Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions

Peter Van Roy
Dept. of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

Programs are fragile for many reasons, including software errors, partial failures, and network problems. One way to make software more robust is to design it from the start as a set of interacting feedback loops. Studying and using feedback loops is an old idea that dates back at least to Norbert Wiener's work on Cybernetics. Up to now almost all work in this area has focused on how to optimize single feedback loops. We show that it is important to design software with multiple interacting feedback loops. We present examples taken from both biology and software to substantiate this. We are realizing these ideas in the SELFMAN project: extending structured overlay networks (a generalization of peer-to-peer networks) for large-scale distributed applications. Structured overlay networks are a good example of systems designed with interacting feedback loops. Using ideas from physics, we postulate that these systems can potentially handle extremely hostile environments. If the system is properly designed, it will perform a reversible phase transition when the node failure rate increases beyond a critical point. The structured overlay network will make a transition from a single connected ring to a set of disjoint rings and back again when the failure rate decreases. We are exploring how to expose this phase transition to the application so that it can continue to provide a service. For validation we are building three realistic applications taken from industrial case studies, using a distributed transaction layer built on top of the overlay. Finally, we propose a research agenda to create a practical design methodology for building systems based on the use of interacting feedback loops and reversible phase transitions.

Keywords: self management, feedback, phase transition, fault tolerance, structured overlay network, distributed computing, distributed transaction, network partition, Internet

1. INTRODUCTION

How can we build software systems that are not fragile? For example, we can exploit concurrency to build systems whose parts are mostly independent. Keeping parts as independent as possible is a necessary first step. But concurrency is not sufficient: as systems become larger, their inherent fragility becomes more and more apparent. Software errors and partial failures become common, even frequent occurrences. Both of these problems can be made less severe by rigorous system design (e.g., designing with formal methods and building with redundancy), but for fundamental reasons the problems will always remain. They must be addressed. One way to address them is to build systems as multiple interacting feedback loops. Each feedback loop continuously observes and corrects part of the system. As much as possible of the system should run inside feedback loops, to gain this robustness. This idea was proposed explicitly by Norbert Wiener in 1948 [32].

Building a system with feedback loops puts conditions on how it must be programmed. We find that message passing is a satisfactory model: the system is a set of concurrent component instances that communicate through asynchronous messages. Component instances may have internal

state but there is no global shared state. Failures are detected at the component level. Using this model lets us reason about the feedback behavior. Similar models have been used by E for building secure distributed systems [20] and by Erlang for building reliable telecommunications systems [1]. More reasons for justifying this model are given in [26]. For the rest of this paper, we will use this model.

Now that we can program systems with feedback loops, the next question is *how* should these systems be organized. A first rule is that systems should be organized as multiple interacting feedback loops. We find that this gives the simplest structure and makes it easier to reason about the system (see Sections 2 and 3). Single feedback loops can be analyzed using techniques specific to their operation; for example Hellerstein *et al* [10] gives a thorough course on how to use control theory to design and analyze systems with single feedback loops. The problem with systems consisting of multiple feedback loops is their global behavior: how can we understand it, predict it, and design for a desired behavior? We need to understand the issues before we can do a theoretical analysis or a simulation.

In the SELFMAN project [22], we are tackling the problem by starting from an area where there is already some understanding: structured overlay networks (SONs). These networks are an outgrowth of peer-to-peer systems. They provide two basic operations, communication and storage, in a scalable and guaranteed way over a large set of peer nodes (see Section 4). By giving the network a particular topology and by managing this topology well, the SON shows self-organizing properties: it can survive node failures, node leaves, and node joins while maintaining its specification. By using concepts and techniques taken from theoretical physics, we are able to understand in a deep way how SONs work and we can begin to understand how to design them to build robust software systems. The concepts of feedback loop and phase transition play an important role in this understanding.

This paper is structured as follows:

- Section 2 defines what we mean by a feedback loop, explains how feedback loops can interact, and motivates why feedback loops are essential parts of any system. We briefly present the mean field approximation of physics and show how it uses feedback to explain the stability of ordinary matter.
- Section 3 gives two nontrivial examples of successful systems that consist of multiple interacting feedback loops: the human respiratory system and the Transmission Control Protocol.
- Section 4 summarizes our own work in this area. We are building a self-management architecture based on a structured overlay network. We conjecture that when designed to support reversible phase transitions, a SON can survive in extremely hostile environments. We support this conjecture by analytical work [15], system design [23], and by analogy from physics [16]. We are currently setting up an experimental framework to explore this conjecture. We target three large-scale distributed applications, built using a transactional service on top of a structured overlay network.
- Section 5 outlines a research agenda to create a practical design methodology for building systems according to these ideas. The developer should be able to design systems consisting of multiple interacting feedback loops that exhibit desired global behavior including reversible phase transitions.

