be seen as two feedback loops (the S loop and the D loop), each executing on one node (S or D), interacting through stigmergy over the unreliable network. If one node fails, then its loop disappears and the other loop sees a change in the behavior of the network. Another way to see the protocol is as a single distributed feedback loop, with parts executing on both source and destination nodes.

An interesting open question raised by this example is how to design distributed feedback loops. This is nontrivial because of the interactions between the design of the loop, its distribution, and the partial failures that it is intended to tolerate. Designing these systems is still mostly an open research question. Structured overlay networks are an interesting special case that is presented below. Other special cases include parts of distributed algorithm theory such as self-stabilizing systems [32]. These systems are able to survive large classes of transient faults.

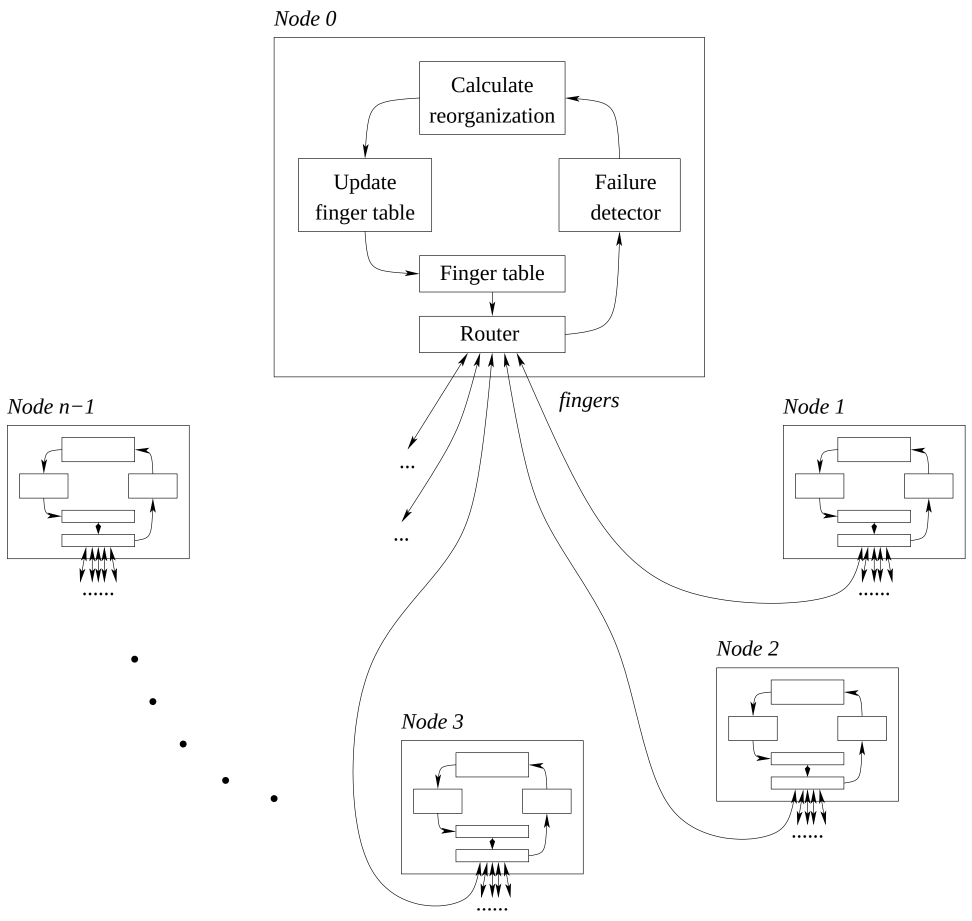

Fig. 8. Feedback loop structure of a structured overlay network

3.5 Feedback loops in a structured overlay network

We complete our series of examples by outlining how a structured overlay network can be formulated in terms of feedback loops. The most primitive functionality of a structured overlay network is to self-organize a large number of computing nodes to provide reliable and efficient routing despite nodes continuously joining and leaving the network [1,17]. A node can leave in two ways, either by a deliberate action or by failure of the node or its network connections. At all times, routing between non-failed nodes must be correct and efficient.

Figure 8 shows the feedback loop structure of a structured overlay network with n computing nodes numbered from 0 to $n-1$. Node 0 is drawn in detail; the other nodes are shown schematically. The routing organization of the structured overlay network consists of two levels. The first level is a ring in which each node has direct communication links (called *fingers*) to a fixed number f of successors. This ensures correctness (each node can reach all the others by walking the ring) and fault tolerance (failure of $f-1$ nodes does not affect reachability). The second level adds additional links to improve efficiency. The routing algorithm uses a convergence criterium to ensure that eventually the destination node is reached. Each routing hop reduces the distance to the destination until the distance reaches zero. Many well-known structured overlay networks, such as Chord and DKS, are organized in this way.

The communication links provide failure detection. When a node detects the failure of a link then it reorganizes its local finger table to provide correct routing. There is also a distributed algorithm to improve routing efficiency. Correct operation of the structured overlay network is therefore based on three convergence properties:

- Within each node, the finger table converges to a correct content.
- Globally, the finger tables converge together to improve routing efficiency.
- When routing, a message in transit converges to its destination node.

From the viewpoint of each node, the subsystem being managed consists of the set of nodes it is linked to. When a node leaves or fails, it is eventually dropped from each set containing it. When a new node joins, it is given an initial set that depends on its position in the ring. Since these operations are common, this means that the feedback structure is undergoing frequent changes. Ghodsi [17] gives algorithms and an implementation of a structured overlay network, DKS, that has the above structure. He proves that it does correct routing assuming that the failure detectors are strongly complete, i.e., every node crash will eventually be detected permanently [18]. The structure modifications done by DKS are designed to be atomic and preserve the topology of the overlay network.

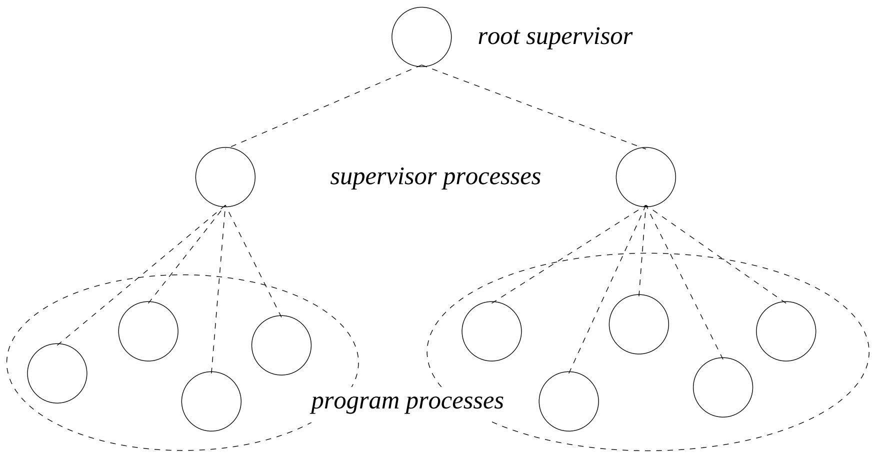

Fig. 9. Supervisor tree architecture of an Erlang program

4 Related work

Several areas of computer science already use a feedback loop architecture. This section gives two examples, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior, and discusses them as instances of a feedback loop architecture.

4.1 The Erlang system

The Erlang system is designed to build distributed systems that survive software and hardware faults [3]. It has been successfully used to build systems of extremely high dependability, for example the AXD301 ATM switch which has a claimed down time of only 30 milliseconds per year [30]. Erlang is designed according to the hypothesis that software faults cannot be eliminated completely. Instead of trying to eliminate them, Erlang allows programs to survive them. An Erlang program is organized as a set of concurrent agents (called *processes* in Erlang terminology) that communicate by asynchronous message passing.

