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

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

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

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

1 Introduction

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

The SELFMAN project is tackling one part of this application space: large-scale distributed systems based on structured overlay networks. Overlay networks are already self managing in the lower layers: they self organize around failures to provide efficient and reliable routing and lookup. We are building a service architecture on top of the overlay network using an advanced component model. To make it self managing, the service architecture is designed as a set of interacting feedback loops. We are building one major service, a distributed transactional storage, that we are using to build two application demonstrators: a distributed Wiki and a media streaming application.

In the rest of this paper, we motivate the need for self-managing systems and we give an overview of our ideas and contributions. The paper is structured as follows:

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

2 Motivation

2.1 Software complexity

Software is fragile. A single bit error can cause a catastrophe. Hardware and operating systems have been reliable enough in the past so that this has not unduly hampered the quantity of software written. Hardware is verified to a high degree. It is much more reliable than software. Good operating systems provide strong encapsulation at their cores (virtual memory, processes) and this has been

polished over many years. New techniques in fault tolerance (e.g., distributed algorithms, Erlang) and in programming (e.g., structured programming, object-oriented programming, more recent methodologies) have arguably kept the pace so far. In fact we are in a situation similar to the Red Queen in Through the Looking-Glass: running as hard as we can to stay in the same place [7].

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

The increase in complexity brings with it a host of problems that must be overcome. For example, one problem is that software errors cannot be eliminated [2, 41]. We have to cope with them. There are many other problems: scale (large numbers of independent nodes), partial failure (part of the system fails, the rest does not), security (multiple security domains) [20], resource management (conflicting demands for limited resources), performance (harnessing multiple nodes or spreading load), and global behavior (emergent behavior of the system as a whole). Of all these problems, global behavior is particularly relevant because it is often the primary reason that the system was built. Experiments show that large networks exhibit global behavior that is not easily predicted by the behaviors of the individual nodes (e.g., the power grid [11]). An important question is therefore how to design a system with a desired global behavior.

2.2 Self-managing systems

What solution do we propose for building a complex software system that overcomes these problems and that has a desired global behavior? For inspiration, we go back fifty years, to the first work on cybernetics and system theory: designing systems that regulate themselves [40, 4, 5]. A *system* is a set of components (called subsystems) that are connected together to form a coherent whole. Can we predict the system's behavior from its subsystems? Can we design a system with desired behavior? These questions are particularly relevant for the distributed systems we are interested in. No general theory has emerged yet from this work. We do not intend to develop such a theory in SELFMAN. Our aim is narrower: to build self-managing *software* systems. Such systems have a chance of coping with the new complexity. Our work is complementary to [19], which

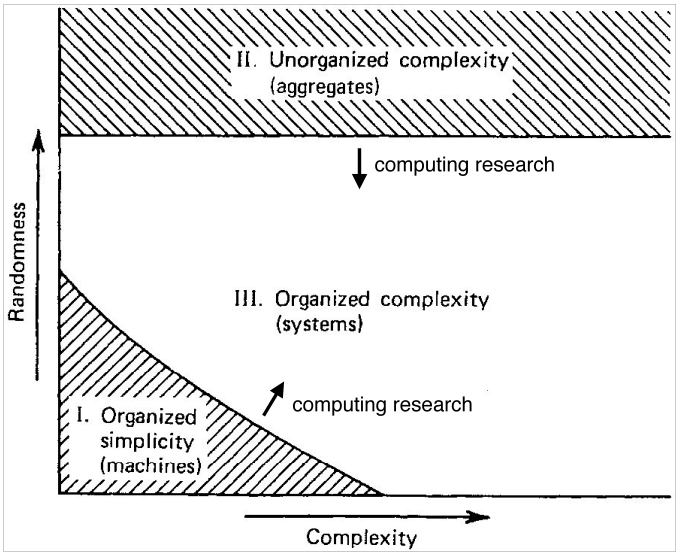

Fig. 1. Randomness versus complexity (taken from Weinberg [38])

applies control theory to design computing systems with single feedback loops. We are interested in distributed systems with many interacting feedback loops.

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

Figure 1 (taken from [38]) classifies systems according to two axes: their complexity (the number of components and interactions) and the amount of randomness they contain (how unpredictable the system is). There are two shaded areas that are understood by modern science: machines (organized simplicity) and aggregates (unorganized complexity). The vast white area in the middle is poorly understood. We extend the original figure of [38] to emphasize that computing research is the vanguard of system theory: it is pushing inwards the boundaries of the two shaded areas. Two subdisciplines of computing are particularly relevant: programming research (developing complex programs) and computational science (designing and simulating models). In SELFMAN we do both: we design

algorithms and architectures and we simulate the resulting systems in realistic conditions.

2.3 Designing self-managing software systems

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

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

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

Most products of human civilization need an implicit management feedback loop, called "maintenance", done by a human. For example, changing lightbulbs, replacing broken windows, or tanking a car. Each human mind is at the center of an enormous number of these feedback loops. The human brain has a large capacity for creating such loops; they are called "habits" or "chores". Most require very little conscious awareness. Repetition has caused them to be programmed into the brain below consciousness. However, if there are too many feedback loops to manage then the brain is overloaded: the human complains that "life is too complicated"! We can say that civilization advances by reducing the number of feedback loops that have to be explicitly managed [39]. A dishwashing machine reduces the work of washing dishes, but it needs to be bought, filled and emptied, maintained, replaced, etc. Is it worth it? Is the total effort reduced?

