Chapter 17

D5.7: Guidelines for building self-managing applications

17.1 Executive summary

This report outlines a methodology to build self-managing applications, consisting of the following steps:

- We assume that self-managing applications use an underlying architecture based on loosely coupled components and asynchronous communication. We show three acceptable variations of this architecture, namely the Kompics component model, the Erlang language (used for Scalaris), and the Oz language (used for Beernet).
- The self-managing application then consists of a set of self-managing managers, each of which is assigned a specific management task.
- Each manager is built as a feedback structure. A feedback structure is a set of interacting feedback loops.
- The application design now consists of two steps: decomposition (determining what managers are needed and what they do) and orchestration (handling interactions between the managers).
- We give one complete example of a self-management architecture. To justify each part of this architecture, we show step by step how it is built in the Kompics model.
- Since feedback structures are omnipresent, we give design rules for building them. The correct design of feedback structures is important for building self-managing managers.

- Finally, we focus on the four main self-management axes, namely self tuning, self protecting, self healing, and self configuring, and explain what to do for each case.

This methodology is based on our experience in SELFMAN augmented with important contributions from the GRID4ALL project and from the literature on self-managing systems. Many of our examples are based on decentralized applications that use a structured overlay network for communication and storage, since that is the area where SELFMAN has mainly worked on. We outline what remains to be done to flesh out our methodology.

17.2 Contractors contributing to the Deliverable

This deliverable was written by UCL(P1) (Peter Van Roy) based on SELFMAN results.

17.3 Introduction

How should self-managing systems be designed? For practical system design, it is important to have a methodology that is simple and that allows to design systems with desired global properties. To our knowledge, such a methodology does not yet exist. Most of the knowledge in this area is fragmented and deriving formal properties is difficult. In the SELFMAN project we have addressed a particularly interesting part of the self-management design space: structured overlay networks that survive in realistically harsh environments (with imperfect failure detection and network partitioning) and that provide a transactional storage interface. During the project we have accumulated experience in how to design self-managing systems. This report summarizes this experience. We have built libraries at different levels of abstraction and we have built application demonstrators and self-* services:

- A self-managing structured overlay network for the communications infrastructure. The Scalaris and Beernet libraries both have structured overlay networks.
- A component model for building self-managing applications. The component model supports isolated concurrent components, with hooks for observation and reconfiguration.
- A replicated storage layer and a transaction protocol for managing sharing and coordination for applications. The Scalaris and Beernet libraries both implement transactions on top of the structured overlay network, using a modified version of the Paxos uniform consensus protocol to handle atomic commit in the face of Internet failure model (permanent node failures and temporary communication failures, i.e., false failure suspicions are possible) [61].
- Application demonstrators, such as a Distributed Wiki and a Decentralized Collaborative Drawing Tool.
- Experiments with self-* services, such as load balancing, network partitioning handling, and security (small-world topologies, monitoring, and component security).

From this experience, this report collects a set of guidelines that we provide to future designers of self-managing applications. This report also takes ideas and examples from other sources, including the GRID4ALL project, which has addressed similar problems [6].

17.3.1 Context of this report

This report gives guidelines for the design of a specific kind of self-managing system, namely one that separates a design phase (done by human developers) and a self-managing execution phase (done by the system itself, which is able to adapt itself according to the abilities contained in the design). More general self-managing architectures exist, e.g., the robotic architecture of Gat [26] as applied to self management by Kramer and Magee [47], in which the system design is itself done by part of the system. In Gat's design there are three layers: a component control layer that consists of feedback loops, a change management layer that applies plans to select new control layers, and a top layer that creates new plans using time-consuming deliberation algorithms. In the present report we focus on the first two layers. The first layer corresponds to our feedback loop architectures and the second layer corresponds to our component reconfiguration. The third layer would redesign the components and devise the plan to implement this redesign (i.e., reconfiguration). The third layer is beyond the scope of this report.

17.3.2 General guidelines

A self-managing application consists of a set of interacting feedback loops. Each of these loops continuously observes part of the system, calculates a correction, and then applies the correction. Each feedback loop can be designed and optimized separately using control theory [38] or discrete systems theory [16]. This works well for feedback loops that are independent. If the feedback loops interact, then your design must take these interactions into account. In a well-designed self-managing application, the interactions will be small and can be handled by small changes to each of the participating feedback loop. This is the kind of design that we will focus on in this report.

It can happen that parts of the self-managing application do not fit into this "mostly separable single feedback loops" structure. We have encountered several examples of this in the SELFMAN project. We recommend the following approach:

- In the case of a large number of agents that collaborate, the best approach is to design a distributed algorithm [32] or a multi-agent system [79] to perform the task. For example, in SELFMAN we needed an algorithm to perform atomic commit for distributed transactions, in the face of possible node failures and communication interruptions (imperfect failure detection). We found that a modified version of the Paxos uniform consensus protocol was an essential part of the solution. This is a complex algorithm whose correctness is not trivial to prove [61].

Instead of trying to reinvent it in terms of interacting simple feedback loops, we used the existing knowledge about this algorithm.

- In the case when the feedback loop structure consists of more than one loop intimately tied together, the global behavior must be determined by analyzing the structure as a whole and not by trying to analyze each loop separately. To our knowledge, no general methodology for doing this exists. We have made progress on two fronts: design rules for feedback structures and patterns for common feedback structures. Section 17.6 summarizes some of the important design rules we have encountered. We are preparing a comprehensive survey of feedback loop patterns [15]. Many commonly occurring complex patterns, such as "Tragedy of the Commons" and "Communication Congestion Control", have been extensively studied and it is possible to take an existing pattern from the literature. Unfortunately, the literature is extremely fragmented. Studies of feedback loop systems exist in widely different disciplines, such as management [76], biology [44, 46], and computer science [16, 38]. A future research topic is to devise a methodology at a similar level of abstraction to an existing methodology of software construction (e.g., such as object-oriented programming).

The SELFMAN project has partly studied these two cases (multiple agents and complex feedback). But the domain is very rich and we can guide the designer only along the paths that we have explored ourselves.