When a problem occurs in a process, the Erlang philosophy is to let the process fail and to let another process handle recovery. Erlang uses a concept called *supervisor tree* to manage this. The program agents form the leaves of the supervisor tree (see Figure 9). Each internal node in the supervisor tree corresponds to a feedback loop in our architecture. The first internal level in the tree consists of supervisor agents that observe pools of agents in the program's execution. If a program agent fails, then a supervisor agent will restart it in a consistent state, using a database to get the consistent state. There are two kinds of supervisors, AND supervisors that restart all processes in a pool if one fails and OR supervisors that restart just the failed processes. The second internal level in the supervisor tree consists of a root agent that handles failures of the supervisor agents. This root agent must be completely reliable. This is possible because it is a very small program.

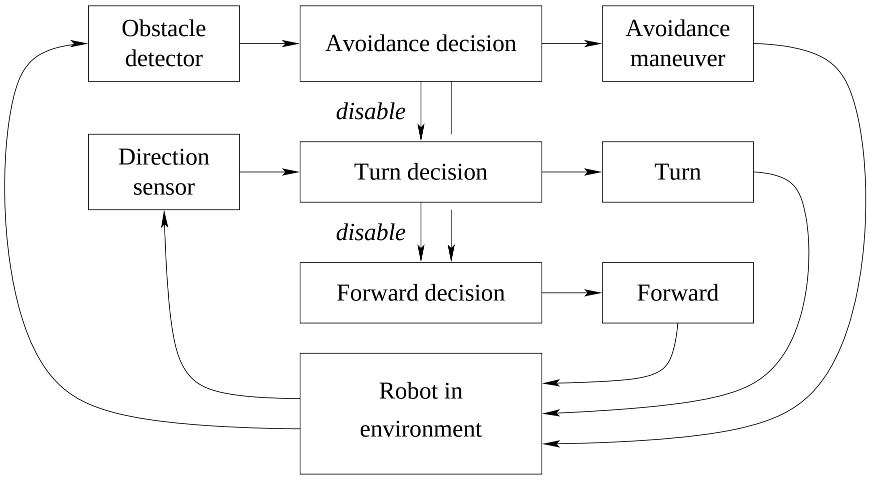

Fig. 10. Feedback loop structure of an obstacle-avoiding robot in the subsumption architecture

4.2 The subsumption architecture

The subsumption architecture of Rodney Brooks is a way to implement intelligent systems by decomposing complex behaviors into layers of simple behaviors that interact through their environment [7,8]. Knowledge is not represented directly inside the system, but indirectly through the system's state in its environment. The subsumption architecture has been used to successfully implement systems that interact with their environment in a life-like fashion. For example, an obstacle-avoiding robot can be designed with three layers: a move forward layer, a turn layer, and an obstacle-avoiding layer. Each layer is a feedback loop that observes the world continuously. The layers are given priorities. If a layer can react, then it disables the lower layers and performs its own actions. In the terminology of Brooks, it suppresses inputs to the lower layers and inhibits outputs from the lower layers. The default behavior is to move forward. If the direction is wrong, then the turn layer disables the move forward layer to turn. If there is an obstacle, then the obstacle-avoiding layer disables the other two layers and performs an obstacle avoidance maneuver. Figure 10 shows this obstacle-avoiding robot as a feedback architecture. This is a simple example that shows the basic principle. There exist more refined versions of the architecture.

In the subsumption architecture, the feedback loops interact through stigmergy. E.g., in a robot, all the loops detect the robot's position and control the robot's movements. In the feedback loop architecture, feedback loops can also have a policy/mechanism relationship, where each loop modifies the policy that is implemented by the next innermost loop.

5 General architectural framework

Let us now take a step back from the above examples and summarize what a general architectural framework can look like for building a self-managing system. The system is organized as a set of concurrent components that communicate by means of asynchronous events. The default behavior is that the components are independent. Any synchronous or dependent behavior must be programmed explicitly. This default gives good results in many cases: for fault-tolerant systems such as Erlang [3], for network-transparent distributed programming systems such as Mozart [13], and for secure distributed programming systems such as E [22]. It also matches well with the complex systems approach taken in physics [14] and used, e.g., in approaches such as belief propagation for solving inference problems [33].

Following the examples of Sections 3.2–3.4 and Section 4, the system consists of a hierarchy of interacting feedback loops, where each feedback loop is implemented by several agents and each agent is an instance of a component. Feedback loops interact either through stigmergy or through management.

5.1 Higher-order component model

In a self-managing system, the system is able to monitor and reconfigure itself, that is, install and update parts of itself while it is running. If the system is built as a set of interacting components then it is possible for components to install other components. Components are therefore first-class entities that can be passed as arguments to other components. This is called *higher-order* component programming. The Fractal component model is an example of such a component model [9]. This model is already being used as a framework for building self-managing systems [6]. In a higher-order component model, it takes some care to determine what component is to blame when a subsystem fails. This has been studied by Findler and Blume [16].

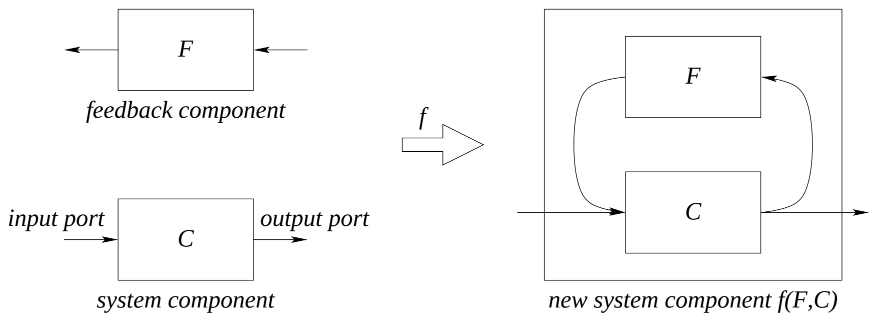

Fig. 11. A component combinator for programming with feedback loops

5.2 Programming with feedback loops

With the right abstractions, a programming language can make programming with feedback loops simple. Each component is a concurrent entity with one input port that accepts a stream of input events and one output port that returns a stream of output events. Components ignore irrelevant events. Both control and content events pass through the same ports. These properties make it easy to compose components in a modular way. This programming model is similar to the model used by Guerraoui and Rodrigues for defining distributed algorithms in a compositional way [18].

Figure 11 shows a component combinator f that takes two components F and C and returns a component $f(F, C)$ that combines F and C in a feedback arrangement. The combinator f satisfies properties such as $f(F_1, f(F_2, C)) = f(F_2, f(F_1, C))$. We can define an operator $||$ such that $f(F_1, f(F_2, C)) = f(F_1||F_2, C)$. This operator is a form of parallel composition that connects the input and output streams of F_1 and F_2. There are variations of f depending on whether C is explicit (part of the program) or implicit (part of an environment) and depending on whether the feedback loop is managed or not. The semantics of the combinator f needs to take into account two effects:

- The interleaving of the input and output streams. That is, C's input is the merge of $f(F, C)$'s input and F's output and $f(F, C)$'s output is also the input to F.
- Both C and F have a propagation delay, i.e., an output event does not appear instantaneously when an input event is given.