Section 6 concludes by recapitulating how feedback loops can overcome software fragility and why all software should be designed with feedback loops. An important lesson is that systems should be constructed so that they can do reversible phase transitions. Most existing fault-tolerant systems are *not* designed with this goal in mind, so they are broken in a fundamental sense. We explain what this means for structured overlay networks and we show how we have fixed them. We then explain what remains to be done: there is a complete research agenda on how to build robust systems based on interacting feedback loops and reversible phase transitions.

2. FEEDBACK LOOPS ARE ESSENTIAL

2.1. Definition and history

In its general form, a feedback loop consists of four parts: an observer, a corrector, an actuator, and a subsystem. These parts are concurrent agents that interact by sending and receiving messages. The corrector contains an abstract model of the subsystem and a goal. The feedback loop runs continuously, observing the subsystem and applying corrections in order to approach the goal. The abstract model should be correct in a formal sense (e.g., according to the definition of abstract interpretation [5]) but there is no need for it to be complete.

An example of a software system that contains a feedback loop is a transaction manager. It manages system resources according to a goal, which can be optimistic or pessimistic concurrency control. The transaction manager contains a model of the system: it knows at all times which parts of the system have exclusive access to which resources. This model is not complete but it is correct.

In systems with more than one feedback loop, the loops can interact through two mechanisms: *stigmergy* (two loops acting on a shared subsystem) and *management* (one loop directly controlling another). Very little work has been done to explore how to design with interacting feedback loops. In realistic systems, however, interacting feedback loops are the norm.

Feedback loops were studied as a part of Norbert Wiener's cybernetics in the 1940's [32] and Ludwig von Bertalanffy's general system theory in the 1960's [3]. W. Ross Ashby's introductory textbook of 1956 is still worth reading today [2], as is Gerald M. Weinberg's textbook of 1975 that explains how to use system theory to improve general thinking processes [30]. System theory studies the concept of a *system*. We define a system recursively as a set of subsystems (component instances) connected together to form a coherent whole. Subsystems may be primitive or built from other subsystems. The main problem is to understand the relationship between the system and its subsystems, in order to predict a system's behavior and to design a system with a desired behavior.

2.2. Feedback loops in the real world

In the real world, feedback structures are ubiquitous. They are part of our primal experience of the world. For example, bending a plastic ruler has one stable state near equilibrium enforced by negative feedback (the ruler resists with a force that increases with the degree of bending) and a clothes pin has one stable and one unstable state (it can be put temporarily in the unstable state by pinching). Both objects are governed by a single feedback loop. A safety pin has two nested loops with an outer loop managing an inner loop. It has two stable states in the inner loop (open and closed), each of which is adaptive like the ruler's. The outer loop (usually a human being) controls the inner loop by choosing the stable state.

In general, anything with continued existence is managed by one or more feedback loops. Lack of feedback means that there is a runaway reaction (an explosion or implosion). This is true at all size and time scales, from the atomic to the astronomic. For example, the binding of atoms in a molecule is governed by a simple negative feedback loop that maintains equilibrium within given perturbation bounds. At the other extreme, a star at the end of its lifetime collapses until it finds a new stable state. If there is no force to counteract the collapse, then the star collapses indefinitely (at least, until it is beyond our current understanding of how the universe works).

2.2.1. The mean field approximation

The stability of ordinary matter is explained by a feedback loop. An acceptable model for ordinary matter is the mean field approximation, which gives good results outside of critical points (see chapter 1 of [16]). To explain this approximation, we start by the simple assumption that a uniform

substance reacts linearly when an external force is applied:

$$\textit{Reaction} = A \times \textit{Force}$$

For example, for a gas we can assume that density n is proportional to pressure p:

$$n = (1/kT) \times p$$

This is the Boyle-Mariotte law for ideal gases, which is valid for small pressures. But this equation gives a bad approximation when the pressure is high. It leads to the conclusion that infinite pressure on a gas will reduce its volume to zero, which is not true.

