Designing Robust and Adaptive Distributed Systems with Weakly Interacting Feedback Structures

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

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

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

***Abstract*—Scalable Internet services are distributed over multiple nodes, which may fail at any time. Events such as partial failures, software errors, and attacks, as well as churn are increasingly becoming normal events. As a result, the design of Internet services is getting more complex and predicting their behavior is daunting.**

We address these problems by designing such systems as a set of weakly interacting feedback structures. Each feedback structure consists of a set of interacting feedback loops that together maintain one desired system property. Depending on the system's operating conditions, different parts of the feedback structure will be active, which enables it to adapt to a wide range of operating conditions.

We motivate this approach with examples of robust systems from biology and computing. We have used weakly interacting feedback structures during the design of the Scalaris scalable transaction store. Scalaris consists of five weakly interacting feedback structures working together in a harmonious way without undesired interactions like thrashing or oscillation. Scalaris achieves high performance with strong consistency and horizontal scalability both in tightly coupled settings (clusters and LANs) and loosely coupled settings (PlanetLab).

***Index Terms*—software design, complex systems, distributed systems, feedback loops, self management, transactions, replication, peer-to-peer**

I. Introduction

It is now possible to build Internet applications that are more complex than ever before, because the Internet has reached a higher level of availability and scale. But experience shows that it is difficult to build applications that take advantage of this complexity, because they are hard to design, predict, and manage. They are subject to hostile environmental conditions with frequent node failures and communication problems. They are subject to global problems such as hotspots, attacks, multicast storms, chaotic behavior, and cascading failures [5].

To address these problems, we propose to design applications as a set of *weakly interacting feedback structures (WIFS)*. Each feedback structure consists of a set of feedback loops that together manage one system property. A feedback structure is often organized as a hierarchy where each feedback loop may control an inner loop and may be controlled by an outer loop. Interaction between feedback structures is implicit via monitoring system properties and reacting on system status changes.

A system's specification consists of a conjunction of properties, each implemented by one feedback structure. In a well-designed system, no part exists outside of a feedback structure.

By designing with WIFS, it is practical to build large-scale systems that are robust, adaptable, easy to understand, and easy to maintain. We motivate this claim with examples of real systems taken from biology and computing. We then substantiate the claim with the Scalaris system, a self-managing transactional store based on a structured overlay network.

Overview of the article: This article presents our methodology in a condensed form, illustrated with practical examples. Section II introduces the concept and notation of weakly interacting feedback structures and it lists a few cases where feedback structures are used today to ease the design of complex systems. Section III gives a nontrivial example of a feedback structure from biology, namely the human respiratory system. We explain how it is able to adapt to a wide range of operating conditions. We give its state diagram where each state corresponds to a range of operating conditions handled by the feedback structure. Section IV presents the open-source Scalaris transactional storage library. We contrast two ways of presenting the Scalaris architecture: a traditional presentation as a layered system and a new presentation as a set of five WIFS. Section V explains how to design such systems by giving a set of guidelines for the design of one feedback structure and for the decomposition and orchestration of multiple feedback structures. Finally, Section VI recapitulates the approach and explains why it is important to justify and complete the methodology with formal techniques.

II. Feedback loops and feedback structures

A *feedback loop* in its general form consists of three parts, a monitor, a corrector, and an actuator, attached to a subsystem. We assume without loss of generality that the feedback loop is completely inside the system (this is important because we will later consider systems with many feedback loops) and that the parts are concurrent components (agents) that interact with each other asynchronously.

Figure 1 shows one feedback loop. Each part can perform either a global or local action. For example, a global monitor can use gossip-based aggregation to continuously calculate global information and a global actuator can use a broadcast

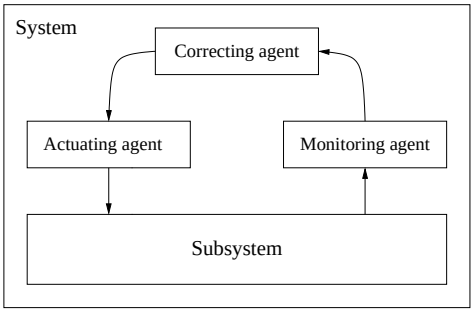

Fig. 1. A feedback loop inside a system

or publish/subscribe mechanism. The corrector contains an abstract model of the subsystem and a goal. The feedback loop runs continuously, monitoring the subsystem and applying corrections in order to approach the goal. The abstract model should be correct in a formal sense (e.g., according to the semantics of abstract interpretation [10]) but there is no need for it to be complete.

Existing software already contains a surprising number of feedback loops in unlikely places. Two examples are garbage collection and virtual memory management. A third example is transaction management: it manages system resources according to a goal, which can be optimistic or pessimistic concurrency control. The monitor accepts lock requests and the actuator gives the response according to the concurrency control algorithm. The transaction manager contains a model of the system: it knows at all times which parts of the system have exclusive access to which resources.