17.3.3 Phase transitions

Feedback loop architectures often show abrupt changes in behavior. These shifts can be seen as *phase transitions*. A *phase* in a feedback structure is similar to a phase in thermodynamics: a situation in which the system has well-defined global properties and reacts in a well-defined way to external stimuli. There is a phase transition when the system's global properties and reactions to stimuli change abruptly. If the feedback structure is properly designed, then it reacts to an increasingly hostile environment by doing a *reversible* phase transition [83]. For example, when the node failure rate increases or the network has communication problems (e.g., a partition), then a large overlay network may become a set of disjoint smaller overlay networks. We can say that the overlay network has made a transition from a "liquid" phase (connected with changing set of neighbors) to a "gaseous" phase (disconnected).

This transition does not necessarily mean the end of the overlay network. It can be a normal part of the overlay network's behavior, if the system is

properly designed. When the failure rate decreases, these smaller networks will coalesce into a large network again if the system is properly designed, e.g., with a merge algorithm. The first practical merge algorithm for SONs was developed in SELFMAN [77]. Earlier SONs could not "condense" (move from a gaseous back to a solid phase) as failure rates decreased or communication was reestablished. They would boil (become disconnected) in a hostile environment and then stay disconnected forever. We conclude that network merge is more than just an incremental improvement that helps improve reliability. It is fundamental because it allows the system to sur vive any number of phase transitions. The system is reversible and therefore does not break. Without it, the system breaks after just a single phase transition.

The lesson for system designers is always to make a design that can perform reversible phase transitions. This implies designing algorithms for all pairs of phases for which a transition is possible, in both directions. In addition to this, we recommend to make a design that takes advantage of phase transitions by exposing them to the application as an API. For example, a transactional store built on an overlay network will become a set of smaller stores when there is a network partition. When the overlay network merges, the application needs to merge the data stored in each subnetwork. How to do this is application-dependent; the transaction layer can just provide an API to permit it to be done. The main design issues are to determine what this API should be and how it affects application design. Full answers to these issues are outside the scope of the SELFMAN project. We propose to address them in a follow-up project.

17.3.4 Interdisciplinary nature

It is clear that designing self-managing systems touches on many disciplines. We have encountered the following disciplines:

- Control theory [38]. This allows quantitative design of systems with one or two feedback loops, typically to optimize throughput or some other quantitative property.
- Discrete event systems [16]. This discipline includes control of automata and queueing theory. This generalizes control theory to discrete systems, in which each component is defined by an automaton. Control of these systems can be defined theoretically but is a combinatorially difficult problem in the general case.
- Multi-agent systems [79]. This discipline includes game theory, auction theory, and collective intelligence. This provides practical solutions to

well-defined collaboration problems, called "games" or "auctions".

- Distributed algorithms [32]. This discipline underlies the construction of distributed computing systems. The algorithms depend strongly on the failure model (e.g., whether false suspicions of failure are possible) and the synchronicity model (whether the system is asynchronous or synchronous). This discipline has reached sufficient maturity to provide algorithms for many practical design problems. In SELFMAN, we use the Paxos uniform consensus algorithm as the heart of the transaction commit algorithm.

- Various limited studies on biological or engineering systems that use feedback. There exist many successful biological systems (e.g., the human respiratory and endocrine systems) and engineering systems (e.g., TCP/IP, which can be formulated as a feedback structure) [82, 84]. These systems can be used as examples when designing new systems.

There is no one body of theory on how to build self-managing applications. For parts of the system we recommend to choose the appropriate discipline, as listed above. For the overall system, we recommend to follow general design rules, as presented in the rest of this report, and to take inspiration from existing successful self-managing systems, from biology, sociology, and engineering. For isolated parts of the system and depending on the application requirements, you may need to draw on one or more of these disciplines, as we have done in SELFMAN.

17.3.5 Structure of this report

This report gives an outline of a methodology to build self-managing systems. It is structured as follows:

- Section 17.4 proposes a general architecture for self-managing systems based on loosely coupled components with asynchronous communication. This architecture supports feedback structures, which are the basic design element in a self-managing system: a feedback structure is a set of interacting feedback loops.

- Section 17.5 gives three examples of self-managing systems that were built in SELFMAN using variants of this architecture, namely Kompics, Scalaris, and Beernet. For each of these systems, we explain how it performs self management. In particular, Section 17.5.1 gives a step-by-step outline how to build a self-management architecture with Kompics.

- Section 17.6 gives design rules for building feedback structures. This section just scratches the surface of an area that we intend to investigate further. Much of the literature in this area is fragmented; we are writing a survey to bring this information together [15].
- Section 17.7 then explains how to design the overall structure of a self-managing system, as a set of managers, each of which is implemented as a feedback structure. We design this structure by means of two techniques called decomposition (determining the managers' roles) and orchestration (handling interactions between managers).
- Section 17.8 targets these ideas to the four main self-management axes, namely self tuning, self protecting, self healing, and self configuring. We give specific examples and rules for each axis.
- Section 17.9 concludes by summarizing briefly the methodology presented in this report and by presenting two important unsolved problems for future work.

For all sections we take examples from SELFMAN and from the literature and we reference the appropriate SELFMAN publications.

17.4 The general architecture

We propose to build self-managing systems as loosely coupled sets of concurrent and distributed components, with an asynchronous communication mechanism between components. The default behavior is that components do not communicate. We then gradually add well-defined communications between components, to implement the algorithms in the design. Components with no communications defined between them cannot affect each other.

We have used this approach in designing the three main software artefacts of SELFMAN: the Scalaris library [73], the Kompics component model [21], and the Beernet library [54]. Scalaris is written in Erlang, a language based on asynchronous message passing between isolated processes. Erlang's failure detection model creates asynchronous messages whenever the failure of a process is detected by the system [8]. Kompics is written in Java and provides independent concurrent components that are connected using channels. A channel is a conduit for asynchronous typed events, and each component connected to a channel can subscribe to specific types of events. Components can be disconnected and reconnected, which is needed for reconfiguration.

Beernet is written in Oz and uses active objects with asynchronous message passing, and fault streams for failure detection [18]. Fault streams generalize Erlang's failure detection to handle also temporary failures (also known as false failure suspicions).

17.4.1 Three-layered architecture

The general architecture has three layers:

1. Components and events. This basic layer corresponds to the service architec- ture mentioned above: services based on concurrent components that interact through events. 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 a transaction service.

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 best-effort broadcast (for actuating) and a monitoring layer. Best-effort broadcast guarantees that nodes will receive the message if the originating node survives [32]. Monitoring detects both local and global properties. Global properties are calculated from local properties using a gossip algorithm [41] or using belief propagation [89]. The multicast and monitoring services are used to implement self management abilities.