We can obtain a much better approximation by making the assumption that throughout the substance there exists a force that is a function of the reaction. This force is called the *mean field*. This gives a new equation:

$$\textit{Reaction} = A \times (\textit{Force} + a(\textit{Reaction}))$$

That is, even in the absence of an external force, there is an internal force $a(\textit{Reaction})$ that causes the reaction to maintain itself at a nonzero value. This internal force is the mean field. There is a feedback effect: the mean field itself causes a reaction, which engenders a mean field, and so on. It is this feedback effect that explains, e.g., why a condensed state such as a liquid can exist at low temperatures independent of external pressure. J. Van der Waals applied this reasoning to the ideal gas law, by adding a term:

$$n = 1/kT \times (p + a(n))$$

where n is the density of the gas and p is the pressure. According to this equation, the density n of a fluid can stay at a high value even though the external pressure is low: a condensed state can exist at low temperature independent of the pressure. The internal pressure $a(n)$ replaces the external pressure. Van der Waals chose $a(n) = a \times n^2$ by following the reasoning that internal pressure is proportional to n, the number of molecules per unit of volume, multiplied by the influence of all neighboring molecules on each molecule. This influence is assumed to be proportional to n. This gives a new equation that is a good approximation over a wide range of densities and pressures.

The mean field approach can be applied to many physical systems. The limits of the approach are attained near critical points. This is because the correlation distance between molecules diverges. Near a critical point, there is a phase change of the fluid, e.g., a liquid can boil to become a gas. The global behavior of the fluid changes. The behavior of matter near critical points no longer follows the mean field approximation but can be explained using scale invariance laws [16]. We are using this behavior as a guide for the design of software systems (see Section 4).

2.3. Feedback loops in human society

Most products of human civilization need an implicit management feedback loop, called "maintenance," done by a human. Each human is at the center of a large number of these feedback loops. The human brain has a large capacity for creating these loops; some are called "habits" or "chores." If there are too many feedback loops to manage, then the brain can no longer cope: the human complains that "life is too complicated"! We can say that civilization advances by reducing the number of feedback loops that have to be managed explicitly [31]. We postulate that this is also true of software.

2.4. Feedback loops in software

Software is in the same situation as other products of human civilization. Existing software products are very fragile: they require frequent maintenance by a human. To avoid this, we propose that software must be constructed as multiple interacting feedback loops, as an effective way to reduce its fragility. This is already being done in specific domains. Here are six examples:

- Brooks' subsumption architecture implements intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [4].
- IBM's Autonomic Computing initiative aims to reduce management costs of computing systems by removing humans from low-level management loops [11]. The low-level loop is managed by a high-level loop that contains a human.
- Armstrong *et al* show how to build reliable telecommunications software in Erlang using the principle of *supervisor trees* [1]. Each internal node in a supervisor tree corresponds to a feedback loop that monitors part of the system.
- Hellerstein *et al* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [10]. Hellerstein gives two examples of adaptive systems with interacting feedback loops: gain scheduling (with dynamic selection among multiple controllers) and self-tuning regulation (where controller gain is continuously adjusted).
- Distributed algorithms for fault tolerance handle a special case of feedback where the observer is a failure detector [18, 9]. The implementation of the failure detector itself requires a feedback loop.
- Structured overlay networks (SONs, closely related to distributed hash tables, DHTs) are inspired by peer-to-peer networks [25]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. Our own work is in the area of SONs; we explain it further in Section 4.

3. EXAMPLES OF INTERACTING FEEDBACK LOOPS

We give two examples of nontrivial systems that consist of multiple interacting feedback loops (for more examples see [27, 29]). Our first example is taken from biology: the human respiratory system. Our second example is taken from software design: the TCP protocol family.