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

It is important to distinguish between the system level (feedback structures) and the building block level (feedback loops). A feedback structure is built using feedback loops as building blocks and maintains a global system property by combining the goal-driven behaviors of its constituent feedback loops.

A. Weakly interacting feedback structures (WIFS)

Very little systematic work exists on how to design with interacting feedback loops. In real systems, however, interacting feedback loops are the norm. But these feedback loops do not interact haphazardly. As far as we can tell from studying working complex systems, they are always organized as *weakly interacting feedback structures (WIFS)*. This is why the study of feedback structures is invaluable for designing and understanding real systems.

Because the interaction is weak, we can design feedback structures separately from their interactions. We design each feedback structure to adapt to different operating conditions, and we design the interactions so that the feedback structures collaborate. The system specification then consists of a conjunction of system properties, each of which is implemented by one feedback structure. We find that dividing system functionality into feedback structures is a natural way to define and to separate concerns in real systems.

B. System design with feedback loops

Using feedback loops for system design is an old idea that dates back at least to Norbert Wiener's work on cybernetics [34]. It is being used successfully in many areas both inside and outside of computing:

- *Artificial intelligence*. For example, Brooks' subsumption architecture implements intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [6].
- *Management of computer systems*. A popular example is IBM's Autonomic Computing initiative, which reduces management costs by removing humans from low-level management loops [17]. It is used primarily for clusters and databases.
- *Telecommunications*. Armstrong *et al.* show how to build reliable telecommunications software in Erlang using the principle of supervisor trees [4]. Each internal node in a supervisor tree corresponds to a feedback loop that monitors part of the system. We used supervisor trees in our Scalaris key/value store (Sec. IV).
- *Control theory*. Hellerstein *et al.* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [14]. Hellerstein gives two examples of adaptive systems with interacting feedback loops: gain scheduling (with dynamic selection among multiple controllers) and self-tuning regulation (where controller gain is continuously adjusted).
- *Distributed algorithms*. These algorithms can be formulated as feedback structures. For example, fault-tolerance algorithms use a feedback loop based on a failure detector [13]. The implementation of the failure detector itself requires a feedback loop.
- *Structured overlay networks*, also called structured peer-to-peer networks. They are inspired by previous generations of peer-to-peer networks with random neighbors but provide guaranteed lookup and performance [25], [30]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. Our work in the SELFMAN project is in this area.
- *Social systems* and *biological systems*. Senge *et al.* show how to debug problems in human organizations by modeling them as feedback structures [27]. Altshuller's theory of inventive problem solving uses feedback and recursion as key parts [3]. Many biological systems use feedback structures and do self organization [11], [8], [21].

We have taken ideas from many of these disciplines to understand how to design with feedback structures [7]. Some disciplines (e.g., control theory) are needed to design a single

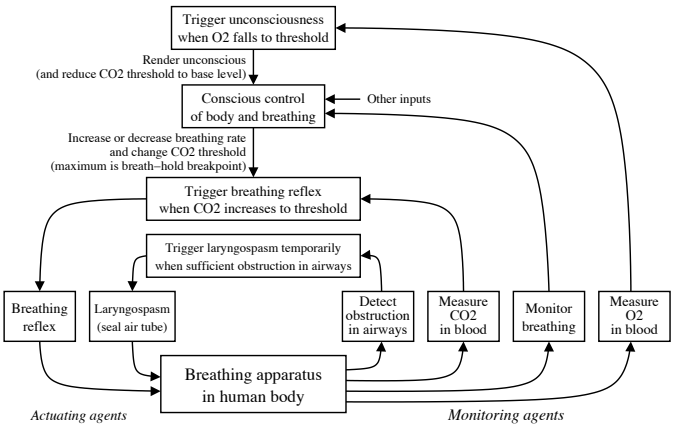

Fig. 2. The human respiratory system as a feedback structure

feedback loop's core algorithm. Others (e.g., system dynamics) are needed to understand design rules and patterns for interacting feedback loops.

III. Examples of feedback structures

To gain insight in the construction and the properties of feedback structures we study working systems. It is important to understand the basic design rules and patterns before doing a formal analysis. We have picked the human respiratory system and the transmission control protocol (TCP) as two examples to explain the issues involved. These examples have quite different origins: the human respiratory system was designed by evolutionary processes over billions of years and the TCP protocol family was designed by human designers over several decades in response to the exponentially growing Internet. Despite this difference in origins, both examples must work well in an environment that can be hostile and they both consist of multiple interacting feedback loops. Other interesting examples are given in [31] (subsumption architecture, fault tolerance in Erlang) and [32] (human endocrine system, Hill equations, collective intelligence).

A. The human respiratory system

Successful biological systems survive in natural environments, which can be particularly harsh. We study them to gain insight in how to design robust software. Figure 2 shows the parts of the human respiratory system and how they interact. We derived this figure from a precise medical description of the system's behavior. The figure is slightly simplified when compared to reality, but it is complete enough to give many insights. There are four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious). Three loops make a hierarchical tower which interacts using stigmergy with the fourth loop. From this figure we can deduce what happens in many realistic cases. For example, holding one's breath increases the CO_2 threshold so that the breathing reflex is delayed. Eventually the breath-hold threshold is reached and the breathing reflex happens anyway. For a trained person the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the breathing reflex is reestablished.