3. Multiple user support. This layer supports users that compete for resources. This is a general problem that requires a general solution in the area of self protection. If the users are independent, one possible approach is to use collective intelligence techniques. 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 [94]. We cite two known partial solutions. One solution is to validate the identities of users using a trusted third party. Another solution is to use algorithms designed for a Byzantine failure model, which can handle multiple identical users up to some upper bound. Both solutions give significant performance penalties. It seems that the only solutions that make sense in a distributed context are ones that exploit graph properties of social networks. See Section

17.8.2 for more information on how to implement self protection using collective intelligence and the graph properties of social networks.

17.4.2 Combining structured overlay networks and components

The SELFMAN project is based on the observation that there is a synergy between structured overlay networks (SONs) and component models. This synergy has confirmed itself during the project as we have built the transactional store and the application demonstrators. This leads to a set of loosely coupled services built on top of a structured overlay network. Feedback loop structures are built within this framework. We recapitulate the initial reasoning:

- SONs already provide low-level self-management abilities. We are reimplementing our SONs using a component model that adds lifecycle management and hooks for supporting services. This makes the SON into a substrate for building services.

- The component model is based on concurrent components and asynchronous message passing. It uses the communication and storage abilities of the SON to enable it to run in a distributed setting. Because the system may need to update and reorganize itself, the components need introspection and reconfiguration abilities. We have designed a process calculus, Oz/K, that has these abilities in a practical form [23].

This leads to a simple service architecture for decentralized systems: a SON lower layer providing robust communication and routing services, extended with other basic services and a transaction service. Applications are built on top of this service architecture. The transaction service is important because many realistic application scenarios need it (see, e.g., the three SELFMAN demonstrator applications: Distributed Wiki, Sindaca recommendation system, and DeTransDraw collaborative drawing tool).

The structured overlay network is the base. It provides guaranteed connectivity and fast routing in the face of random failures [81]. It does not protect against malicious failures: in our current design we must consider the network nodes as trusted. We assume that untrusted clients may use the over- lay as a basic service, but cannot modify its algorithms. See chapter 16 for more on security for SONs and its effect on SELFMAN. We have designed and implemented robust SONs based on the DKS, Chord#, and P2PS protocols. These implementations use different styles and platforms, for example DKS is implemented in Java and uses locking algorithms for node join

and leave. P2PS is implemented in Oz and uses asynchronous algorithms for managing connectivity (which gives a relaxed ring topology, explained in Appendix A.4). We have also designed an algorithm for handling network partitions and merges, which is an important failure mode for structured overlay networks [77].

The transaction service uses a replicated storage service (Section 6). The transaction service is implemented with a modied version of the Paxos non-blocking atomic commit and uses optimistic concurrency control [30, 61]. This algorithm is based on a majority of correct nodes and eventual leader detection (the so-called partially synchronous model). It should therefore cope with failures as they occur on the Internet.

17.4.3 Failure detection

An important part of a self-management architecture is the approach used for failure detection. We find that using objects with RMI is difficult; it breaks abstraction boundaries and is generally hard to program with [55]. A much better approach is to use an independent failure detection component in the system that informs the application asynchronously. We have explored two variations of this idea:

a In Beernet, we use the Mozart failure detection architecture, which uses fault streams attached to distributed language entities. The fault stream is created by failure detection in the run-time system. The fault stream is read asynchronously by a part of the application independent of the failing part. The fault stream contains events for permanent failures, temporary failures (failure suspicions), and for resuming (lifting a suspicion).

b In Scalaris, we use the Erlang failure detection architecture, which is based on message passing. The run-time system creates messages when it detects failures. The messages are sent to part of the application, which can then take action. The messages detect permanent failures only. Erlang does not support failure suspicions.

This general approach works well in both these cases and we can recommend it for fault tolerance in new self-managing applications. The Erlang system is designed for working in cluster or cloud computing environments, in which there are only permanent failures. The Mozart system is designed for working in an Internet environment, in which there can be frequent failure suspicions (which are often unjustified, but which must be handled anyway).

Critique of RMI for building distributed systems

Our experience shows that RMI (remote method invocation) is the wrong primitive for building distributed systems, for two reasons: exceptions break transparency and synchronous communication adds a useless and slow dependency [55]. The classical view of distributed computing sees partial failure as an error. For instance, a RMI on a failed object triggers an exception. This goes against distribution transparency, because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another issue is both that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives. Synchronous communication also creates a dependency between the sender and the receiver. This dependency is often not needed by the application, but it adds a failure mode to the application that must be handled anyway.

Instead of RMI, we use asynchronous message sending and fault streams. Asynchronous messages introduce no extra dependency. Fault streams indicate errors independently of the rest of the application and hence do not break abstraction boundaries. The Scalaris application is built in Erlang and the Beernet application is built in Oz, both of which support asynchronous messages and asynchronous notification of faults.

More recent trends, such as ambient intelligence and peer-to-peer networks, see partial failure as an inherent characteristic of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. Together with asynchronous messages and fault streams, our experience shows that this approach leads to more realistic language abstractions to build distributed systems.

17.5 Examples of the general architecture

In SELFMAN we have built three significant software designs that use the general architecture of the previous section: the Kompics component model, the Scalaris library, and the Beernet library. Here we summarize these three designs and highlight the lessons they have taught us for building self-managing systems.

17.5.1 A self-management architecture built with the Kompics component model

We show how to build one self-management architecture with the Kompics component model. Kompics was designed to support the operations needed for self-managing systems [21] and the design of this section uses many of its abilities. Here we give a brief overview of Kompics followed by a presentation of the architecture that we have built. The architecture respects the following rules of thumb:

- Fault tolerance requires Isolation requires Concurrency
- Isolation requires Loose Coupling requires Event Passing
- Loose Coupling requires Publish-Subscribe Interaction
- Complexity Management requires Compositionality and Encapsulation
- Resource Reuse requires Sharing
- Self-management requires Online Reconfiguration requires Loose Coupling

We recommend to follow these rules in any self-managing application.

Overview of Kompics

The Kompics model consists of concurrent components that communicate with each other by asynchronously passing typed events on channels (for more information see the Kompics user manual [21]). The current implementation of Kompics is in Java, however, we have made an effort to make it language-independent. Events are different from messages in that they can be received by all the components that are connected to the channel. Components are loosely coupled: they do not know the type, availability, or identity of the components with which they communicate. Components can be nested and shared. Components react to events by atomically executing event handlers, which may trigger new events. Because components are independent, it was possible for us to make Kompics support multi-core scheduling, which allows it to transparently take advantage of multi-core processors.