Software is in the same situation as other products of human civilization. In the current state, most software products are very fragile: they require frequent maintenance by a human. This is one of the purposes of SELFMAN: to reduce

this need for maintenance by designing feedback loops into the software. This is a vast area of work; we have decided to restrict our efforts to large-scale distributed systems based on structured overlay networks. Because they have low-level self management built in, we consider them an ideal starting point. SONs have greatly matured since the first work in 2001 [36]; current SONs are (almost) ready to be used in real systems. We are adapting them in two directions for SELFMAN. First, we are extending the SON algorithms to handle important network issues that are not handled in the SON literature, such as network partitioning (see Section 5). Second, we are rebuilding the SON using a component model [1]. This is needed because the SON algorithms themselves have to be managed and updated while the SON is running, for example to add new basic functionality such as load balancing or new routing algorithms. The component model is also used for the other services we need for self management.

3 The SELFMAN project

The SELFMAN project is building a decentralized service architecture and two demonstrator applications that use the architecture. In this section we introduce the service architecture and the demonstrator applications. We also mention two important inspirations of SELFMAN: IBM's Autonomic Computing Initiative and the Chord system. Section 4.3 explains how the service architecture is used as a basis for self management.

3.1 Decentralized service architecture

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

- SONs already provide low-level self-management abilities. We are reimplementing our SONs using a component model that adds lifecycle management and hooks for supporting services. This makes the SON into a substrate for building services.
- The component model is based on concurrent components and asynchronous message passing. It uses the communication and storage abilities of the SON to enable it to run in a distributed setting. Because the system may need to update and reorganize itself, the components need introspection and reconfiguration abilities. We have designed a process calculus, Oz/K, that has these abilities in a practical form [25].

This leads to a simple service architecture for decentralized systems: a SON lower layer providing robust communication and routing services, extended with other basic services and a transaction service. Applications are built on top of this service architecture. The transaction service is important because many realistic application scenarios need it (see Section 3.2).

The structured overlay network is the base. It provides guaranteed connectivity and fast routing in the face of random failures (Section 5). It does not protect

against malicious failures: our current design is limited in that we must consider the network nodes as trusted. We are exploring how to modify the overlay network to better address security issues; one possibility is to use a small-world network [17]. We assume that untrusted clients may use the overlay as a basic service, but cannot modify its algorithms. See [45] for more on security for SONs and its effect on SELFMAN.

We have designed and implemented robust SONs based on the DKS, Chord#, and Tango protocols [13, 32, 8]. These implementations use different styles and platforms, for example DKS is implemented in Java and uses locking algorithms for node join and leave. Tango is implemented in Oz and uses asynchronous algorithms for managing connectivity (Section 5.2). We have also designed an algorithm for handling network partitions, which is an important failure mode for structured overlay networks. Network partitioning is handled by a merge algorithm that combines the partitioned subrings back into a single ring (Section 5.1).

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

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

Table 1. Requirements for selected self-managing applications

3.2 Demonstrator applications and guidelines

The design of the self-management architecture was guided by four application scenarios [12]. Table 1 lists these scenarios and what they need in four areas: self-* properties, components, overlay networks (decentralized execution), and transactions. Two pluses (++) mean strong need and one plus (+) means some need. An empty space means no need for that area. All these applications have a strong need for self-management support. Out of these four scenarios, we are building two application demonstrators:

– A distributed Wiki application (specified by the Zuse Institute Berlin). This is a Wiki (a user-edited set of interlinked Web pages) that is distributed over a SON using transactions with versioning and replication, supporting both

editing and search. Our prototype of this application won first prize in the First IEEE International Scalable Computing Challenge (SCALE 2008) [30].
– An on-demand media streaming application (specified by Peerialism). This application provides distributed live media streams with quality of service to large and dynamically varying numbers of customers. Dynamic reconfiguration is needed to handle the fluctuating structure. This application will become a product of the Peerialism company.

The table shows two other applications that were initially considered but subsequently dropped: a machine-to-machine messaging application (specified by France Télécom) and a J2EE application server (specified by Bull). The messaging application was dropped because of resource limitations in the project. The application server was dropped because it does not have any requirements for decentralized execution.

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

3.3 Related work

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

– IBM's Autonomic Computing Initiative [21]. This initiative started in 2001 and aims to reduce management costs by removing humans from low-level system management loops. The role of humans is then to manage policy and not to manage the mechanisms that implement it.
– Structured overlay network research. The most well-known SON is the Chord system, published in 2001 [36]. Other important early systems are Ocean Store and CAN. Inspired by popular peer-to-peer applications, these systems led to much active research in SONs, which provide low-level self management of routing, storage and smart lookup in large-scale distributed systems.

There is other important related work in ambient and adaptive computing and in biophysics on how biological systems regulate and adapt themselves. For example, [23] shows how systems consisting of two coupled feedback loops behave in a biological setting.

4 Understanding and designing feedback structures

A self-managing system consists of a large set of interacting feedback loops. We want to understand how to build systems that consist of many interacting

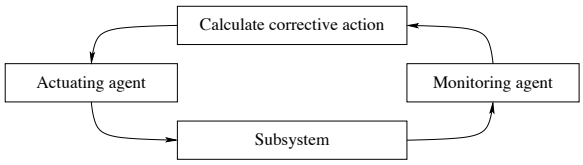

Fig. 2. A feedback loop

feedback loops. Systems with one feedback loop are well understood, see, e.g., the book by Hellerstein *et al* [19], which shows how to design computing systems with feedback control, for example to maximize throughput in Apache HTTP servers, TCP communication, or multimedia streaming. The book focuses on regulating with single feedback loops. Systems with many feedback loops are quite different. To understand them, we start by doing explorations both in analysis and synthesis: we study existing systems (e.g., biological systems) and we design decentralized systems based on SONs.