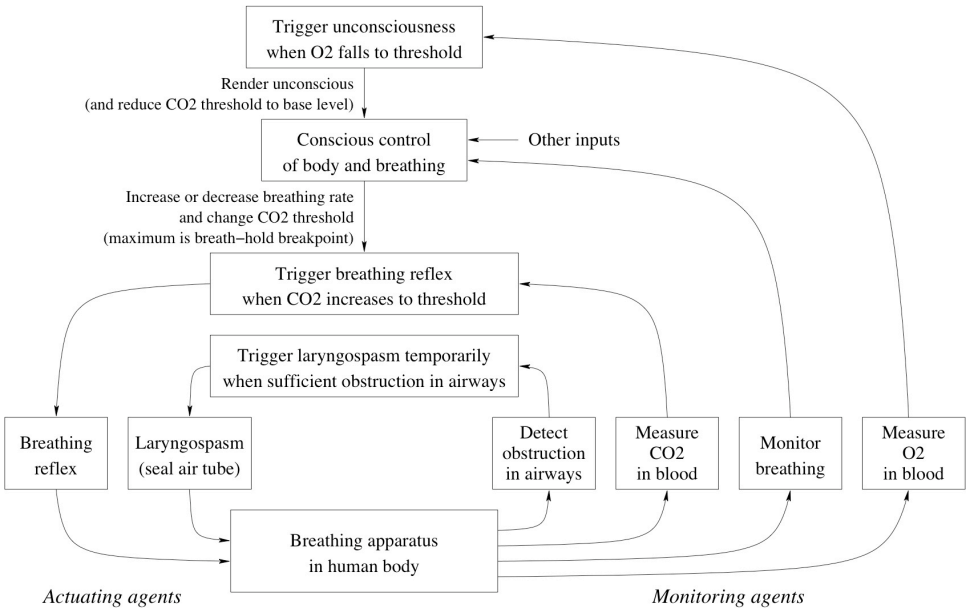

FIGURE 1: The human respiratory system as a feedback loop structure

3.1. The human respiratory system

Successful biological systems survive in natural environments, which can be particularly harsh. Studying them gives us insight in how to design robust software. Figure 1 shows the components of the human respiratory system and how they interact. The rectangles are concurrent component instances and the arrows are message channels. We derived this figure from a precise medical description of the system's behavior [33]. The figure is slightly simplified when compared to reality, but it is complete enough to give many insights. There are four feedback loops: two inner loops

(breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious). From the figure we can deduce what happens in many realistic cases. For example, when choking on a liquid or a piece of food, the larynx constricts so we temporarily cannot breathe (this is called laryngospasm). We can hold our breath consciously: this increases the CO_2 threshold so that the breathing reflex is delayed. If you hold your breath as long as possible, then eventually the breath-hold threshold is reached and the breathing reflex happens anyway. A trained person can hold his or her breath long enough so that the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the breathing reflex is reestablished.

We can infer some plausible design rules from this system. The innermost loops (breathing reflex and laryngospasm) and the outermost loop (falling unconscious) are based on negative feedback using a monotonic parameter. This gives them stability. The middle loop (conscious control) is not stable: it is highly nonmonotonic and may run with both negative or positive feedback. It is by far the most complex of the four loops. We can justify why it is sandwiched in between two simpler loops. On the inner side, conscious control manages the breathing reflex, but it does not have to understand the details of how this reflex is implemented. This is an example of using nesting to implement abstraction. On the outer side, the outermost loop overrides the conscious control (a fail safe) so that it is less likely to bring the body's survival in danger. Conscious control seems to be the body's all-purpose general problem solver: it appears in many of the body's feedback loop structures. This very power means that it needs a check.