Building the architecture step by step

Figure 17.1 gives the structure of a realistic self-management architecture that we have built in Kompics, with components for overlay networks and

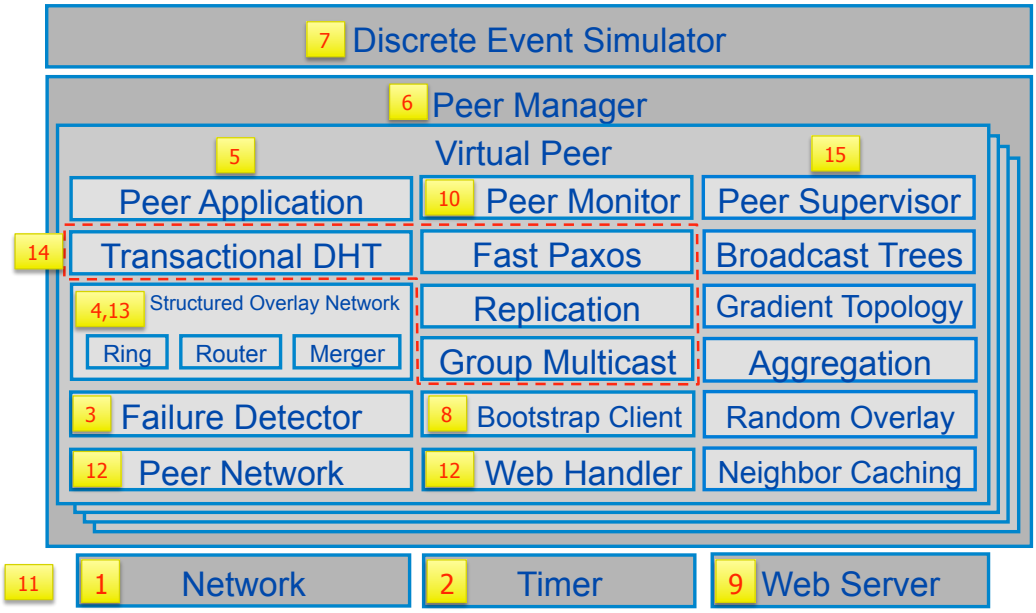

Figure 17.1: A self-management architecture built with Kompics

transactions and with various self-management services. This design seems complex, with a lot of components and nesting structure, but it is actually quite simple and coherent. We reconstruct the system in steps so that it is clear why each component is needed. We build the design with reusable components, instead of keeping it as a monolithic block. Here is a step-by-step explanation of the design (item numbers correspond with the numbers in Figure 17.1):

1. Encapsulate the communication primitives inside a Network abstraction.

2. Encapsulate the timeout and alarm primitives inside a Timer abstraction.

3. Encapsulate the failure detection primitives inside a Failure Detector abstraction.

4. Decompose the SON into its functional aspects, namely the Ring (with periodic stabilization), the Router (with topology maintenance), and the Merger (ring unification). All of these fit into the SON abstraction, which can be used as a building block for higher-level abstractions.

5. Encapsulate everything so far (Peer Application, SON, Failure Detector, Network, and Timer) into a Virtual Peer component. Making the

complete application so far into a component is an important prerequisite for reconfiguration.

6. Allow an enclosing Peer Manager component to add and remove Virtual Peers.

7. The Peer Manager is now generally available as a component in its own right; for example it can be driven by a Discrete Event Simulator component for complex tests.

8. Encapsulate the bootstrapping procedure into a separate component, the Bootstrap Client. This is again important for reconfiguration: the procedure is itself a component that can be replaced.

9. Enable Web-based peer state visualization and debugging with a Web Server component, added at the same level as Network and Timer.

10. Collect global state from local state pushed periodically by a monitoring component, the Peer Monitor, added at same level as Peer Application component (all inside Virtual Peer!).

11. Share the Network, Timer, and Web Server components among all Virtual Peers. Note that the communication part of Kompics allows the Virtual Peers to talk separately to each of these three components.

12. Inside Virtual Peer, keep peer subcomponents unchanged by adding a Peer Network component and a Web Handler component. These are actually proxies. Note that Peer Network can also act as a network simulator.

13. The three SON subcomponents can be replaced, for example the Ring component can be replaced by a Beernet relaxed ring [57] or a Chord# ring [74], the Router component can be replaced by a Chord# router. This can be done without changing anything else: it is a nice form of modularization using the reconfiguration ability.

14. Now add protocol components: Transactional DHT, Fast Paxos, Replication, and Group Multicast. These are dependent: the Transactional DHT uses the other three to implement the transactional store service.

15. We can also add a new pillar inside the Virtual Peer, of components that provide other useful services: Peer Supervisor, Broadcast Trees, Gradient Topology, Aggregation, Random Overlay, and Neighbor Caching. The Peer Supervisor is inspired by Erlang: it supervises peer components for *software* faults [8].

This gives a fairly complex, but flexible and manageable software architecture. Each part has its place and the whole works together well. Many of the subcomponents perform self-management tasks. We propose this architecture as a guideline: it shows one way to bring components together in the context of a structured overlay network. We have done many experiments with Kompics and with this architecture. We are confident that it performs well with no unexpected surprises.

17.5.2 Using self management to provide availability and scalability: the Scalaris example

Scalaris is an example application providing a self-managing data management service for Web 2.0 applications [73]. Web 2.0 initiated a business revolution: service providers offer Internet services for many activities, shopping, online banking, information, social networking, and recreation. In today's society Web 2.0 is no longer a convenience, but customers rely on its continuous availability, regardless of time and space. How to cope with such strong demands, especially in case of interactive community services that cannot be simply replicated? All users access the same Wikipedia, meet in the same Second Life environment and want to discuss with others via Twitter. Even the shortest interruption, caused by system downtime or network partitioning may cause huge losses in reputation and revenue. Web 2.0 services are not just an added value, but they must be dependable. Apart from 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Indeed, scalability is a key challenge. Any scalable service, to be affordable, somehow requires the system to be self managing.

Scalaris provides a comprehensive solution for self-managing scalable data management. Scalaris provides the traditional ACID properties of transactions in a scalable decentralized setting. Scalaris does not attempt to replace current database management systems with their general, fullfledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multiplayer online games. Our system consists of three layers:

1. At the bottom, an enhanced structured overlay network, with logarithmic routing performance, provides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographical ordering instead of random hashing enablescontrol of data

placement which is necessary for low latency access in multidatacenter environments.