A feedback loop consists of three parts that interact with a subsystem (see Figure 2): a monitoring agent, a correcting agent, and an actuating agent. The agents and the subsystem are concurrent components that interact by sending each other messages. We call them "agents" because they can be considered as independent entities in the feedback loop; an agent can of course have subcomponents. As explained in [37], feedback loops can interact in two ways:

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

How can we design systems with many feedback loops that interact both through stigmergy and management? We want to understand the rules of good feedback design, in analogy to structured and object-oriented programming. Following these rules should give us good designs without having to laboriously analyze all possibilities. The rules can tell us what the global behavior is: whether the system converges or diverges, whether it oscillates or behaves chaotically, and what states it settles in.

We start by studying existing feedback loop structures that work well, in both biological and software systems. We then try to understand these systems by analysis and by simulation. Many feedback systems and feedback patterns have been investigated in the literature [37, 29, 24]. Section 4.1 gives two nontrivial examples from biology. Section 4.2 then presents one approach to analyze these kinds of systems. Section 4.3 outlines a tentative methodology for designing feedback structures. Finally, we address the issue of multiple users that may have conflicting goals. Section 4.4 explains one approach, called collective intelligence, to manage users with conflicting goals.

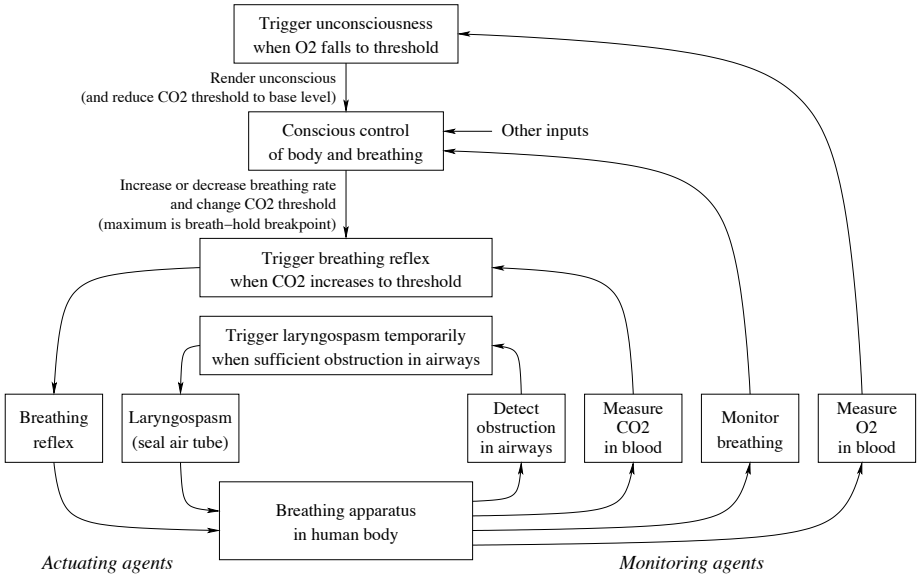

Fig. 3. The human respiratory system

4.1 Feedback structures in the human body

We investigate two feedback loop structures that exist in the human body: the human respiratory system and the human endocrine system. We then make some observations on the computational architecture of the human endocrine system.

Human respiratory system Figure 3 (taken from [37]) shows the human respiratory system, which has four feedback loops: three are arranged in a management hierarchy and the fourth interacts with them through stigmergy. This design works quite well. Laryngospasm can temporarily interfere with the breathing reflex, but after a few seconds it lets normal breathing take over. Conscious control can modulate the breathing reflex, but it cannot bypass it completely: in the worst case, the person falls unconscious and normal breathing takes over. We can already infer several design rules from this system: one loop managing another is an example of data abstraction, loops can avoid interference by working at different time scales, and since complex loops (such as conscious control) can have an unpredictable effect (they can be either stabilizing or unstabilizing) it is a good idea to have an outer "fail-safe" management loop. Conscious control is a powerful problem solver but it needs to be held in check.

Human endocrine system The respiratory system is a simple example of a feedback loop structure that works; we now give a more complex biological example, namely the human endocrine system (shown in part in Figure 4) [10]. The

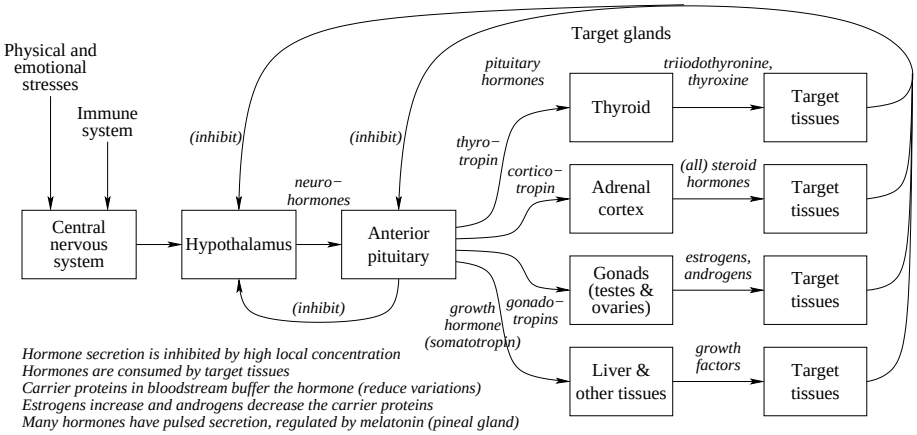

Fig. 4. The hypothalamus-pituitary-target organ axis (in human endocrine system)

endocrine system regulates many quantities in the human body. It uses chemical messengers called *hormones* which are secreted by specialized glands and which exercise their action at a distance, using the blood stream as a diffusion channel. By studying the endocrine system, we can obtain insights in how to build large-scale self-regulating distributed systems. There are many feedback loops and systems of interacting feedback loops in the endocrine system. It provides homeostasis (stability) and the ability to react properly to environmental stresses. Much of the regulation is done by simple negative feedback loops. For example, the glucose level in the blood stream is regulated by the hormones glucagon and insulin. In the pancreas, A cells secrete glucagon and B cells secrete insulin. An increase in blood glucose level causes a decrease in the glucagon concentration and an increase in the insulin concentration. These hormones act on the liver, which releases glucose in the blood. Another example is the calcium level in the blood, which is regulated by parathyroid hormone (parathormone) and calcitonine, also in opposite directions, both of which act on the bone. The pattern here is of two hormones that work in opposite directions (push-pull). This pattern is explained by [23] as a kind of dual negative feedback loop (an NN loop) that improves regulation.