5.3 Global properties

An important part of any general system theory concerns the global properties of a system. Can they be determined for an existing system and can we design systems with desired global properties? The latter question is especially important for large-scale computer systems, such as the Internet or distributed systems built on top of the Internet. Some of the important points are the system's stability, its behavior when stressed, and whether the system's imminent collapse can be detected before it happens. Answers to some of these questions exist for complex systems in physics. Such systems consist of large numbers of very simple components, but they can sometimes be a useful approximation to computer systems. For example, Krishnamurthy *et al* [20] have done an analytic study of the Chord structured overlay network using a master equation approach. Another example is the belief propagation algorithm. This algorithm is defined in terms of message passing between large numbers of simple nodes [33]. It has been used to give solutions to the SAT problem and other problems. Belief propagation is a general technique that can determine global properties of a system in terms of local properties. It can be used for monitoring global properties as part of a feedback loop.

6 Conclusions

This paper motivates that a good approach for building large-scale distributed systems is to consider them as general self-managing systems. We propose to build self-managing software systems as sets of concurrent agents interacting by means of asynchronous events and implemented using a component model with first-class components and component instances. In this framework, self-managing systems are built as hierarchies of interacting feedback loops. The first design rule is that the whole system (except perhaps a small kernel) should be inside a feedback loop. Feedback loops interact through two mechanisms, stigmergy (shared environment parameters) or management (one loop controls another). The feedback loop structure is designed to provide a desired global behavior. This behavior should also be predictable from the loop structure. We relate this proposal to two other architectures, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior.

These ideas are being realized in SELFMAN, a project in the European 6th Framework Programme that started in June 2006 [27]. We intend to elaborate these ideas into a programming methodology together with an implementation. It should be as easy to program with and reason about a feedback loop as it is for an object or a component. We will design and formalize a component model that is based on the Oz kernel language extended with elements from the Fractal model. We will use this component model as the basis of a programming model along the lines of Section 5 and implement this model in Mozart [26,9,13,23]. We will build a feedback loop architecture on top of this implementation and use it to implement a self-managing replicated transactional storage service.

References

[1] Aberer, K., L. Onana Alima, A. Ghodsi, S. Girdzijauskas, M. Hauswirth, and S. Haridi, *The essence of P2P: A reference architecture for overlay networks*, 5th International Conference on Peer-to-Peer Computing (P2P 05), IEEE Computer Society, 2005.

[2] Andrzejak, Artur, Alexander Reinefeld, Florian Schintke, and Thorsten Schütt, *On Adaptability in Grid Systems*, Future Generation Grids, Springer LNCS, 2005.

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

[4] Ashby, W. Ross, "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): `http://pcp.vub.ac.be/books/IntroCyb.pdf`.

[5] von Bertalanffy, Ludwig, "General System Theory: Foundations, Development, Applications," George Braziller, 1969.

[6] Bouchenak, S., F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quéma, and J.-B. Stefani, *Architecture-Based Autonomous Repair Management: Application to J2EE Clusters*, 2nd International Conference on Autonomic Computing (ICAC'05), 2005, pp. 369–370.

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

[8] Brooks, Rodney A., *Intelligence without representation*, Artificial Intelligence 47, 1991, pp. 139–159.

[9] Bruneton E., V. Quéma, T. Coupaye, M. Leclercq, and J.-B. Stefani, *An Open Component Model and its Support in Java*, Proceedings 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Springer LNCS 3054, 2004.

[10] Carroll, Lewis, "Through the Looking-Glass and What Alice Found There," 1872 (Dover Publications reprint 1999).

[11] Ceruzzi, Paul E., "Beyond the Limits: Flight Enters the Computer Age," MIT Press, Cambridge, MA, 1989.

[12] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, *PlanetLab: An Overlay Testbed for Broad-Coverage Services*, ACM SIGCOMM Comp. Comm. Review, 33(3), 2003.

[13] Collet, Raphaël, and Peter Van Roy, *Failure Handling in a Network-Transparent Distributed Programming Language*, in Recent Advances in Exception Handling Techniques, C. Dony *et al* (Eds.), Springer LNCS 4119, 2006.

[14] *EVERGROW: Ever-growing global scale-free networks, their provisioning, repair and unique functions*, Integrated Project, European 6th Framework Programme, 2004-7. Internet: `http://www.evergrow.org`.

[15] Fairley, Peter, *The Unruly Power Grid*, IEEE Spectrum Online, Oct. 2005.

[16] Findler, Robert Bruce, and Matthias Blume, *Contracts as Pairs of Projections*, FLOPS 2006, April 24-26, 2006.

[17] Ghodsi, Ali, "Algorithms for Large Scale Self Managing Overlay Networks," Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, 2006.

[18] Guerraoui, Rachid, and Luis Rodrigues, "Introduction to Reliable Distributed Programming," Springer-Verlag Berlin, 2006.

[19] IBM, *Autonomic computing: IBM's perspective on the state of information technology*, 2001. Internet: `http://researchweb.watson.ibm.com/autonomic/`.

[20] Krishnamurthy, S., S. El-Ansary, E. Aurell, and S. Haridi, *A statistical theory of Chord under churn*, The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), 2005.

[21] Lynch, Nancy, "Distributed Algorithms," Morgan Kaufmann, San Francisco, CA, 1996.

[22] Miller, Mark, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, May 2006.

[23] Mozart Programming System, version 1.3.2, June 2006. Internet: `http://www.mozart-oz.org`.

[24] Principia Cybernetica Web. Entry "system," August 2006. Internet: `http://pespmc1.vub.ac.be/ASC/SYSTEM.html`.

[25] SETI@home, August 2006. Internet: `http://setiathome.berkeley.edu/`.

[26] Van Roy, Peter, and Seif Haridi, "Concepts, Techniques, and Models of Computer Programming," MIT Press, Cambridge, MA, 2004.

[27] Van Roy, Peter, Ali Ghodsi, Seif Haridi, Jean-Bernard Stefani, Thierry Coupaye, Alexander Reinefeld, Ehrhard Winter, and Roland Yap, *Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components*, CoreGRID Technical Report TR-0018, Dec. 14, 2005. Internet: `http://www.ist-selfman.org`.

[28] Weinberg, Gerald M., "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).

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

[30] Wiger, Ulf, *Four-fold increqse in productivity and quality* industrial-strength functional programming in telecom-class products, Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems, 2001.

[31] Wikipedia, the free encyclopedia. Entry "drowning," August 2006. Internet: `http://en.wikipedia.org/wiki/Drowning`.

[32] Wikipedia, the free encyclopedia. Entry "self-stabilization," August 2006. Internet: `http://en.wikipedia.org/wiki/Self-stabilization`.

[33] Yedidia, J.S., W.T. Freeman, and Y. Weiss, *Understanding Belief Propagation and Its Generalizations*, Exploring Artificial Intelligence in the New Millennium, Chap. 8, Jan. 2003. Also MERL Technical Report TR-2001-22, Jan. 2002.

B Implementing Self-Adaptability in Context-Aware Systems

Implementing Self-Adaptability in Context-Aware Systems [⋆]

Boris Mejías[1] and Jorge Vallejos[2]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
boris.mejias@uclouvain.be
[2] Vrije Universiteit Brussel, Brussels, Belgium
jvallejo@vub.ac.be

1 Introduction

Context-awareness is the property that defines the ability of a computing system to dynamically adapt to its context of use [1]. Systems that feature this property should be able to monitor their context, to reason about the changes in this context and to perform a corresponding adaptation. Programming these three activities can become cumbersome as they are tangled and scattered all over in the system programs.

We propose to model context-aware systems using feedback loops [2]. A feedback loop is an element of system theory that has been previously proposed for modelling self-managing systems. A context-aware system modelled as a feedback loop ensures that the activities of monitoring, reasoning and adapting to the context are modularised in independent components. In this work, we take advantage of such modularisation to explore different programming paradigms for each component of the loop.