2. The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures.

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

As a challenging benchmark for Scalaris, we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". The Wikipedia on Scalaris is fast. Using eight servers it executes 2,500 transactions per second. All operations are performed within transactions to guarantee data consistency and replica synchronization. Adding more computers improves the performance almost linearly. The public Wikipedia, in contrast, employs ten servers to execute the 2,000 requests per second on its large master/slave MySQL database in Tampa.

For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. Scalaris greatly reduces the operation cost with two builtin self-management properties:

- Self healing: Scalaris continuously monitors the hosts it is running on. When it detects a node crash, it immediately repairs the overlay network and the database. Management tasks such as adding or removing hosts require none or minor human intervention.

- Self tuning: Scalaris monitors the nodes's workload and autonomously moves items to distribute the load evenly over the system to improve the response time of the system. When deploying Scalaris over multiple datacenters, these algorithms are used to place frequently accessed items nearby the users.

In traditional database systems these operations require human interference which is error prone and costly. With Scalaris the same number of system administrators can operate much larger installations than with legacy databases.

17.5.3 Using a relaxed ring to simplify overlay maintenance: the Beernet example

The Beernet library is similar in its functionality and underlying implementation platform to Scalaris. Both Beernet and Scalaris use asynchronous message passing between lightweight "objects" to implement a replicated transactional store over a structured overlay network. There are three main differences between Beernet and Scalaris:

- Beernet is written in Oz (using the Mozart Programming System) [62] and Scalaris is written in Erlang [9]. The Oz language provides a network-transparent distribution layer that implements failure detection using fault streams. This can be seen as a generalization of Erlang's failure detection to detect temporary as well as permanent failures [18].

- Beernet uses a relaxed ring overlay network [58] and Scalaris uses a Chord# overlay network [73]. The relaxed ring differs from overlays descended from Chord, such as Chord# and DKS, in that it simplifies ring maintenance. The node join and failure algorithms need the agreement of only two nodes at each step instead of three. The latter requires the atomic update of three peers, which leads to increased perceived node failures when the update fails. In the relaxed ring, there is no need for periodic maintenance of the ring, because the ring remains correct after each step. The relaxed ring algorithms, failure detection and join, are presented as feedback structures in [57].

- Beernet extends the transactional store algorithm to do eager locking and to provide notifications of object updates also to nodes that do not participate in transactions. This extension is needed for the collaborative drawing application [56].

These differences are large enough to see Beernet as a different point in the design space. One of the lessons learned from Beernet is to avoid shared-state concurrency (i.e., threads accessing shared objects through monitors). We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for dataow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives needed, ports and lightweight threads, are also present in Erlang and are not specic to object-oriented programming. Single-assignment variables also appear in other languages such as E and AmbientTalk, in the form of promises.

In Beernet as in Scalaris, we organize the system in terms of active objects only, where an active object consists of a single thread reading a message

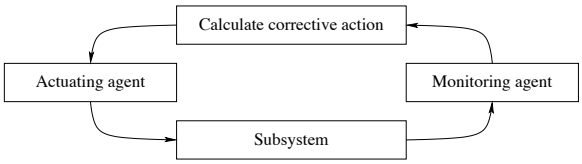

Figure 17.2: A feedback loop

queue and serializing messages to an internal object. We make no distinction in the send operation between a local and a remote active object. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an active object, and only one send operation. The Kompics model is closely related to this: corresponding to active objects and message queues, there are event-driven components and typed event channels.

17.6 Design rules for feedback structures

When building a self-managing system, one of the basic building blocks is the feedback structure: a set of interacting feedback loops. This section gives some basic design rules for feedback structures. We first define what a feedback loop is. A feedback loop consists of three parts that interact with a subsystem (see Figure 17.2): 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. As explained in [82], feedback loops can interact in two ways:

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

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

To find these rules, we start by studying existing feedback loop structures that work well, in both biological and software systems. We have studied different kinds systems: artificial systems (Wiener's hotel lobby example),

computing systems (TCP/IP implementation), and biological systems (human respiratory and endocrine systems). We summarize the design rules that we have found. For more details on the particular examples we refer to the original papers [82, 84, 83].

17.6.1 Stigmergy should be used with care

In Wiener's hotel lobby example, two independent loops attempt to control the temperature of a hotel lobby: a thermostat connected to an airconditioner and a primitive savage lighting a bonfire. This is an example of uncontrolled stigmergy: the two loops will compete and this may lead to a runaway situation such as the hotel being set on fire. The solution in this case is to replace stigmergy by management: the primitive savage should control the temperature by manipulating the thermostat. This is a case where stigmergy is undesirable. In other cases, stigmergy can be used in a positive way. In the example of Section 17.7.4, two agents (a replica manager and a storage manager) communicate through stigmergy. One agent deliberately sets up a situation that will be detected and corrected by the second agent. This is a correct use of stigmergy.

17.6.2 Loop management corresponds to data abstraction

In the human respiratory system, breathing can be consciously controlled. This is modeled by a complex component that manages a respiratory loop. The respiratory loop handles the details of controlling the muscles of the human breathing apparatus and timing the breathing cycles. The complex component does not have to understand these details, but interacts through several parameters in the respiratory loop: the timing of the cycle and the depth of the breathing.

17.6.3 Loop management should control a natural parameter

In the TCP/IP implementation, an inner loop handles the sliding window protocol and an outer loop manages the inner loop to control congestion. The outer loop does this in a simple way: it changes the size of the sliding window. This changes the bandwidth needed by the inner loop, since the size of the sliding window determines the number of packets that can be "in transit" at any moment in time. This is an example of a natural management: the

outer loop adjusts a simple parameter of an inner loop. No other interaction is needed.

17.6.4 Take advantage of different time scales

Different parts of a system, such as two feedback loops, can take advantage of different time scales. For example, one loop can work at short times and another at long times, thus avoiding interference. Or one loop can gather information using short times, and then pass this information to another loop that works at long times. Norbert Wiener [88] gives a simple example of a human driver braking an automobile on a surface whose slipperiness is unknown. The human "tests" the surface by small and quick braking attempts; this allows to infer whether the surface is slippery or not. The human then uses this information to modify how to brake the car. This technique uses a loop at a short time scale to gain information about the environment, which is then used for regulation at a long time scale. The fast loop manages the slow loop.