More complex regulatory mechanisms exist in the endocrine system, e.g., the hypothalamus-pituitary-target organ axis. Figure 4 shows its main parts as a feedback structure. This figure is derived from the medical description in [10]. This system consists of two superimposed groups of negative feedback loops (going through the target tissues and back to the hypothalamus and anterior pituitary), a third short negative loop (from the anterior pituitary to the hypothalamus), and a fourth loop from the central nervous system. The hypothalamus and anterior pituitary act as master controls for a large set of other regulatory loops. Furthermore, the nervous system affects these loops through the hypothalamus. This gives a time scale effect since the hormonal loops are slow and the

nervous system is fast. The nervous system's input allows to react quickly to external events.

Figure 4 shows only the main components and their interactions; there are many more parts in the full system. There are more interacting loops, "short circuits", special cases, interaction with other systems (nervous, immune). Negative feedback is used for most loops, saturation (like in the Hill equations introduced in Section 4.2) for others. Realistic feedback structures can be complex. Evolution is not always a parsimonious designer! The only criterion is that the system has to work.

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

4.2 Analysis of feedback structures

How can we design a system with many interacting feedback loops, like the systems of Figure 3 and 4? Mathematical analysis of interacting feedback loops is quite complex, especially if they have nonlinear behavior. Can we simplify the system to have linear or monotonic behavior? Even then, analysis is complex. For example, Kim *et al* [23] analyze biological systems consisting of just two feedback loops interacting through stigmergy. They admit that their analysis only has limited validity because the coupled feedback loops they analyze are parts of much larger sets of interacting feedback loops. Their analysis is based on Matlab simulations using the Hill equations, first-order nonlinear differential equations that model the time evolution and mutual interaction of molecular concentrations. The Hill equations model nonlinear monotonic interaction with saturation effects. We give a simple example using two molecular concentrations X and Y. The equations have the following form (taken from [23]):

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

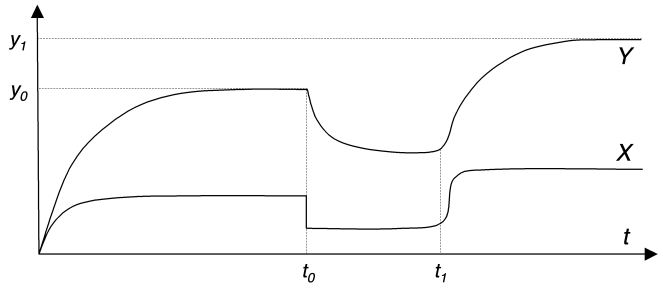

Fig. 5. Example of a biological system where X activates Y

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

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

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

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

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

Many patterns of feedback loops have been analyzed in this way. For example, [29] shows how to model oscillations in biological systems by cycles of feedback loops. The cycle consists of molecules where each molecule activates or inhibits the next molecule in the cycle. If the total effect of the cycle is a negative feedback then the cycle can give oscillations. If a cycle shows oscillatory behavior, then its topology (the molecules involved and their interaction types) can be reconstructed from the observed behavior. Many other patterns have been analyzed as well in biological systems, but there is as yet no general theory for analyzing these feedback structures. In SELFMAN we are interested in investigating the kinds of equations that apply to software. In software, the feedback structures may not follow the Hill equations. For example, they may not be monotonic. Nevertheless, the Hill equations are a useful starting point because they model saturation, which is an interesting form of nonlinearity.

4.3 Feedback structures for self management

From the examples given in the previous sections and elsewhere [37, 4, 40, 5, 38], we can give a tentative methodology for designing feedback structures. We assume that the overall architecture follows the decentralized structure given in Section 3.1: a set of loosely-coupled services built on top of a structured overlay network. We build the feedback structure within this framework. We envisage the following three layers for a self-managing system:

1. *Components and events.* This basic layer corresponds to the service architecture of Section 3.1: services based on concurrent components that interact through events [1, 9]. There can be publish/subscribe events, where any component that subscribes to a published type will receive the events. There is a failure detection service that is eventually perfect with suspect and resume events. There can be more sophisticated services, like the transaction service mentioned in Section 3.1 and presented in more detail in Section 6.
2. *Feedback loop support.* This layer supports building feedback loops. This is sufficient for cooperative systems. The two main services needed for feedback loops are a pseudoreliable broadcast (for actuating) and a monitoring layer. Pseudoreliable broadcast (called best-effort broadcast in [16]) guarantees that nodes will receive the message if the originating node survives [13]. Monitoring detects both local and global properties. Global properties can be calculated from local properties using a gossip algorithm [22] or using belief propagation [42]. The broadcast and monitoring services are used to implement self management abilities.
3. *Multiple user support.* This layer supports competitive systems (users with conflicting goals). This is a general problem that requires a general solution. If the users are independent, one possible approach is to use collective intelligence techniques (see Section 4.4). These techniques guarantee that when each user maximizes its private utility function, the global utility will also be maximized. This approach does not work for Sybil attacks (where one user appears as multiple users to the system). No general solution to Sybil

attacks is known. A survey of partial solutions is given in [45]. We cite two of these solutions. One possibility is to validate the identities of users using a trusted third party. Another possibility is to use algorithms designed for a Byzantine failure model, which can handle multiple identical users up to some upper bound. Both solutions give significant performance penalties.