2 Feedback Loops for Self-Adaptable Context-Aware Systems

Modelling software systems using feedback loops implies for the developers to identify which kind of information needs to be monitored, dedicating particular agents for this task. Once the monitored information is collected, another component is in charge of deciding correcting actions, using an actuator agent to apply the corrections to the system.

Consider the case of a computer-assisted system for managing the lights of a so called *intelligent house*. This system consists of a set of lights and sensors that detect the presence of people in the house. The detection of a person is monitored by a specialised component that decides whether to turn on or off the lights, or simply modify their intensity. The loop is depicted at the left side of figure 1.

Since the use of mobile devices such as phones, PDAs, media players or GPSs are becoming very common, we can expect that users will use her/his mobile device to communicate with the house. We also expect that these devices can adapt their behaviour according to their context. The context can represent locality, CPU use, battery load,

[⋆] This work has been partially funded by the European projects EVERGROW and SELFMAN, and by the flemish project of Context-Driven Adaptation of Mobile Services (CoDAMoS).

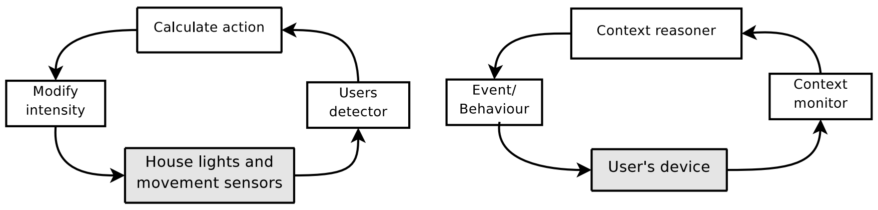

Fig. 1. Feedback loops modelling an automated light system and a context aware mobile device.

or a particular situation such as being busy, in a meeting, etc. The context is constantly monitored by a *context reasoner*, which decides the behaviour of the device in order to react to external events, or to trigger certain events to communicate with other devices.

These simple loops already provide self-adaptability to the house lights system and to the user's mobile device. The former adapts light's intensity according to the detection of users, and the later adapts its behaviour depending on the context. Consider now both models collaborating as a self-organising system. We first extend the house lights system to also monitor context. Having a context reasoner, lights are able to adapt their behaviour not only to users' movement, but also to particular context dependent scenarios. For instance, you do not want to turn on the lights and wake up the kids when they are in the sleeping context. We also add other sensors in order to receive message from users' devices.

Figure 2 depicts the interaction between both loops. User's device monitors the intensity of the lights while still monitors context. Being in the context of *arriving home* may triggers an event to turn on the lights. The context *watching a film* with *high light intensity* may triggers the event of lowing the intensity of the lights.

Since the house lights system is enriched with a context reasoner, some events triggered from user's device may not have always the same result. For instance, turning on the lights when arriving home may not work as expected if kids are in the sleeping context. Like this, two users can communicate through the lights systems as stigmergy. We can also observe that sensors and lights serve as stigmergy for the communication of user's device, and the controller of the house, because both of them monitor the system, and trigger events to modify the intensity of lights.

3 Implementing Feedback Loops

We have started to implement a prototype of the system using Mozart [3], a multi-paradigm programming system implementing the Oz language [4]. We have identified several ways of communicating components of a loop, which can be done using an event-driven approach, or stream communication, which can achieve by pulling or pushing information (lazy or eager execution). To communicate distributed components, message passing seems to be the most appropriated paradigm.

User's devices follow naturally the actor model [5], but inside the actor we can introduce other paradigms as well. For instance, the context reasoner applies a set of rules to

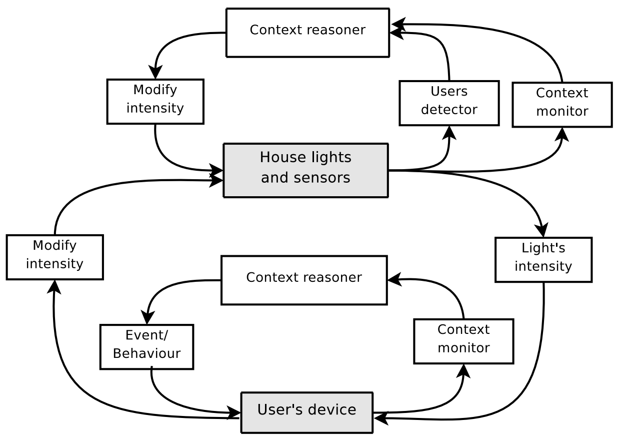

Fig. 2. Communicating two feedback loops.

the monitor information in order to determine the correspondent rule. This component fits better logic or declarative programming. To implement adaptive behaviour, we have chosen a model representing roles [6], where split objects [7] are used as the general architecture.

Since every component communicate with other by events or messages, they are quite independent, and the decision of the implementation of each of them, do no affect the implementation of the others. We still need to investigate more about the explicitness of the components matching the design and the implementation, because sometimes they appear clearly at the conceptual level, by they integrated to other components in the implementation.

References

1. Group, I.A.: Ambient intelligence: from vision to reality (2003)
2. Van Roy, P.: Self management and the future of software design. In: Formal Aspects of Component Software (FACS '06). (2006)
3. Consortium, M.: The mozart-oz programming system. *http://www.mozart-oz.org* (2007)
4. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
5. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. of the 3rd IJCAI, Stanford, MA (1973) 235–245
6. Vallejos, J., Ebraert, P., Desmet, B., Cutsem, T.V., Mostinckx, S., Costanza, P.: The context-dependent role model. In Indulska, J., Raymond, K., eds.: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '07). Lecture Notes in Computer Science, Springer-Verlag (2007) 277–299
7. Bardou, D., Dony, C.: Split Objects: a Disciplined Use of Delegation within Objects. In: Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), San Jose, California, USA (1996) 122–137

C A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks

A Relaxed-Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks

Boris Mejías and Peter Van Roy
Université catholique de Louvain
Louvain-La-Neuve, Belgium
{bmc|pvr}@info.ucl.ac.be

***Abstract*—There is no doubt about the increase in popularity of decentralised systems over the classical client-server architecture in distributed applications. These systems are developed mainly as peer-to-peer networks where it is possible to observe many strategies to organise the peers. The most popular one for structured networks is the ring topology. Despite many advantages offered by this topology, the maintenance of the ring is very costly, being difficult to guarantee lookup consistency and fault-tolerance at all time. By increasing self-management in the system we are able to deal with these issues. We model ring maintenance as a self-organising and self-healing system using feedback loops. As a result, we introduce a novel relaxed-ring topology that is able to provide fault-tolerance with realistic assumptions concerning failure detection. Limitations related to failure handling are clearly identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.**

***Index Terms*—Decentralised systems, Peer-to-peer, Fault-tolerance, Self-management, Feedback-loops**

I. Introduction

Decentralised applications has rapidly increased their popularity in the last years due to several factors and motivations. The increase of Internet bandwidth with a sufficient reliability is already an important element. The fact that home computers have augmented their computing power has decreased the dependency on big servers, because clients are powerful enough to play the role of a server for several tasks. These factors have allowed the introduction of peer-to-peer networks. Such networks have reduced the problem of traffic congestion and single point of failures as in the client-server architecture, making decentralised applications popular.

Building decentralised applications requires several guarantees from the underlay peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Other wished properties such as efficient routing, scalability and full reachability moved randomly connected peer-to-peer networks towards structured overlay networks. Many of these structured networks implements a Distribute Hash Table (DHT). Among many of them - Pastry [1], Tapestry [2], Kademlia [3], HyperCup [4], P-Grid [5] - we focus on Chord [6], because it is quite representative and it introduces a ring topology that has influenced many other networks.