17.6.5 Complex components should be sandboxed

In the human respiratory system, there is a conscious control of the breathing apparatus. This has the advantage that all the power of conscious reasoning can be brought to bear in the case of catastrophes. For example, if the person is in a car that falls into a river, the conscious control can stop breathing temporarily until the person is outside of the car. Conscious control is also dangerous, however: it can introduce instability. If the person decides to stop breathing (for example, because of a wager), then the system must somehow defend itself. This can be done by having an outer loop observe the conscious control. In the case of the human respiratory system, the brain falls unconscious if the blood oxygen level drops too low. When this happens, the conscious control disappears and the breathing apparatus starts working normally again. The general rule is that complex components can improve the power of the system (for example, they can stabilize an unstable system, like a pilot who stabilizes an unstable airplane) at the price of possibly introducing instability at other occasions. They must therefore always be observed by an outer loop that can take action when this happens.

With respect to stability, there is no essential difference between human components and programmed complex components; both can introduce stability and instability. Human components excel in adaptability (dynamic creation of new feedback loops) and pattern matching (recognizing new situations as variations of old ones). They are poor whenever a large amount of

precise calculation is needed. Programmed components can easily go beyond human intelligence in such areas. Whether or not a component can pass a Turing test is irrelevant for the purposes of self management.

17.6.6 Use push-pull to improve regulation

Many systems are regulated by simple negative feedback. When a parameter becomes too high, there is a reaction to reduce its value. The reaction is a initiated by a second parameter, which is a regulator. The effectiveness of this kind of regulation can be improved by making it "push-pull", that is, by having two regulators, one which actively increases the parameter and one which actively decreases it. In this way, the parameter can be changed quickly in both directions ("pushed" and "pulled").

We give two examples from biology [84]. In the first 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 the blood glucose level causes a decrease in the glucagon concentration and an increase in the insulin concentration, and conversely. These hormones act on the liver, which releases glucose in the blood. The second example is the calcium level in the blood, which is regulated by parathyroid hormone (parathormone) and calcitonine, both of which act on the bone but in opposite directions. The pattern here is of two hormones that act in opposite directions on the regulated substance. This allows improved regulation: quicker changes in the substance's concentration and faster convergence. This pattern is a generally useful one to improve control.

17.6.7 Handle failures with reversible phase transitions

The basic idea is that a system controlled by feedback loops may have several macroscopic (global) states, similar to "phases" in thermodynamics. If the system is exposed to a hostile environment, it may change its global state. In order for the system to survive such changes, they should be reversible. This is explained in detail in Section 17.3.3.

17.7 Overall design of a self-managing system

We now focus on the overall design of a self-managing system: how it is organized as a set of feedback structures. There are two main steps in determining the structure of a self-managing system: *decomposition* and *orchestration*. We first explain what each step is and then we give examples to

show how it is done.

This overall design technique is taken from [6]. In that article, there are two other steps: *assignment* (of tasks to autonomic managers) and *mapping* (of autonomic managers to nodes in the distributed environment). These two steps are not relevant for SELFMAN since the autonomic managers in SELFMAN are not separate entities in the system, but rather feedback loop structures that exist in distributed fashion.

17.7.1 Decomposition: defining the management tasks

The first step is to perform a decomposition, i.e., divide the management into separate tasks. Each task will be performed by a single manager, where a manager corresponds to a feedback loop structure. For example, in the distributed store, we can distinguish connectivity, routing, replicated storage, and transactions. Each of these is done by a different feedback structure. Connectivity is done through ring maintenance. Routing is done through finger table maintenance. And so forth for the other tasks.

To bring clarity into the decomposition, we can divide the tasks into five general areas: functionality and the four non-functional areas of self tuning, self protection, self configuration, and self healing. In the four tasks mentioned before, we distinguish the following non-functional areas:

- Connectivity management is self healing (correctness is the issue) (deliverable D4.2b, see chapter 7).
- Routing is self tuning (performance is the issue) (deliverable D4.3b, see chapter 9).
- Replicated storage is self healing (creating a new replica when one fails) and self configuration (coherent updating of the replicas) (deliverable D4.2b in chapter 7 and deliverable D4.1b in chapter 5).
- Transaction management is self healing (Paxos uniform consensus algorithm for coherent concurrency control [61]).

Each of these four tasks is studied in detail in the corresponding deliverable or paper.

17.7.2 Orchestration: handling the interactions

The second step is to perform the orchestration. This consists of handling the interaction between the management tasks. Each management task is done by a manager, which is a self-contained feedback loop structure that

maintains itself and pursues its own goals. But because all these tasks affect parts of the same system, there can be interactions between them. The possible interactions must be carefully studied and each management task must be written to take them into account.

We find that orchestration is the most challenging part of building a self-managing application. Ideally, the self-management tasks will be designed to have minimal interaction. But some interaction always exists, therefore some coordination is always necessary. For example, the finger table maintenance has to take the connectivity management into account. The replicated storage has to take the routing into account. If a node fails, then the replicated storage has to create a new replica, but taking the routing into account which must also be repaired. We give some rules of thumb to help the designer approach this ideal goal. One rule of thumb is to program each task using a monotonic function, which always increases as the task is performed. In that way, each task can be done with as little interference as possible from other tasks.

17.7.3 Forms of interaction

It is important to understand *how* managers can interact [5]. We identify three possible ways that interaction can happen:

- *Stigmergy*: This is the most ubiquitous and hardest to control. Stigmergy happens because managers make changes to a shared subsystem. Each change made by a manager may be sensed by another manager. Stigmergy is unavoidable. We address it in two ways: by using it to aid coordination when possible and when this is not possible by minimizing its ill effects.

- *Hierarchical management*: This occurs when a manager directly controls another manager. This situation occurs inside a feedback loop structure, when an outer loop controls an inner loop. For example, this situation occurs in the TCP structure or in the human respiratory system. We will not address this further as part of orchestration, but as part of the design of a single feedback loop structure.

- *Direct interaction*: This occurs when two managers interact directly with one another. It does not mean that a manager controls the other, but one manager may request something from another. We also call this peer-to-peer interaction since the managers are peers (not client and server). Direct interaction is sometimes needed since two independent loops managing the same resource may cause undesired behavior. It

must be handled carefully to avoid oscillation or other undesirable behavior. In our case, we handle this by giving each manager a monotonic function with a limiting value that corresponds to perfect behavior. If each manager increases its own function in discrete steps greater than some existing minimal increment then there will be no oscillation.