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

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

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

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

How do we design a system that contains complex components? If the component is external to the designed system (e.g., human users connecting to a system) then we must design defensively to limit the effect of the component on

the system's behavior. We need to protect the system from the users and the users from each other. For example, the techniques of collective intelligence can be used, as explained in Section 4.4. Getting this right is not just an algorithmic problem; it also requires social engineering such as incentive mechanisms [31].

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

Time scales Feedback loops that work at different time scales can often be considered to be completely independent of each other. That is, each loop is sensitive to a particular frequency range of system behavior and these ranges are often nonoverlapping. Wiener [40] gives an example of a human driver braking an automobile on a surface whose slipperiness is unknown. The human "tests" the surface by small and quick braking attempts, which allows to infer whether the surface is slippery or not. The human then uses this information to modify how to brake the car. This technique uses a loop at a short time scale to gain information about the environment, which is then used to improve the performance at a long time scale.

4.4 Managing multiple users through collective intelligence

Large systems often have multiple users with conflicting goals. One promising technique to handle this situation is called collective intelligence [43, 44]. It can give good results when the users are independent (no Sybil attacks or collusion). The basic question is how to get selfish agents to work together for the common good. Let us define the problem more precisely. We have a system that is used by a set of agents. The system (called a "collective" in this context) has a global utility function that measures its overall performance. The agents are selfish: each has a private utility function that it tries to maximize. The system's designers define the reward (the increment in private utility) given to each of the agent's actions. The agents choose their actions freely within the system. The overall goal is that agents acting to maximize their private utilities should also maximize the global utility. There is no other mechanism to force cooperation. This is in fact how society is organized. For example, employees act to maximize their salaries and work satisfaction and this should benefit the company.

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

fun? We want to avoid a "Tragedy of the Commons" situation where maximizing private utilities causes a minimization of the global utility [18].

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

$$G = \sum_w W(w)$$
$$W(w) = \sum_d \phi_d(a_d)$$

This sums the week utility $W(w)$ over all weeks w. The week utility $W(w)$ is the sum of the day utilities $\phi_d(a_d)$ for each weekday d where the attendance a_d is the total number of people attending the bar that day. The system designer picks the function $\phi_d(y) = \alpha_d y e^{-y/c}$. This function is small when y is too low or too high and has a maximum in between. Now that we know the global utility, we need to determine the agents' reward function. This is what the agent receives from the system for its choice of weekday. We assume that each agent will try to maximize its reward. For example, [43] assumes that each agent uses a learning algorithm where it picks a night randomly according to a Boltzmann distribution following the energies in a 7-vector. When it gets its reward, it updates the 7-vector accordingly. Real agents may use other algorithms; this one was picked to make it possible to simulate the problem.

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

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

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

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

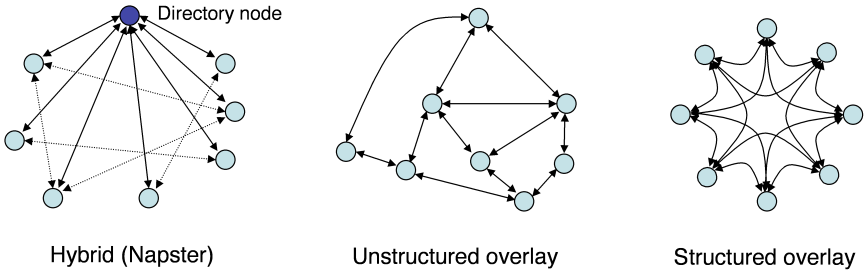

Fig. 6. Three generations of peer-to-peer networks

behavior is that the agents cannot or will not talk to each other, since they do not know each other or are competing. Collective intelligence is one way to get them to cooperate.

5 Structured overlay networks

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

- The first generation is a hybrid: all client nodes are equal but there is a centralized node that holds a directory. This is the structure used by the Napster file-sharing system.
- The second generation is an unstructured overlay network. It is completely decentralized: each node knows a few neighbor nodes. This structure is used by systems such as Gnutella, Kazaa, Morpheus, and Freenet. Lookup is done by flooding: a node asks its neighbor, which asks its neighbors, up to a fixed depth. There are no guarantees that the lookup will be successful (the item may be just beyond the horizon) and flooding is highly wasteful of network resources. Improved versions of this structure use a hierarchy with two kinds of peer nodes: normal nodes and super nodes. Super nodes have higher bandwidth and reliability than normal nodes. This alleviates somewhat the disadvantages.
- The third generation is the structured overlay network. A well-known early example of this generation is Chord [36]. The nodes are organized in a structured way called an exponential network. Lookup can be done in logarithmic time and will guarantee to find the item if it exists. If nodes fail or new nodes join, then the network reorganizes itself to maintain the structure. Since 2001, many variations of structured overlay networks with different advantages and disadvantages have been designed: Chord, Pastry, Tapestry, CAN, P-Grid, Viceroy, DKS, Chord#, Tango, etc. In SELFMAN we build on our previous experience in DKS, Chord#, and Tango.

Structured overlay networks provide two basic services: name-based communication (point-to-point and group) and distributed hash table (also known as DHT, which provides efficient storage and retrieval of (key,value) pairs). Routing is done by a simple greedy algorithm that reduces the distance of a message between the current node and the destination node. Correct routing means that the distance converges to zero in finite time.