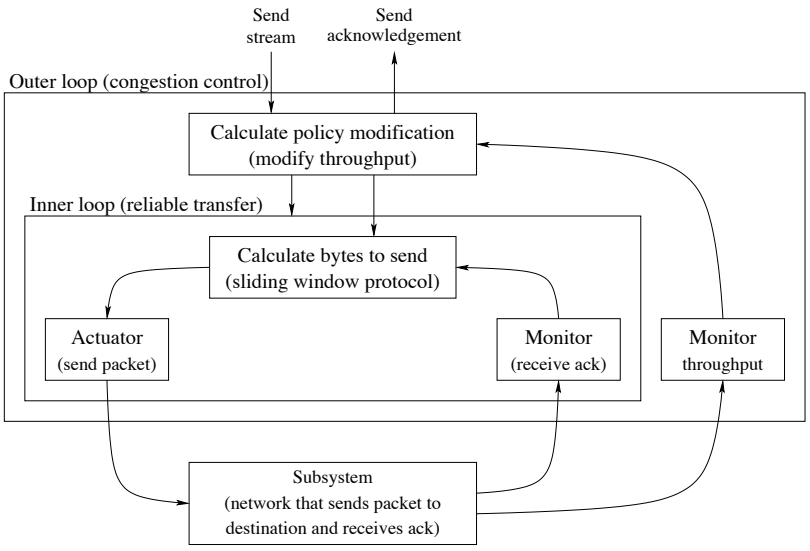

FIGURE 2: TCP as a feedback loop structure

3.2. TCP as a feedback loop structure

The TCP family of network protocols has been carefully tailored over many years to work adequately for the Internet. We consider therefore that its design merits close study. We explain the heart of TCP as two interacting feedback loops that implement a reliable byte stream transfer protocol with congestion control [12]. The protocol sends a byte stream from a source to a destination node. Figure 2 shows the two feedback loops as they appear at the source node. The inner loop does reliable transfer of a stream of packets: it sends packets and monitors the acknowledgements of the packets that have arrived successfully. The inner loop manages a sliding window: the actuator sends packets so that the sliding window can advance. The sliding window can be seen as a case of negative feedback using monotonic control. The outer loop does

congestion control: it monitors the throughput of the system and acts either by changing the policy of the inner loop or by changing the inner loop itself. If the rate of acknowledgements decreases, then it modifies the inner loop by reducing the size of the sliding window. If the rate becomes zero then the outer loop may terminate the inner loop and abort the transfer.

4. STRUCTURED OVERLAY NETWORKS AS A FOUNDATION FOR FEEDBACK ARCHITECTURES

Our own work on feedback structures targets large-scale distributed applications. This work is being done in the SELFMAN project [22]. Summarizing briefly, we are building an infrastructure based on a transaction service running over a structured overlay network [21, 29]. We target our design on three application scenarios taken from industrial case studies: a machine-to-machine messenging application, a distributed knowledge management application (similar to a Wiki), and an on-demand media streaming service [6].

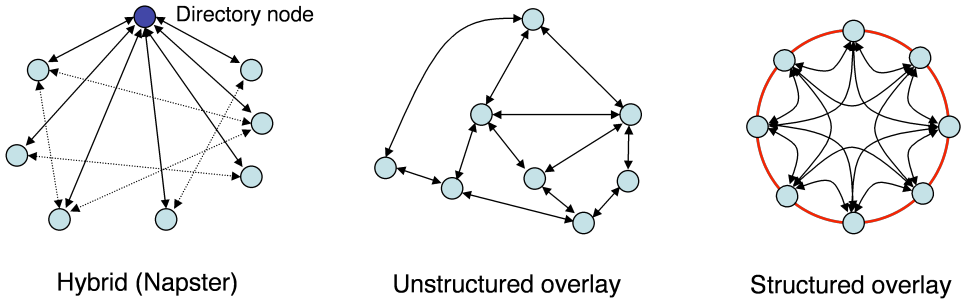

FIGURE 3: Three generations of peer-to-peer networks

4.1. Structured overlay networks

Structured overlay networks are inspired by peer-to-peer networks [25]. In a peer-to-peer network, all nodes play equal roles. There are no specialized client or server nodes. Figure 3 summarizes the history of peer-to-peer networks in three generations. In the first generation (exemplified by Napster), clients are peers but the directory is centralized. In the second generation (exemplified by Gnutella), peer nodes communicate by random neighbor links. The third generation is the structured overlay network. Compared to peer-to-peer systems based on random neighbor graphs, SONs guarantee efficient routing and guarantee lookup of data items. Almost all existing structured overlay networks are organized as two levels, a ring complemented by a set of fingers:

- *Ring structure.* All nodes are connected in a simple ring. The ring is kept connected despite node joins, leaves, and failures.
- *Finger tables.* For efficient routing, extra links called fingers are added to the ring. The fingers can temporarily be in an inconsistent state. This has an effect only on efficiency, not on correctness. Within each node, the finger table is continuously converging to a consistent state.

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

Structured overlay networks are already designed as feedback structures. They already solve the problem of self management for scalable communication and storage. We are using them as the basis for designing a general architecture for self-managing applications. To achieve this goal, we are extending the SONs in three ways:

- We have devised algorithms for handling imperfect failure detection (false suspicions) [19], which vastly reduces the probability of lookup inconsistency. Imperfect failure detection is handled by relaxing the ring invariant to obtain a so-called “relaxed ring,” which maintains

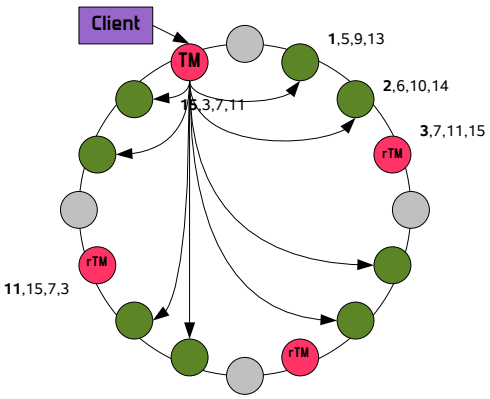

FIGURE 4: Distributed transactions on a structured overlay network

connectivity even with nodes that are suspected (possibly falsely) to be failed. The relaxed ring is always converging to a perfect ring as suspicions are resolved.