17.7.4 Examples of interaction

We give some examples of how managers work and interact. These examples are taken from storage management architectures developed in both the SELFMAN project and from other work (in particular, from the GRID4ALL project).

Replica management

This manager is responsible for maintaining the desired replication degree for each stored object in spite of nodes failing and leaving. One possible implementation is as follows. The manager consists of two agents, a replica aggregator and a replica manager. The aggregator subscribes to fail and leave events caused by any object's node. The manager then performs the restoration by creating a new replica.

Storage management

This manager is responsible for maintaining the total storage capacity and total free space of storage, in the presence of dynamism, to meet QoS (tuning) requirements. The dynamism comes from nodes failing or leaving (affecting both total and free storage space) or objects being created or deleted (affecting free storage space). The manager will reconfigure the total free space and/or the total storage space to meet the requirements. The reconfiguration is done by allocating free nodes and deploying additional storage components.

Direct interaction between replica and storage management

There is a race condition between the two above managers. If a node fails, the storage manager may start allocating more nodes and deploying components. Meanwhile the replica manager will be restoring the objects on the failed node. The replica manager might fail to restore the files due to space shortage if the storage manager is slower and does not have time to finish. This may prevent the users, temporarily, from creating objects.

The solution is to let the replica manager wait until the storage manager has completed its work. This can be done through direct interaction between the two managers. This direct interaction does not mean that one manager controls the other. For example, if there is only one replica available for an object, the replica manager may ignore the request to wait from the storage manager and proceed anyway.

Interaction between the replica and storage managers gives an example of stigmergy. When the utilisation of storage on the nodes drops, the storage manager can deallocate some nodes. This deallocation resembles a node failure at the replica manager, which will then restore replicas on other nodes. Because of stigmergy, the replica manager has done the right thing.

Availability of an object can be increased by changing the replication degree. This can be done by an "availability manager" that monitors the frequency of access to the object. If the frequency increases, the availability manager can decide to change the object's replication degree. Before doing this, the replica manager checks with the storage manager to see whether there is enough storage for the new replicas. In this example, the managers collaborate. It is not so that one manager controls another.

17.8 Design rules for the self-management axes

Now that we have explained how to build feedback structures and how to organize them into a self-managing system, we narrow our focus to the four main main self-management axes: self tuning, self protection, self healing, and self configuration. We illustrate the principles in the previous sections with examples taken from the SELFMAN project and from the literature on self management and feedback. We assume that all examples use the right programming model, i.e., concurrent components with asynchronous messages and fault streams instead of distributed exceptions. The examples given have been implemented in different systems, mainly Kompics, Erlang, or Oz, which all provide this basic programming model, with variations.

17.8.1 Making it self-tuning

We outline a load-balancing algorithm for structured overlay networks with good properties and explain how it interacts with replica management. See [39] for a precise definition of the algorithm and an evaluation of its behavior. The load-balancing algorithm is decentralized, i.e., each node acts independently depending on knowledgte at the node. Each node selects a set of nodes with which to balance. A balancing operation consists of an under-

loaded node leaving the network and rejoining at its new location. When the node leaves it first moves its data items to adjacent nodes. When it joins it takes part of the load of adjacent nodes. It takes a maximum of the average global load at its new location. This heuristic reduces the number of data items that are moved unnecessarily. The algorithm uses gossip to maintain an approximation to the average global load.

There is a dependency between the load balancing and the replica management since each node is responsible for a set of items depending on its position in the overlay. If a node decides to balance (leave and rejoin), then the replica manager must recreate the data according to the changes in responsibility. So there is a trade-off between how often to do load-balancing steps (this affects the convergence towards the balanced state) and the cost of replica maintenance. The rate of load balancing needed is mainly dictated by the churn.

To summarize, the replica manager and the load-balancing algorithm are independent of each other, but their parameters must be tuned to reduce impact on the system stability.

17.8.2 Making it self-protecting

Depends on threat model. We propose an approach based on three successive steps, for successively higher levels of security. Each step can work only if the previous step has been successfully implemented. SELFMAN has done experiments in these three steps but we have not built a full security architecture. We can give advice but we do not have user experience.

1. Assume nodes are trusted and network is not. After this step, we can forget about the network and think only about the nodes/users. Node authentication is performed here.

2. Handle non-collusion (nodes do not communicate with each other directly) We distinguish two cases: first where the nodes each use a service independently, and second where the nodes depend on each other. We explain each of the two cases below. The first case is the simplest and can be handled with local techniques. The second case can be handled with collective intelligence techniques.

3. Handle collusion (malicious nodes can communicate with each other directly) This is the most difficult case. In the case of structured overlay networks, changing the topology of the network to become a small-world network can help.

No collusion, with independent nodes

If there is no collusion in a system in which nodes each use a service independently, then end-to-end techniques and trust component techniques suffice. End-to-end security assumes that the nodes are trusted and the network is not. Communication between nodes is encrypted. Then, to assure the trusted nodes are correct, authentication is used to verify the integrity of installed components at the user nodes.

The Scalaris library is designed for working in a data center, which is implicitly considered secure. The Peerialism product is a closed network infrastructure with end-to-end security. So for both of these software products, no security mechanisms are identified. But this does not mean that security is irrelevant. For Scalaris, we have looked at Wikipedia-specific issues, which are relevant to applications using Scalaris. In Wikipedia there can be anonymous edits. Information may not be trustworthy. There may be spam. The question is how to increase the credibility of the content. One technique is to use citations. But this does not directly address credibility, since the cited documents may not be easily accessible or applicable. In the Google Knol application, anonymous edits are not possible. Edits are signed with Google accounts. Edits are signed and therefore put the author's reputation on the line. To enhance Wikipedia credibility (and any application using storage tools such as Scalaris), we propose to use author credentials which are attached to the edits and verified using a secure mechanism.

No collusion, with dependent nodes

It is possible for a system to have no collusion, but the global performance can still depend on the nodes collaborating in some way. A promising technique to achieve this is collective intelligence, which can give good results when the users are independent (no Sybil attacks or collusion). The basic question is how to get selfish agents to work together for the common good. Let us define the problem more precisely. We have a system that is used by a set of agents. The system (called a "collective" in this context) has a global utility function that measures its overall performance. The agents are selfish: each has a private utility function that it tries to maximize. The system's designers define the reward (the increment in its private utility) given to each of the agent's actions. The agents choose their actions freely within the system. The goal is that agents acting to maximize their private utilities should also maximize the global utility. There is no other mechanism to force cooperation. This is in fact how society is organized. For example, employees act to maximize their salaries and work satisfaction and this should benefit

the company.