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

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

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

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

5.1 Handling network partitioning: the ring merge algorithm

Network partitioning is a real problem for any long-lived application on the Internet. A single router crash can cause part of the network to become isolated from another part. SONs should behave reasonably when a network partition arrives. If no special actions are taken, what actually happens when a partition arrives is that the SON splits into several rings. We need to detect when a split happens and merge the rings back into a single ring when communication is restored [34]. Protocols such as DKS automatically behave as multiple rings when a partition occurs, but they do not automatically merge. We need to extend the protocol with a merge algorithm.

The merge algorithm consists of two parts. The first part detects when the merge is needed. When a node detects that another node has failed, it puts the

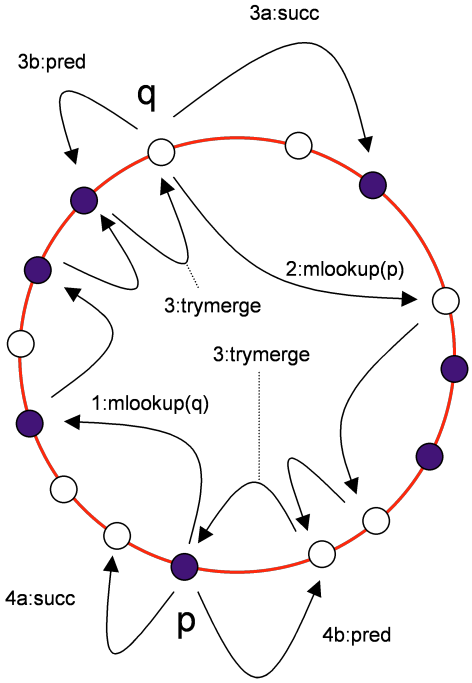

Fig. 7. The ring merge algorithm

node in a local data structure called the passive list. It periodically pings nodes in its passive list to see whether they are in fact alive. If so, it triggers the ring unification algorithm. This algorithm can merge rings in $O(n)$ time for network size n. We also define an improved gossip-based algorithm that can merge the network in $O(\log n)$ average time.

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

5.2 Handling failure suspicions: the relaxed ring algorithm

A typical Internet failure mode is that a node suspects another node of failing. This suspicion may be true or false. In both cases, the ring structure must

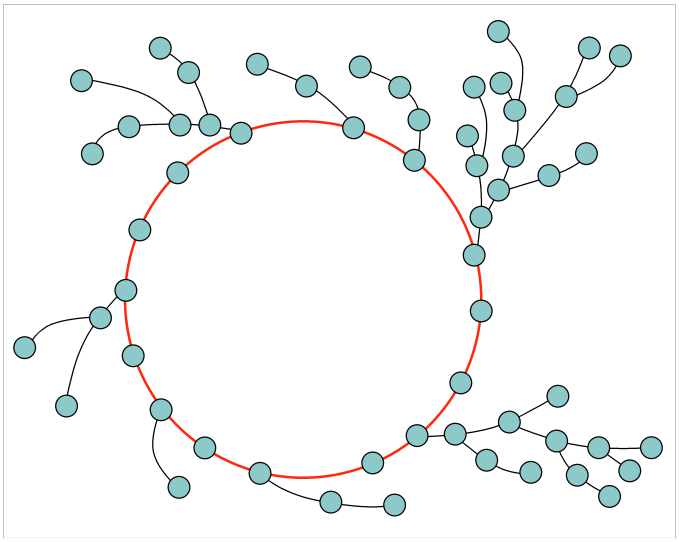

Fig. 8. The relaxed ring structure

be maintained. This can be handled through the relaxed ring algorithm [26]. This algorithm maintains the invariant that every peer is in the same ring as its successor. Furthermore, a peer can never indicate another peer as the responsible node for data storage: a peer knows only its own responsibility. If a successor node is suspected of having failed, then it is ejected from the ring. However, the node may still be alive and point to a successor. This leads to a structure we call the *relaxed ring*, which looks like a ring with "bushes" sticking out (see Figure 8). The bushes appear only if there are failure suspicions. At all times there is a perfectly connected ring at the core of the relaxed ring. The relaxed ring is always converging toward a perfect ring. The number of nodes in the bushes existing at any time depends on the churn (the rate of change of the ring, the number of failures and joins per time).

5.3 Handling multidimensional range queries with load balancing

Efficient data lookup is at the heart of peer-to-peer computing. Many SONs, including DKS and Tango, use consistent hashing to store (key,value) pairs in a distributed hash table (DHT). The hashing distributes the keys uniformly over the key space. Unfortunately, this scheme is unable to handle queries with partial information (such as wildcards and ranges) because adjacent keys are spread over all nodes. In this section, we argue that using DHTs is not a good idea in SONs. We support this argument by showing how to build a practical SON that stores the keys in lexicographic order. We have developed a first protocol, Chord#, and a generalization for multidimensional range queries, SONAR [32].