- We have devised algorithms for detecting and merging network partitions [23]. This is a crucial operation when the SON crosses a critical point (see Section 4.3).
- We have devised and implemented a transaction algorithm on top of the SON using a symmetric replicated storage [7] and a modified version of the Paxos uniform consensus algorithm to achieve atomic commit with the Internet failure model [21].

4.2. Transactions over a SON

The highest-level service that we are implementing on a SON is a transactional storage. Implementing transactions over a SON is challenging because of churn (the rate of node leaves, joins, and failures and the subsequent reorganizations of the overlay) and because of the Internet's failure model (crash stop with imperfect failure detection). The transaction algorithm is built on top of a reliable storage service. We implement this using symmetric replication [7].

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of Paxos [21, 8]. The Paxos algorithm uses a coordinator node to find a consensus. The coordinator waits for a majority to achieve consensus. If the coordinator node fails, then the algorithm changes coordinators. Since the failure detection is imperfect, the algorithm may change coordinators too often, but this only affects efficiency, not correctness. This failure detection model, in which false suspicions of failure may occur, is called eventually perfect failure detection. It is implementable on the Internet. We have shown that majority techniques work well for DHTs [24]: 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 [23].

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

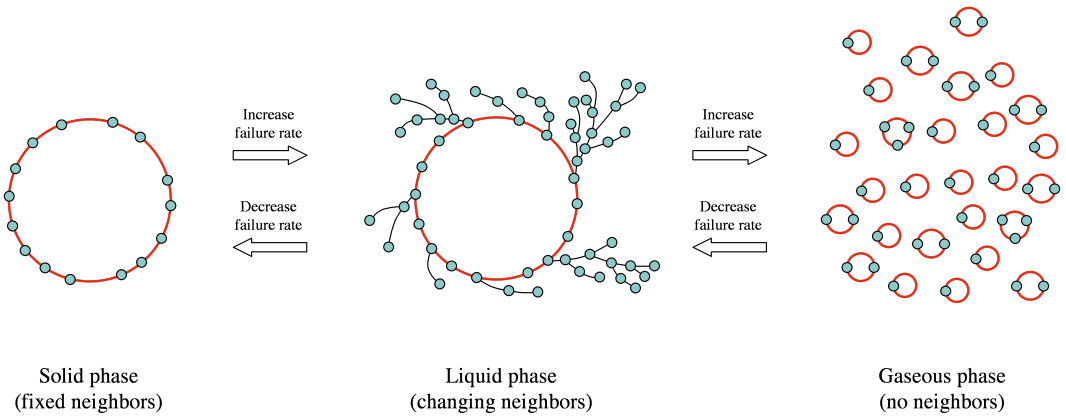

FIGURE 5: Conjectured phase transitions for a relaxed ring SON

4.3. Phase transitions in SONs and their effect on application design

At low node failure rates, a SON is a single ring in which each node has fixed neighbors. This corresponds to a solid phase. At high failure rates, a SON will separate into many small rings. At the limit, a SON with n nodes will separate into n single-node SONs. This is the gaseous phase. In between these two extremes we conjecture that there is a liquid phase, the relaxed ring, where the ring is connected but each node does not have a fixed set of neighbors. When a node is subject to a failure suspicion then its set of neighbors changes [19].

We conjecture that for properly designed SONs phase transitions can occur for changing values of the failure rate. Figure 5 shows the kind of behavior we expect for the relaxed ring. In this figure, we assume that the node failure rate is equal to the node join rate, so that the total number of nodes is stationary. In accord with the Internet's failure model, we also assume that some of the reported failures are not actual failures (they are called failure suspicions [9]). At low failure rates, the ring is connected and does not change (solid phase). At high failure rates, the ring "boils" to become a set of small rings (of size 1, in the extreme case). At intermediate failure rates, the ring may stay connected but because of failure suspicions some nodes get pushed into side branches (relaxed ring).