In Chord, peers are organised in a ring, having a set of pointers to efficiently find any other peer in the network. The resources of the system are distributed among the peers where each one is responsible for a set of them. Performing a lookup for a resource must result in a consistent answer, finding the right responsible. To add or remove a peer from the network, the peer only needs to synchronise with its direct neighbours, making the network self-organising. More details are explain in section II.

Despite the self-organising nature of the ring architecture, its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents temporary inconsistency when several peers join the network concurrently. This problem occurs even in fault-free scenarios. To fix these inconsistencies, a stabilisation protocol must be run periodically. The system must also deal with peers gently leaving the network, which can occur massively and concurrent to other joining events. The most challenging issue though, is failure handling, where peers just leave the network breaking the ring without following any protocol.

As we can see, ironically, the advantages of decentralised systems with respect to the classical client-server architecture, have the drawback of higher complexity due to the lack of a single point of control and synchronisation. Increasing self-management of decentralised systems can help us to reduce this new complexity. By self-management we mean the ability of a system to maintain its functionality despite changes in its environment. The system constantly monitor itself triggering corrective actions when the current state deviates from the desired one. In order to achieve self-management, the use of feedback loops in the design of the system appears as a straight forward approach.

We use feedback loops to model the ring-maintenance of our peer-to-peer system, called P2PS [7], which also uses a ring topology. As a result of this new design, we introduce a novel *relaxed-ring* topology that simplifies the "join" algorithm and greatly improves failure recovery. Having the ability of handling failures, there is no need for a "leave" algorithm, because this case is already covered by failure recovery.

The main contribution of this work is the design of a peer-to-peer network as a self-managing system, introducing a relaxed-ring topology that is able to provide fault-tolerance with realistic assumptions concerning failure detection. The use of feedback loops for modelling the system can be reused not only in other decentralised systems, but also in software design in general.

Section II gives a more detailed introduction to peer-to-peer networks using ring topology, describing some existing solutions for ring maintenance. Section III briefly introduces feedback loops for self-managing systems and how they can be applied to software design. The result of applying feedback loops to the ring maintenance is given in section IV with a detailed description of the *relaxed-ring*. After a deep analysis of failure handling, the paper provides conclusions for this work.

II. Peer-to-Peer Rings

Peer-to-peer networks appear as the evident framework for working with decentralised systems. Looking at the history of peer-to-peer systems, we find Napster [8] as the icon of the first generation. Napster uses a hybrid architecture with a centralised directory of the location of the resources of the systems. A client-server strategy was needed in order to find other peers.

A second generation characterised by Gnutella [9] and FreeNet [10] removed the servers from the topology becoming the first real peer-to-peer network. Peers build an overlay network on top of the Internet, being able to route with its own topology. No structure is used for the network because peers are connected randomly to other peers. Therefore, no strong guarantees can be provided with respect to reachability, time to find items or availability. Unfortunately, these kind of network have limited scalability and induce a huge amount of traffic [11].

Structured overlay networks - see introduction for references - appear as the third generation of peer-to-peer systems, claiming self-organisation of the network with fault-tolerance in addition to the guarantees that cannot be found in the second generation.

Figure 1 depicts a structured overlay network using ring topology and providing a Distributed Hash Table (DHT) with election of fingers based on the Tango [12] algorithm. This structure was first introduced by Chord [6]. Every peer is identified with a hash key, and it is connected to a successor and a predecessor respecting the order of the keys in clockwise direction. The DHT is used for storing and finding items in the network using basically two operations: $put(key, value)$ to store a value with a certain key, and $get(key)$ to recover the value. Every peer is responsible for all keys between its predecessor's identifier and itself, excluding of course the predecessor to avoid overlapping. When a lookup for a key is trigger from any part of the ring, consistency must be guaranteed, i.e., only one responsible for the key must be found.

As we mentioned already, ring maintenance is costly and it is not trivial to guarantee correctness. Chord's algorithms for ring maintenance handling *joins* and *leaves* present well known problems of temporary inconsistency, where more that one peer appears to be the responsible for the same key. For this reason, Chord needs to trigger periodic stabilisation in order to fix the inconsistencies. Existing analyses [13] conclude that the problem comes from the fact that joins and leaves

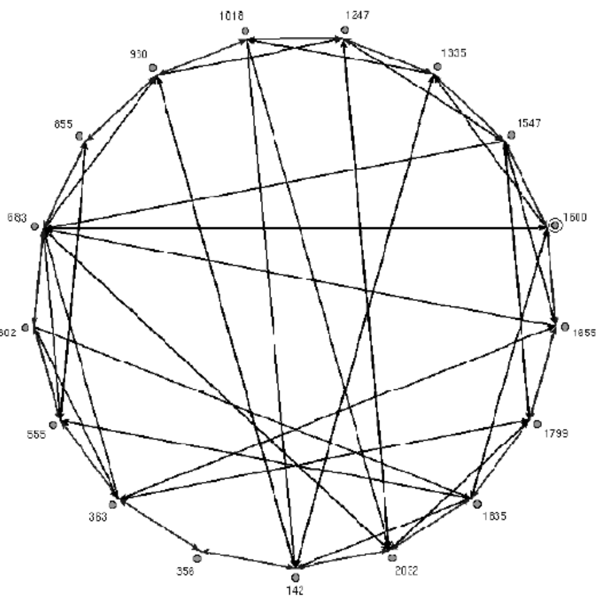

Fig. 1. Structured overlay network using ring topology

are not atomic operations. We also raise the issue that these operations always need the synchronisation of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [14], [15] introduce locks in the algorithms in order to provide atomicity to the *join* and *leave* operations. Locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free systems, which is not realistic.

A better solution is provided by DKS [13], simplifying the locking mechanism and proving correctness of the algorithms in absent of failures. Even when this approach offers strong guarantees, we consider locks extremely restrictive for a dynamic network based on asynchronous communication. Every lookup request involving the locked peers must be suspended in presence of join or leave in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted with the relevant locks. Given that, peers crashing can be seen as peers just leaving the network without respecting the protocol of the locking mechanism breaking the guarantees of the system.

Another critical problem for performance is presented when a peer crashes while some joining or leaving peer is holding its lock. The situation is worse when the peer holding the relevant lock is the one that crashes. Under this considerations, we can observer that locks in a distributed system can hardly present an efficient fault-tolerant solution.

III. Feedback Loops

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

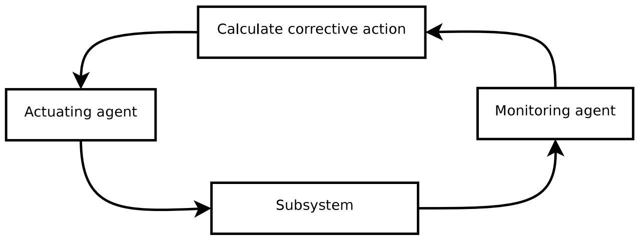

Fig. 2. Basic structure of a feedback loop (taken from [16])

be found in [16], 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 2 depicts the interaction of these three concurrent components in a feedback loop. These three components together with the subsystem forms the entire system.

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.

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 2, is because that is a description of the model, and not the model applied to a system.