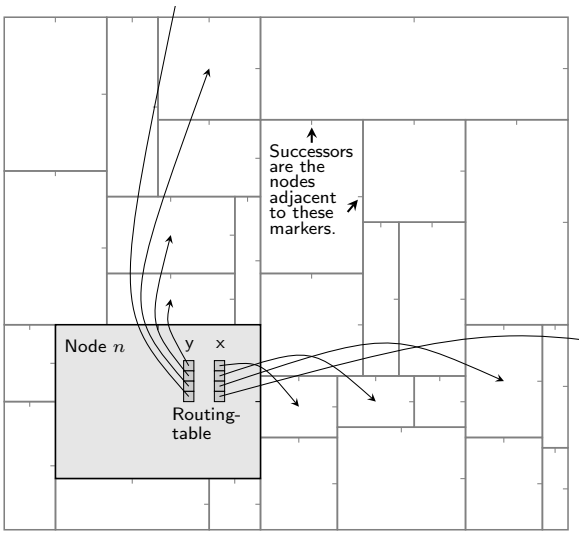

Fig. 9. Two-dimensional routing tables in SONAR

In SONAR the overlay has the shape of a multidimensional torus, where each node is responsible for a contiguous part of the data space. A uniform distribution of keys on the data space is not necessary, because denser areas get assigned more nodes. To support logarithmic routing, SONAR maintains, per dimension, fingers to other nodes that span an exponentially increasing number of nodes. Figure 9 shows an example in two dimensions. Most other overlays maintain such fingers in the key space instead and therefore require a uniform data distribution (e.g., which is obtained using hashing). SONAR, in contrast, avoids hashing and is therefore able to perform range queries of arbitrary shape in a logarithmic number of routing steps, independent of the number of system- and query-dimensions.

6 Transactions over structured overlay networks

For our three decentralized application scenarios, we need a decentralized transactional storage. We need transactions because the applications need concurrent access to shared data. We have therefore designed a transaction algorithm over SONs. We are currently simulating it to validate its assumptions and measure its performance [27, 28]. Implementing transactions over a SON is challenging because of churn (rate of node leaves, joins, and crashes and subsequent reorganizations of the SON) and because of the Internet's failure model (crash stop with imperfect failure detection).

The transaction algorithm is built on top of a reliable storage service. We implement this using replication. There are many approaches to replication on a SON. For example, we could use file-level replication (symmetric replication) or

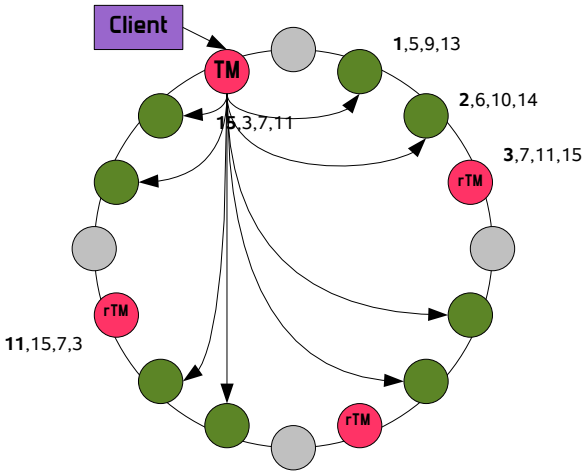

Fig. 10. Transaction with replicated manager and participants

block-level replication using erasure codes. These approaches all have their own application areas. Our algorithm uses symmetric replication [14].

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of the Paxos algorithm [15]. In a companion paper, we have shown that majority techniques work well for DHTs [35]: the probability of data consistency violation is negligible. If a consistency violation does occur, then this is because of a network partition and we can use the network merge algorithm of Section 5.1.

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

7 Conclusions and future work

The SELFMAN project is using self-management techniques to build large-scale distributed systems. This paper gives a snapshot of the SELFMAN project at its halfway point. We explain why self management is important for software design

and we give some first results on how to design self-managing systems as feedback loop structures. We are using structured overlay networks (SONs) as the basis of large-scale distributed self-managing systems. We adapt SONs to a practical setting by extending them to handle network partitioning, failure suspicions, and range queries with load balancing. We are building a transactional storage service running over the SON to support two realistic application scenarios: a distributed Wiki and an on-demand media streaming application. In the rest of the project, we will complete the transactional store and the demonstrator applications and we will evaluate the self-management abilities of our system. The final result will be a set of guidelines on how to build decentralized self-managing applications.

Acknowledgements

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). SELFMAN is a specific targeted research project (STREP) in the Information Society Technologies (IST) Strategic Objective 2.5.5 (Software and Services) of the European Sixth Framework Programme [33]. Peter Van Roy is the coordinator of SELFMAN. He acknowledges all SELFMAN partners for their insights and research results, some of which are summarized in this paper. He also acknowledges Mahmoud Rafea for encouraging him to look at the human endocrine system and Mohamed El-Beltagy for introducing him to collective intelligence.

References

1. Arad, Cosmin, Roberto Roverso, Seif Haridi, Yves Jaradin, Boris Mejias, Peter Van Roy, Thierry Coupaye, B. Dillenseger, A. Diaconescu, A. Harbaoui, N. Jayaprakash, M. Kessis, A. Lefebvre, and M. Leger. *Report on architectural framework specification*, SELFMAN Deliverable D2.2a, June 2007, `www.ist-selfman.org`.
2. Armstrong, Joe. "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, Nov. 2003.
3. Arthur, W. B. *Complexity in economic theory: Inductive reasoning and bounded rationality.* The American Economic Review, 84(2), May 1994, pages 406-411.
4. Ashby, W. Ross. "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): `pcp.vub.ac.be/books/IntroCyb.pdf`.
5. von Bertalanffy, Ludwig. "General System Theory: Foundations, Development, Applications," George Braziller, 1969.
6. Capra, Frank. "It's a Wonderful Life," Liberty Films, 1946.
7. Carroll, Lewis. "Through the Looking-Glass and What Alice Found There," 1872 (Dover Publications reprint 1999).
8. Carton, Bruno, and Valentin Mesaros. *Improving the Scalability of Logarithmic-Degree DHT-Based Peer-to-Peer Networks*, 10th International Euro-Par Conference, Aug. 2004, pages 1060–1067.