We support this conjecture by citing [15], which uses the analytical model of [14] to show that phase transitions should occur in the Chord SON [25]. Specifically, [15] shows that three phases are traversed when the average network delay increases, in the following order: a region of efficient lookup, followed by a region where the longest fingers are dead (inefficient lookup), followed by a region where the ring is disconnected. We are setting up simulation experiments to verify this behavior and further explore the phase behavior of SONs.

A SON that behaves in this way will never "fail," it will just change phase. Each phase has a well-defined behavior that can be programmed for. Phase transitions should therefore be considered as normal behavior that can be exposed to the application running on top of the SON. An important research question is to determine what the application API should be for phase transitions. At high failure rates, the application will run as many separate parts. When the rate lowers, these parts will combine (they will "condense" using the merge algorithm) and the application should resolve conflicts between the information stored in the separate rings. We can see that the application will probably have different consistency models at different failure rates.

The transaction algorithm of the previous section behaves correctly in the case of phase transitions, but liveness is reduced in the gaseous phase: all transactions will abort since the majority is never achieved. In some cases a more lively algorithm might be useful, for example, by counting the majority relative to the number of replicas in the current partition. In that way, the application can continue to run when there are network partitions. When the partitions merge, the application has to resolve the conflicts between merged replicas.

As a final remark, we conclude that the merge algorithm is a necessary part of a SON. Without the merge algorithm, condensation of a gaseous system is not possible. The SON is incomplete without it. With the merge algorithm, the SON and its applications can live indefinitely at any failure rate.

5. RESEARCH AGENDA FOR A PRACTICAL DESIGN METHODOLOGY

We have now motivated why it is plausible to design systems using multiple feedback loops that exhibit reversible phase transitions. But we have not given a design methodology: what practical steps the designer should take to build systems according to these ideas. The methodology should be easy to apply and give correct results for the system's global behavior. Formal analysis of systems with multiple interacting feedback loops is difficult [14], e.g., analytical techniques from theoretical physics are necessary to show the existence of phase transitions [15]. Clearly, it is not practical for a system developer to do formal analysis at this level. We propose the following research agenda to create a practical methodology.

- Study existing feedback loop systems (as in Section 3) to build a library of "design patterns." At this stage, these patterns are simply rules of thumb and no formal properties have been derived.
- Translate these patterns into a process calculus, such as one of the numerous variants of the π calculus. The translation should be correct in a formal sense (e.g., according to the definition of abstract interpretation [5]). In our own work, we use the Oz kernel language of [28], which is a process calculus that contains many programming concepts in a factorized manner. We have extended the Oz kernel language with primitives for components and open programming [17].
- Prove the relevant properties of these patterns in the process calculus. Important properties include correctness, stability (convergence, divergence, oscillation, or chaotic behavior), compositionality, and phase behavior. For example, it has been shown empirically that a negative feedback loop can provide stability under certain conditions, oscillatory behavior under other conditions (with time delays), and can be combined with other feedback loops to give desired results (increased reaction times, improved stability, etc.) [13]. We propose to formalize these results and the conditions under which they hold.
- Use the original patterns as design elements. A developer can use the original patterns and immediately derive properties of the resulting system by relying on the proofs done for the formal translations in the process calculus.

If the set of proved patterns is complete enough, in particular if it includes patterns for composition and abstraction, then we have a practical design methodology.

6. CONCLUSIONS

To overcome the fragility of software systems, we propose to build them as a set of interacting feedback loops. Each feedback loop monitors and corrects part of the system. Interaction between feedback loops defines the global behavior of the system. No part of the system should exist outside of a feedback loop. We motivate this approach by analogy from physics and by giving examples of real systems taken from biology and software (the human respiratory system and the Internet's TCP protocol family).

If the feedback structure is properly designed, then it reacts to an increasingly hostile environment by doing a reversible phase transition. For example, when the node failure rate increases, a large overlay network may become a set of disjoint smaller overlay networks. When the failure rate decreases, these smaller networks will coalesce into a large network again. These transitions can be exposed to the application as an API so that it can be written to survive the transition. Important research questions are to determine what this API should be and how it affects application design.

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 Section 5 we propose a research agenda to create a practical methodology that unifies this knowledge and that simplifies deriving properties so that developers can build systems with multiple interacting feedback loops that have desired global behavior including phase transitions.