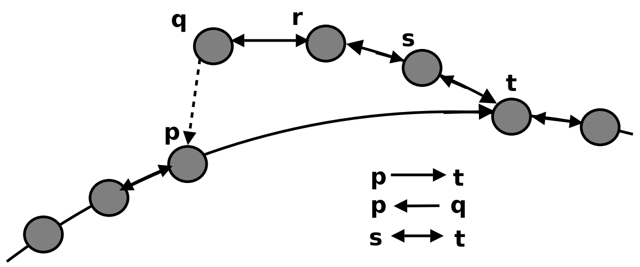

Fig. 3. Branch created due to connection problems between peers p and q

IV. Self-Organising and Self-Healing Relaxed-Ring

Section II described the problem of guaranteeing consistent lookup while multiple joins, leaves and failures occur in a peer-to-peer network using ring architecture. As a solution to this problem we design a novel topology based on a relaxed-ring. This topology also allows as to provide failure recovery using imperfect failure detectors and handling broken links which are often ignored. The relaxed-ring topology is part of the new version of P2PS [17], implemented with Mozart-Oz programming system [18].

During this section we will use the terms *peer* and *node* indistinctly to refer to an independent process running with its own address space, i.e., a network node. We also use the term *pointer* as a network reference to a node. The terms *key* and *identifier* represent keys from the DHT, and they are used to identify peers.

The algorithms of the relaxed-ring are designed using feedback loops, and the description of their implementation is given using event-driven notation. 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. The ring provides a DHT with key-distribution formed by integers from 0 to N growing clockwise.

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 by 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. The lookup consistency is guaranteed between every step and therefore, the network can still answer lookup requests while simultaneous peers are joining the network. Another relevant difference with the mentioned related work, is that we do not rely on graceful leaving of peers, because anyway, we have to deal with leaves due to network and node failures.

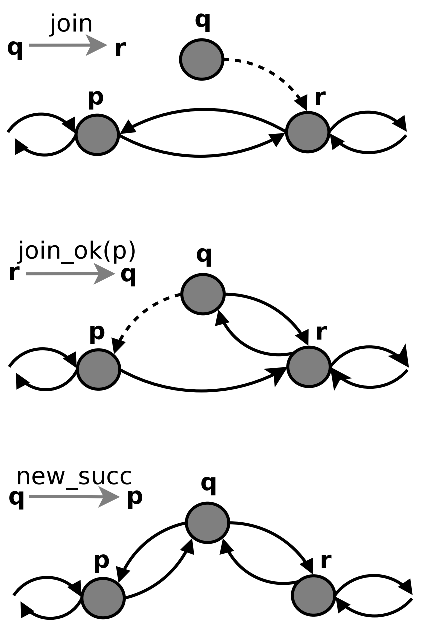

Fig. 4. Messages and pointers update during a join

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 plus 1, 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 *core ring*. When there are no branches, and every peer is connected bidirectionally with its successor and predecessor, then we have a *"perfect ring"*.

Figure 3 shows a fraction of a relaxed ring where peer t is the root of a branch, and where the connection between peers p and q is broken. We say that p and t belongs to the *core ring*, and that q, r and s are part of a branch.

Before starting the description of the algorithms that maintain the relaxed-ring topology, we first define what 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.*

A. The join algorithm

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 4. 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 5 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.

Every peer is independently monitoring the network, and the correcting action performing the ring maintenance is running concurrently in every peer. As events triggered by peers are monitored by other peers, we observe that they use the network as a mean for communicating using stigmergy.

Algorithm 1 Join step 1 - adding a new node

```
upon event ⟨ $join$ | q ⟩ do
    if succ = nil then
        send ⟨ $try\_later$ | self ⟩ to $q$
    else
        if betterPredecessor(q) then
            oldp := pred
            pred := q
            predlist := {oldp} ∪ {predlist}
            send ⟨ $join\_ok$ | oldp, self, succlist ⟩ to $q$
        else if ($i < pred$) then
            send ⟨ $goto$ | pred ⟩ to $q$
        else
            send ⟨ $goto$ | succ ⟩ to $q$
        end if
    end if
end event

upon event ⟨ $join\_ok$ | p, r, sl ⟩ do
    succ := r
    succlist := {r} ∪ sl
    if ($pred = nil$) ∨ ($p \in (pred, self)$) then
        pred := p
        send ⟨ $new\_succ$ | self, succ, succlist ⟩ to $p$
    end if
end event
```

Algorithm 1 describes one implementation of the feedback loop. Every event is handled by the computing component

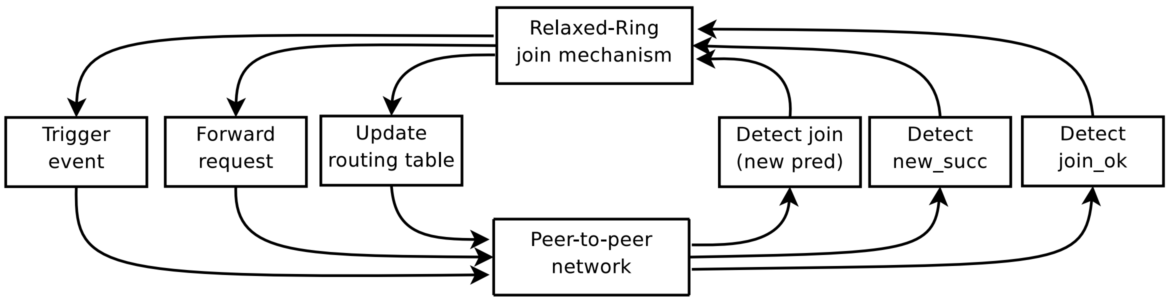

Fig. 5. Join algorithm as a feedback loop

running in every peer. This component decides which correction has to be performed. In event $join$, the messages $goto$ and try_later represent the forwarding of the request. The request can be accepted when the joining peer is a $betterPredecessor$. This is the case when $q \in (pred, self]$. As part of the joining process, there is an update of the routing table. This update is done explicitly by assigning the corresponding pointer $pred$ and the $pred\_list$.

Operator **send** is a reliable point-to-point send. If the receiver presents a failure before the message arrives, the sender is notified.

Triggering correcting events is represented by the message $join\_ok$, which will be monitored by the joining peer. Handling event $join\_ok$ also shows how the routing table is updated by assigning pointer $succ$ and set $succ\_list$. A second correcting event is triggered: new_succ. The set named $succ\_list$ is used later for failure recovery. This set represents the list of peers that follows after the current successor. This peers can be contacted in order to fix the ring when the successor is suspected of having a failure.

Note that the algorithm is divided into two steps. Like this, we do not need the synchronisation of three peers performing an atomic operation. Instead, two correcting actions are triggered in order to fix the perturbation. Algorithm 2 describes the implementation of the second action where the ring is closed again. This is achieved by updating pointer $succ$ and set $succ\_list$, which are part of the routing table. A notification event $join\_ack$ is triggered to improve the knowledge of the system about its global state, but it is not strictly needed.

It is important to signalise that the routing algorithm of Chord or DKS [19] cannot be used in the relaxed-ring. The algorithm would creates cycles due to the introduction of branches in the ring topology. The routing algorithm of the relaxed-ring works as follows. A peer i must choose the closest peer j to the key k from its routing table. The routing table also consider predecessors for routing. The distance function between two keys is given by $d(k, j) = (j - k) mod N$, where N is the highest value of the key domain.

Given the join and routing algorithms, the relaxed-ring guarantees consistent lookup at any time in presence of multiple joining peers. To prove this guarantee, let us assume the contrary. Then, there are two peers p and q responsible for key k. In order to have this situation, p and q must have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$. The join algorithm updates the predecessor pointer upon events $join$ and $join\_ok$. In the event $join$, the predecessor is set to a new joining peer j. This means that no other peer was having j as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event $join\_ok$, the joining peer j initiates its responsibility having a member of the ring as predecessor, say i. The only other peer that had i as predecessor before is the successor of j, say p, which is the peer that triggered the $join\_ok$ event. This message is sent only after p has updated its predecessor pointer to j, and thus, modifying its responsibility from $(i, p]$ to $(j, p]$, which does not overlap with j's responsibility $(i, j]$. Therefore, it is impossible that two peers has the same predecessor.