9. Collet, Raphaël, Michael Lienhardt, Alan Schmitt, Jean-Bernard Stefani, and Peter Van Roy. *Report on formal operational semantics (components and reflection)*, SELFMAN Deliverable D2.3a, Nov. 2007, `www.ist-selfman.org`.
10. Encyclopaedia Britannica. Article *Human Endocrine System*, 2005.
11. Fairley, Peter. *The Unruly Power Grid*, IEEE Spectrum Online, Oct. 2005.
12. France Télécom, Zuse Institut Berlin, and Peerialism AB. *User requirements*, SELFMAN Deliverable D5.1, Nov. 2007, `www.ist-selfman.org`.
13. Ghodsi, Ali. "Distributed K-ary System: Algorithms for Distributed Hash Tables," Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, Oct. 2006.
14. Ghodsi, Ali, Luc Onana Alima, and Seif Haridi. *Symmetric Replication for Structured Peer-to-Peer Systems*, Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Springer-Verlag LNCS volume 4125, pages 74–85.
15. Gray, Jim and Leslie Lamport. *Consensus on transaction commit.* ACM Trans. Database Syst., ACM Press, 2006(31), pages 133-160.
16. Guerraoui, Rachid, and Luís Rodrigues. "Introduction to Reliable Distributed Programming," Springer-Verlag, Berling, 2006.
17. Halim, Felix, Yongzheng Wu, and Roland Yap. *Security Issues in Small World Network Routing*, Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008), Oct. 2008.
18. Hardin, Garrett. *The Tragedy of the Commons*, Science, Vol. 162, No. 3859, Dec. 13, 1968, pages 1243–1248.
19. Hellerstein, Joseph L., Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. "Feedback Control of Computing Systems," Aug. 2004, Wiley-IEEE Press.
20. Hoglund, Greg and Gary McGraw. "Exploiting Online Games: Cheating Massively Distributed Systems," Addison-Wesley Software Security Series, 2008.
21. IBM. *Autonomic computing: IBM's perspective on the state of information technology*, 2001, `researchweb.watson.ibm.com/autonomic`.
22. Jelasity, Márk, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations*, Springer LNCS volums 3231, 2004, pages 79–98.
23. Kim, Jeong-Rae, Yeoin Yoon, and Kwang-Hyun Cho. *Coupled Feedback Loops Form Dynamic Motifs of Cellular Networks*, Biophysical Journal, 94, Jan. 2008, pages 359-365.
24. Kobayashi, Tetsuya, Luonan Chen, and Kazuyuki Aihara. *Modeling Genetic Switches with Positive Feedback Loops*, J. theor. Biol., 221, 2003, pages 379-399.
25. Lienhard, Michael, Alan Schmitt, and Jean-Bernard Stefani. *Oz/K: A Kernel Language for Component-Based Open Programming*, Sixth International Conference on Generative Programming and Component Engineering (GPCE'07), Oct. 2007.
26. Mejias, Boris, and Peter Van Roy. *A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks*, XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Nov. 2007.
27. Moser, Monika, and Seif Haridi. *Atomic Commitment in Transactional DHTs*, Proc. of the CoreGRID Symposium, Rennes, France, Aug. 2007.
28. Moser, Monika, Seif Haridi, Thorsten Schütt, Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. *First report on formal models for transactions over structured overlay networks*, SELFMAN Deliverable D3.1a, June 2007, `www.ist-selfman.org`.
29. Pigolotti, Simone, Sandeep Krishna, and Mogens H. Jensen. *Oscillation patterns in negative feedback loops*, Proc. National Academy of Sciences, vol. 104, no. 16, April 2007.

30. Plantikow, Stefan, Alexander Reinefeld, and Florian Schintke. *Transactions for Distributed Wikis on Structured Overlays*, 18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007), Springer-Verlag LNCS volume 4785, Oct. 2007, pages 256–267.
31. Salen, Katie, and Eric Zimmerman. "Rules of Play: Game Design Fundamentals," MIT Press, Oct. 2003.
32. Schütt, Thorsten, Florian Schintke, and Alexander Reinefeld. *Range Queries on Structured Overlay Networks*, Computer Communications 31(2008), pages 280-291.
33. SELFMAN: Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components, European Commission 6th Framework Programme, June 2006, `www.ist-selfman.org`.
34. Shafaat, Tallat M., Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*, Journal of Peer-to-Peer Networking and Applications, Springer-Verlag, 2008 (to appear).
35. Shafaat, Tallat M., Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks*, CoreGRID Integration Workshop, Heraklion, Greece, Springer LNCS, 2008.
36. Stoica, Ion, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, SIGCOMM 2001, pages 149-160.
37. Van Roy, Peter. *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS 2006), ENTCS volume 182, June 2007, pages 201-217.
38. Weinberg, Gerald M. "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).
39. Whitehead, Alfred North. Quote: *Civilization advances by extending the number of important operations which we can perform without thinking of them.*
40. Wiener, Norbert. "Cybernetics, or Control and Communication in the Animal and the Machine," MIT Press, Cambridge, MA, 1948.
41. Wiger, Ulf. *Four-Fold Increase in Productivity and Quality – Industrial-Strength Functional Programming in Telecom-Class Products*, Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems, 2001.
42. Wikipedia, the free encyclopedia. Article *Belief Propagation*, March 2008, `en.wikipedia.org/wiki/Belief_propagation`.
43. Wolpert, David H., Kevin R. Wheeler, and Kagan Tumer. *General principles of learning-based multi-agent systems*, Proc. Third Annual Conference on Autonomous Agents (AGENTS '99), May 1999, pages 77-93.
44. Wolpert, David H., Kevin R. Wheeler, and Kagan Tumer. *Collective intelligence for control of distributed dynamical systems*, Europhys. Lett., 2000.
45. Yap, Roland, Felix Halim, and Yongzheng Wu. *First report on security in structured overlay networks*, SELFMAN Deliverable D1.3a, Nov. 2007, `www.ist-selfman.org`.