A well-known example of collective intelligence is the El Farol bar problem [10], 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 do not 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 [37].

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

$$G = \sum_{w} W(w) \tag{17.1}$$

$$W(w) = \sum_{d} \Phi_d(a_d) \tag{17.2}$$

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 $\Phi_d(y) = a_d y e^{-y/c}$. 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, [91] assumes that each agent uses a learning algorithm where it picks a night randomly according to a Boltzmann distribution distributed according to the energies in a 7-vector. When it gets its reward, it updates the 7-vector accordingly. Real agents may use other algorithms; this one was picked to make it possible to simulate the problem.

How do we design the agents 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 ay 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:

$$R_{WL}(w) = W(w) - W_{\text{agent absent}}(w) \tag{17.3}$$

$W_{\text{agent absent}}(w)$ is calculated in the same way as $W(w)$ but assuming the agent is missing (dropped from the attendance vector). We can say that

$R_{WL}(w)$ is the difference that the agent's existence makes, hence the name Wonderful Life taken from the eponymous the Frank Capra movie. 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.

Collusion

The approach of the previous sections does not work when there is collusion, i.e., 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 some cases, the default behavior is that the agents cannot or will not talk to each other, since they do not know each other or are competing. Collective intelligence is one way to get them to cooperate.

Collusion and the topology of the overlay network

One kind of collusion is the Sybil attack, where an attacker can assume identities of many users and gain a large influence on the system. SONs suffer from the Sybil attack. They are vulnerable to attacks that exploit churn and network failures. If Sybil attacks are to be expected, then the topology of the overlay must be changed. We propose to use a small-world network topology (SWN), which has a natural protection against Sybil attacks. It has low maintenance cost and exploits random graph properties that make it hard to disconnect. It is immune to a churn attack. Routing can be done well assuming that node IDs follow a power law distribution.

The SWN routing has a vulnerability in the self-tuning algorithm. A single malicious node can poison the entire network (the poison is propagated through acquaintances) causing the self-tuning algorithm to break down, since node IDs become incorrect. A decentralized self-protection algorithm can help to contain the attack (contain the infection rate to 10%). The idea is that each node resets its ID periodically, which corrects erroneous node IDs.

17.8.3 Making it self-healing

In SELFMAN we have three main examples of self-healing design:

- Replication of the storage over a structured overlay network. This example uses redundancy (each stored item exists multiple times) to allow self healing. Replication is used for keeping the storage intact (one replica always must exist for this to be true) and for the transaction algorithm (the Paxos algorithm requires a majority of replicas to exist to allow a commit).

- Relaxing the structural invariant of a structured overlay network. This is used in the relaxed ring underlying the Beernet library. The relaxed ring handles failure suspicions by using a two-level structure: there is a perfect ring at the center and there may be "bushes" sticking out of it because of failure suspicions. The bushes are either merged back into the perfect ring (when the suspicion goes away) or ejected from the relaxed ring (if the suspicion persists). A node may be incorrectly ejected if there are too many false failure suspicions; in that case the node will simply ask to reconnect with the ring. The connectivity manager is driven by the size of the bushes; as long as there is at least one node in one bush, it will be active.

- Make phase transitions reversible. A phase transition happens in the case of a network partition. The ring may be split into several smaller rings because of network partitions. If node communication in the smaller rings is restored, then the merge algorithm is activated to merge the separate rings into one ring [77].

Each of these three examples uses a different technique to achieve self healing: replication, relaxation of an invariant, and reversibility of a phase transition.

17.8.4 Making it self-configuring

To support self configuration, the system has to be built with components that provide the appropriate reflective primitives. In SELFMAN we have explored these primitives in two models: the Kompics component model and the more extensive WorkflOz/FructOz/LactOz libraries. It is interesting to look at both. Kompics provides primitives for monitoring and reconfiguration; it is sufficient for many basic self-configuration tasks as well as other tasks such as self healing (detection of failures and reconfiguration). The WorkflOz/FructOz/LactOz libraries provide more extensive abilities: they

support more sophisticated self-configuration structures. The basic operations needed for self configuration are the following:

- *Observation and navigation.* The LactOz library allows dynamic navigating and monitoring of the component structure.
- *Reconfiguration.* The FructOz library allows dynamic deploying and configuring of the component structure.
- *Workflow.* The WorkflOz library was designed in Year 3. It allows to create dependencies between different managers through which information can flow. This can simplify the system structure since managers are informed in a more uniform way of important events in the system.

17.9 Conclusions

We have given a first approximation of a methodology to build self-managing systems. We start by proposing a general architecture (Sections 17.4 and 17.5), with examples from Kompics, Scalaris, and Beernet. This architecture supports feedback structures, which are the basic design element in a self-managing system: a feedback structure is a set of interacting feedback loops. Section 17.6 gives design rules for building feedback structures. Section 17.7 then explains how to design the overall structure of a self-managing system, which consists of self-management managers implemented with feedback structures. We determine this structure by means of two techniques called decomposition and orchestration. Finally, Section 17.8 targets these ideas to the four main self-management axes. All these sections take examples from SELFMAN and from the literature and reference the appropriate SELFMAN publications.

17.9.1 Future work

SELFMAN has made some progress in understanding how to build self-managing systems but much remains to do. We have encountered several important unsolved problems that are outside the scope of the project. Specifically, here are two:

- To create a complete methodology for *program design with feedback structures* that is usable by practicing programmers. This means it should be at a similar abstraction level as existing methodologies, e.g.,

for object-oriented programming. Two important parts of this methodology are *feedback patterns* and *design rules*. In SELFMAN we have explored a few patterns and rules in the context of building self-managing applications, but we have only begun to formulate a methodology.

- To study how to build applications that can undergo *reversible phase transitions*. This is essential for building long-lived applications that can survive in realistically hostile situations. Any future research direction in fault tolerance must take phase transitions into account. In SELFMAN we have explored one case of reversible phase transitions, namely the effect of network partitions on structured overlay networks and the use of a merge algorithm to make the partition reversible.

These problems should be addressed in future research.