Algorithm 2 Join step 2 - Closing the ring

```
1:  upon event ⟨ $new\_succ$ | q, olds, sl ⟩ do
2:      if ($succ = olds$) then
3:          oldsucc := succ
4:          succ := q
5:          succlist := {q} ∪ sl
6:          send ⟨ $join\_ack$ | self ⟩ to $oldsucc$
7:          send ⟨ $upd\_succlist$ | self, succlist ⟩ to $pred$
8:      end if
9:  end event

10: upon event ⟨ $join\_ack$ | op ⟩ do
11:     if ($op \in predlist$) then
12:         predlist := predlist \ {op}
13:     end if
14: end event
```

B. Resilient information

During the join algorithm 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

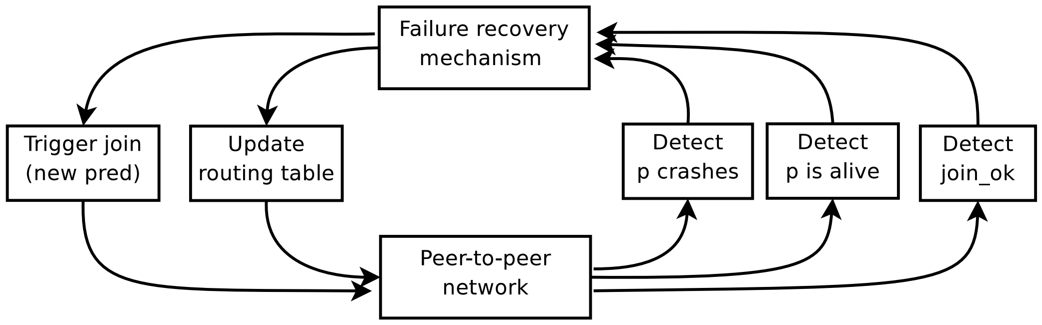

Fig. 6. Failure recovery as a feedback loop

no predecessor that triggers the recovery mechanism. This is expected to happen only when the tail of a branch has crashed. Section IV-C gives more details about the recovery algorithms. Initially, we do not use extra fingers for recovery because it is not efficient. They may help to solve network partitioning, but we delegate this kind of recovery to upper layers of P2PS.

Algorithm 3 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 3 Update of successor list

```
1: upon event ⟨ $upd\_succlist$ | s, sl ⟩ do
2:     newsl := {s} ∪ sl \ getLast(sl)
3:     if ($s == succ$) ∧ ($succlist \neq newsl$) then
4:         succlist := newsl
5:         send ⟨ $upd\_succlist$ | self, succlist ⟩ to $pred$
6:     end if
7: end event
```

C. 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 detectors 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 re-use 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 loop can be observe in figure 6.

Algorithm 4 describes an implementation of the feedback loop. If a failure is detected, the $crash$ event is triggered. The detected node is removed from the 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. A successor candidate is taken from the successors list. 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. Note that as every crashed peer is immediately removed from the resilient sets, $getFirst$ always returns 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. This action generates an interaction between the two feedback loops, using the involved peers as stigmergy for communication. If the successor candidate also fails, a new candidate will be chosen. This is verified with the if condition.

Algorithm 4 Failure recovery

```
1: upon event ⟨ $crash$ | p ⟩ do
2:     succlist := succlist \ {p}
3:     predlist := predlist \ {p}
4:     chrased := {p} ∪ crashed
5:     if ($p = succ$) ∨ ($p = succ\_candidate$) then
6:         succ := nil
7:         succ_candidate := getFirst(succlist)
8:         send ⟨ $join$ | self ⟩ to $succ\_candidate$
9:     else if ($p == pred$) then
10:        if ($predlist \neq \emptyset$) then
11:            pred_candidate := getLast(predlist)
12:        end if
13:    end if
14: end event

15: upon event ⟨ $alive$ | p ⟩ do
16:     crashed := crashed \ {p}
17: end event
```

When the detected peer p is the predecessor, no recovery mechanism is triggered because p's predecessor will contact the current peer. The algorithm decides a predecessor candidate from the $predlist$ to recover from the case when the tail of a branch is the crashed peer. We will not explore this case further in this paper because it does not violate our definition of consistent lookup. To solve it, it is necessary to set up a time-out to replace the faulty predecessor by the predecessor candidate.

When a peer recovers from a temporary failure, the $alive$ event is triggered. This can be implemented by using watchers or using a fault stream attached to the distributed entities [20]. To handle the *alive* event is enough to remove the peer from the $crashed$ set. This will terminate any pending recovery algorithm. The faulty peer will trigger by itself the corresponding recovery events with the relevant peers.

Algorithm 5 Verifying predecessor candidate

```
1: function betterPredecessor($q$) is
2:     if ($q \in (pred, self)$) then
3:         return ($true$)
4:     else
5:         return ($pred \in crashed$)
6:     end if
7: end function
```

Having now the knowledge of the $crashed$ set, algorithm 5 gives a complete definition of the function $betterPredecessor$ used in algorithm 1. Since the $join$ event is used both for a regular join and for failure recovery, the function will decide 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.

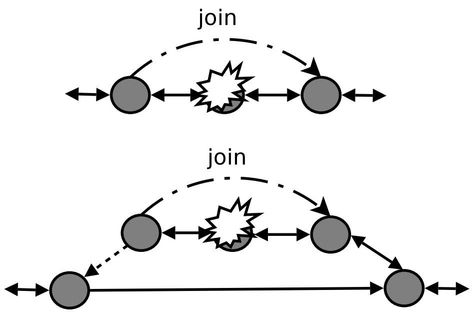

Fig. 7. Failure recovery triggered in the ring and in a branch

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example of figure 4 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.

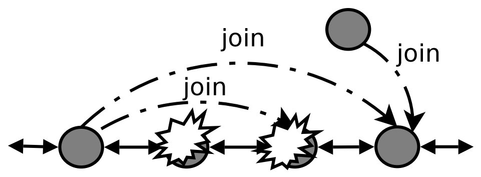

Fig. 9. Simultaneous crashes together with a join event

Figure 7 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 bellow shows that a crash in a branch is equivalent as long as there is a predecessor that detects the failure.

Figure 9 shows two simultaneous crashes together with a new peer concurrently joining the network. 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 be forwarded to the new peer. The contact of these two peers will finally fix the ring and removing the branch.

There are failures more difficult to handle than the ones we have already analysed. Figure 10 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 reaches accuracy. This will happen when the link is recover from the failure, and the *alive* event is monitored.