In our own work in the SELFMAN project [22], we have built structured overlay networks (SONs) that survive in realistically harsh environments (with imperfect failure detection and network partitioning). We have developed a network merge algorithm that allows structured overlay networks to do reversible phase transitions. We are extending our SON with transaction management to implement three application scenarios derived from our industrial partners. We are currently finishing our implementation and evaluating the behavior of our system. Much remains to be done, e.g., we need to extend the transaction algorithm of Section 4.2 according to what each application requires during phase transitions.

One important lesson from this work is that software systems should be designed to support reversible phase transitions. For example, the first practical merge algorithm for SONs is the one reported in [23]. Earlier SONs could not "condense" (move from a gaseous back to a solid phase) as failure rates decreased. They would "boil" (become disconnected) when failure rates increased and they would stay disconnected when the failure rates decreased. We conclude that network merge is more than just an incremental improvement that helps improve reliability. It is *fundamental* because it allows the system to survive any number of phase transitions. The system is reversible and therefore does not break. Without it, the system breaks after just a single phase transition.[1]

7. ACKNOWLEDGEMENTS

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). Peter Van Roy is the coordinator of SELFMAN. He acknowledges all SELFMAN partners for their insights and research results. In particular, he acknowledges the work on the relaxed ring, network partitioning, symmetric replication, distributed transactions, and the analytic study of SONs, all done by SELFMAN partners. Some of this work was done in the earlier PEPITO and EVERGROW projects.

REFERENCES

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

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

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

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

[5] Cousot, Patrick, and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, 4th ACM Symposium on Principles of Programming Languages (POPL 1977), Jan. 1977, pp. 238-252.

[6] France Télécom, Zuse Institut Berlin, and Peerialism AB. *User requirements for self managing applications: three application scenarios*, SELFMAN Deliverable D5.1, Nov. 2007, `www.ist-selfman.org`.

[7] Ghodsi, Ali, Luc Onana Alima, and Seif Haridi. *Symmetric Replication for Structured Peer-to-Peer Systems*, Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Springer-Verlag LNCS volume 4125, pages 74-85.

[8] Gray, Jim, and Leslie Lamport. *Consensus on transaction commit*, ACM Trans. Database Syst., ACM Press, 2006(31), pages 133-160.

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

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

[1] An interesting problem is to explain why aggregates of simple molecules implicitly behave in reversible fashion while software has to be designed for reversibility.

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

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

[13] 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.

[14] Krishnamurthy, Supriya, Sameh El-Ansary, Erik Aurell, and Seif Haridi. *A statistical theory of Chord under churn*, Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, New York, Feb. 2005.

[15] Krishnamurthy, Supriya, and John Ardelius. *An Analytical Framework for the Performance Evaluation of Proximity-Aware Overlay Networks*, Tech. Report TR-2008-01, Swedish Institute of Computer Science, Feb. 2008 (submitted for publication).

[16] Laguës, Michel and Annick Lesne. "Invariances d'échelle: Des changements d'états à la turbulence" ("Scale invariances: from state changes to turbulence"), Belin éditeur, Sept 2003.

[17] Lienhard, Michael, Alan Schmitt, and Jean-Bernard Stefani. *Oz/K: A kernel language for component-based open programming*, Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), Oct. 2007.

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

[19] 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.

[20] Miller, Mark S., Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. *E: Open source distributed capabilities*, 2001, www.erights.org.

[21] Moser, Monika, and Seif Haridi. *Atomic Commitment in Transactional DHTs*, Proc. of the CoreGRID Symposium, Rennes, France, Aug. 2007.

[22] SELFMAN: Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components, European Commission 6th Framework Programme three-year project, June 2006 – May 2009, www.ist-selfman.org.

[23] 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).

[24] Shafaat, Tallat M., Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks*, CoreGRID Integration Workshop, Heraklion, Greece, Springer LNCS, 2008 (to appear).

[25] Stoica, Ion, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, SIGCOMM 2001, pp. 149-160.

[26] Van Roy, Peter. *Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place*, 8th International Symposium on Functional and Logic Programming (FLOPS 2006), April 2006, Springer LNCS volume 3945, pp. 2-12.

[27] Van Roy, Peter. *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS 2006), Springer ENTCS volume 182, June 2007, pages 201-217.

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

[29] Van Roy, Peter, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. *Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project*, Springer LNCS, 2008 (to appear). Revised postproceedings of FMCO 2007, Oct. 2007.

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

[31] Whitehead, Alfred North. Quote: *Civilization advances by extending the number of important operations which we can perform without thinking of them.*

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

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