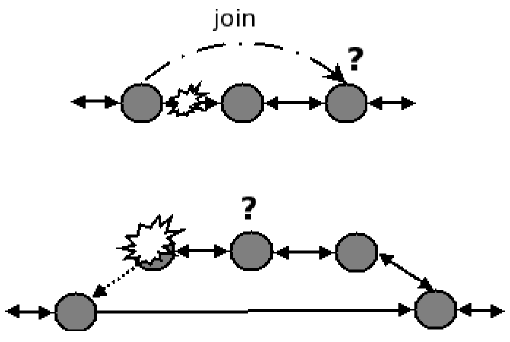

Fig. 10. Broken link and failure of the tail of branch

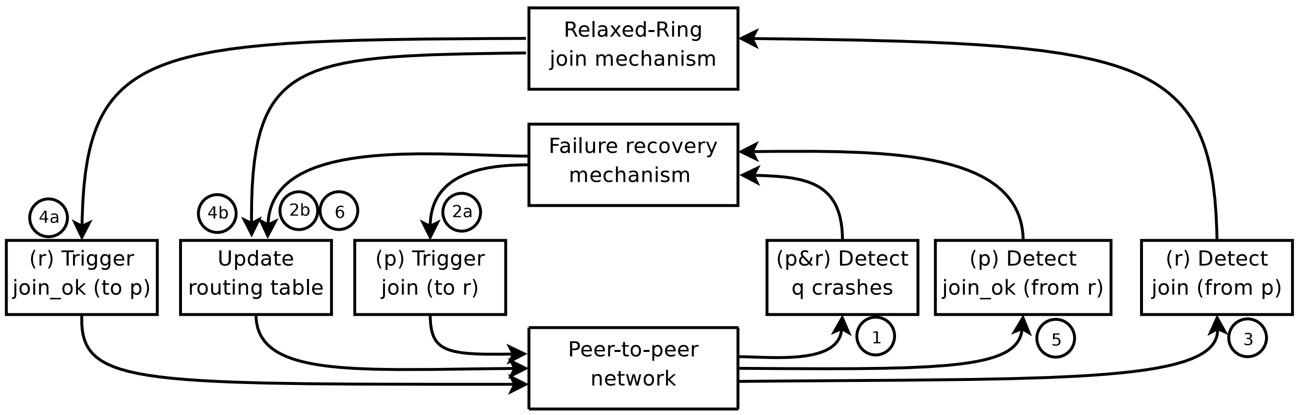

Fig. 8. Peers p and r detect failure of q, fixing the ring with an interaction of feedback loops

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, which will be discussed in section IV-E.

With respect to failure handling, the relaxed-ring guarantees that simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition. To prove this guarantee, we must consider that every failure of a peer 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.

D. Combining feedback loops

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. Feedback loops may communicate directly or using some subsystem as stigmergy. Let us consider a particular section of the ring having peers p, q and r connected through successor and predecessors pointers. Figure 8 describes how the ring is perturbed and stabilised in the presence of a failure of peer q. Only relevant monitored and actuating actions are included in the figure to avoid a bigger and verbose diagram.

Initially, the crash of peer q is detected by peers p and r (1). Both peers will update their routing tables removing q from the set of valid peers (2a). But, since p is q's predecessor, only p will trigger the correcting event $join$ (2b). 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). Once again the network is used as stigmergy, because 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.

E. Limitations

Figure 11 depicts a temporary network partition that can occur in the relaxed-ring topology. Previously, we have analysed cases where 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. This situation introduces a temporary inconsistency. This limitation is not unique to the relaxed-ring topology. It is related to the proof given by Ghodsi in [13], where it is not possible to provide at the same time consistency, availability and partition-tolerance in presence of network partitioning. The limitation of the particular case of the relaxed-ring is well defined in the following theorem.

Theorem 4.1: Let r be the root of a branch, $succ$ its successor, $pred$ its predecessor, and $predlist$ the set of peers having r as successor. Let p be any peer in the set, so that $p \in predlist$. Then, the crash of peer r may introduce temporary inconsistent lookup if p contacts $succ$ for recovery before $pred$. The inconsistency will involve the range $(p, pred]$, and it will be corrected as soon as $pred$ contacts $succ$ for recovery.

Proof: There are only two possible cases. First, $pred$ contacts $succ$ before p does it. In that case, $succ$ will consider $pred$ as its predecessor. When p contacts $succ$, it will redirect

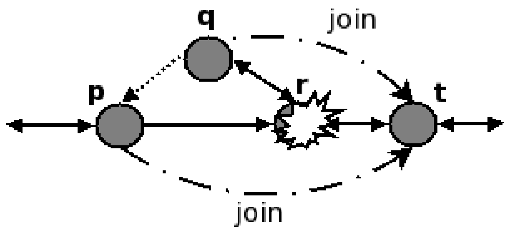

Fig. 11. The failure of the root of a branch triggers two recovery events

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 4.1 clearly states the limitation of branches in the system. This helps developers to identify the scenarios where special failure recovery must be taken into account. 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.

V. Conclusions

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 paper we show how feedback-loops, taken from existing self-managing systems, can be applied in the design of a peer-to-peer network. The result is a novel relaxed-ring topology for fault-tolerant and self-organising networks. 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.

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 help the system to work in realistic scenarios. The ability to handle failures removes the need for a *leave* algorithm, because it is just a special case in the failure recovery mechanism.

Related work is discussed along the paper, but it is specially analysed in section II. The guarantees and limitations of the relaxed-ring of P2PS are clearly identified and formally stated in section IV. These specifications provide helpful indications to developers in order to build fault-tolerant applications on top of this structured overlay network.

Acknowledgement

This work is being funded and developed in the context of projects EVERGROW (contract number:001935) and SELFMAN (contract number: 034084).

References

[1] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” *Lecture Notes in Computer Science*, vol. 2218, pp. 329–??, 2001. [Online]. Available: citeseer.ist.psu.edu/rowstron01pastry.html

[2] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A global-scale overlay for rapid service deployment,” *IEEE Journal on Selected Areas in Communications*, 2003, special Issue on Service Overlay Networks, to appear. [Online]. Available: citeseer.ist.psu.edu/article/zhao03tapestry.html

[3] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” 2002. [Online]. Available: citeseer.ist.psu.edu/maymounkov02kademlia.html

[4] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl, “Hypercup – hypercubes, ontologies and efficient search on p2p networks,” 2002. [Online]. Available: citeseer.ist.psu.edu/article/schlosser02hypercup.html

[5] K. Aberer, “P-Grid: A self-organizing access structure for P2P information systems,” *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science*, vol. 2172, pp. 179–194, 2001. [Online]. Available: citeseer.ist.psu.edu/aberer01pgrid.html

[6] 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*, 2001, pp. 149–160. [Online]. Available: citeseer.ist.psu.edu/stoica01chord.html

[7] V. Mesaros, B. Carton, and P. Van Roy, “P2PS: Peer-to-peer development platform for mozart.” in *MOZ*, ser. Lecture Notes in Computer Science, P. Van Roy, Ed., vol. 3389. Springer, 2004, pp. 125–136.

[8] Napster, “Open source napster server,” 2002. [Online]. Available: http://opennap.sourceforge.net

[9] Gnutella, “http://gnutella.com,” 2003.

[10] FreeNet, “http://freenet.sourceforge.net,” 2003.

[11] E. 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. [Online]. Available: citeseer.ist.psu.edu/article/markatos02tracing.html

[12] B. Carton and V. Mesaros, “Improving the scalability of logarithmic-degree dht-based peer-to-peer networks.” in *Euro-Par*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds., vol. 3149. Springer, 2004, pp. 1060–1067.

[13] A. Ghodsi, “Distributed k-ary System: Algorithms for distributed hash tables,” PhD Dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[14] X. Li, J. Misra, and C. G. Plaxton, “Active and concurrent topology maintenance.” in *DISC*, 2004, pp. 320–334.

[15] ——, “Concurrent maintenance of rings.” *Distributed Computing*, vol. 19, no. 2, pp. 126–148, 2006.

[16] P. Van Roy, “Self management and the future of software design,” in *Formal Aspects of Component Software (FACS ’06)*, September 2006.

[17] DistOz Group, “P2PS: A peer-to-peer networking library for Mozart-Oz,” *http://gforge.info.ucl.ac.be/projects/p2ps*, 2007.

[18] Mozart Community, “The Mozart-Oz programming system,” http://www.mozart-oz.org, 2007.

[19] L. O. 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,” in *CCGRID ’03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 344.

[20] R. Collet and P. V. Roy, “Failure handling in a network-transparent distributed programming language.” in *Advanced Topics in Exception Handling Techniques*, 2006, pp. 121–140.