B. Inferring design rules

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 by far the most complex and unpredictable of all. It is not stable: it is highly nonmonotonic and may run with both negative or positive feedback. For example, if a person falls into a lake, conscious control may devise a plan to reach safety where breathing is part of a swimming movement to get to the shore. We can justify why conscious control is sandwiched in between two simpler loops. On the inner side, conscious control manages the breathing reflex, but it does not have to understand the details of how this reflex is implemented. This is an example of using nesting to implement abstraction. On the outer side, the outermost loop overrides the conscious control (a fail safe) so that it is less likely to bring the body's survival in danger. Conscious control seems to be the body's all-purpose general problem solver : it appears in many of the body's feedback structures. This very power means that it needs a check.

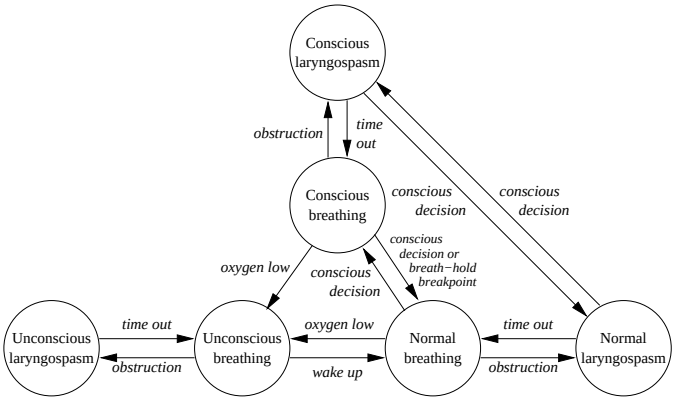

Fig. 3. State diagram of the human respiratory system

C. State diagram and phase behavior

A feedback structure's purpose is to maintain a global system property. A consequence of this purpose is that not all feedback loops need be active at all times. The behavior at any given time is determined by the subset of feedback loops that is necessary to maintain the system property. Which subset depends on the operating conditions of the feedback structure. If the feedback structure is properly designed, this gives it a built-in ability to adapt to widely different operating conditions. For example, a common behavior of the human respiratory system is determined by a single loop, namely the breathing reflex. In this situation the other loops are inactive, in the sense that the behavior would not change if they would be

removed. If the operating conditions change, then a different subset of the loops will become active and the behavior will adapt. This happens if there is an obstruction in the airways, a drop in the blood's oxygen level, or a conscious decision to control breathing.

The set of possible behaviors with the transitions between them can be modeled by a *state diagram*. Figure 3 shows a state diagram that gives the most common states of the human respiratory system and their transitions. Each state corresponds to a specific set of active feedback loops that can handle a well-defined range of operating conditions. Conscious breathing is consciously controlled, whereas normal breathing is automatic (although it may be observed by the conscious control, it is not influenced by it).

The human respiratory system has a single feedback structure and therefore a single global state at any time. In distributed systems (such as the TCP system of the next section or the Scalaris system of Section IV), there can be many copies (instances) of a feedback structure existing on many nodes. These feedback structures can be in different states because the operating conditions are not identical at each node. The global state is then a combination of the local states of each feedback structure. In a structured overlay network like Scalaris, there is redundancy between the nodes so that the overall behavior can be correct even if some of the nodes fail.

However, the overall behavior of a distributed system can be more complicated than just being correct or failing. It is possible for different parts of the system to have different degrees of correctness: some nodes may provide the complete functionality of the system, some nodes may provide partial functionality, and some nodes may provide none at all. This is a completely normal situation for a distributed system. It can be proved that this situation cannot be avoided in general for asynchronous distributed systems (it is a consequence of the CAP theorem [12]).[1] For example, if the network is unreliable, then it is possible for Scalaris to provide full transactional functionality at the majority of nodes, only join and leave functionality at some other nodes, and no functionality at all at yet other nodes.

We can model this situation, in which the system is divided into parts each providing different functionality, by introducing the concept of *phase*. We define a phase as a set of feedback structures whose states provide the same functionality. Different parts of the system can be in different phases. There can be phase transitions, i.e., large parts of the system can change phase, when the operating conditions at the nodes change. These transitions do not require global synchronization, but are a natural consequence of the local change at each node.

D. Transmission Control Protocol (TCP)

The TCP family of network protocols has been carefully tailored over many years to work adequately for the Internet.

[1]The CAP theorem states that for an asynchronous network, it is impossible to guarantee that the following three properties hold simultaneously in all fair executions: (1) consistency (all operations are atomic, i.e., there is a total order between them), (2) availability (every request eventually returns a result), and (3) partition tolerance (any messages may be lost).

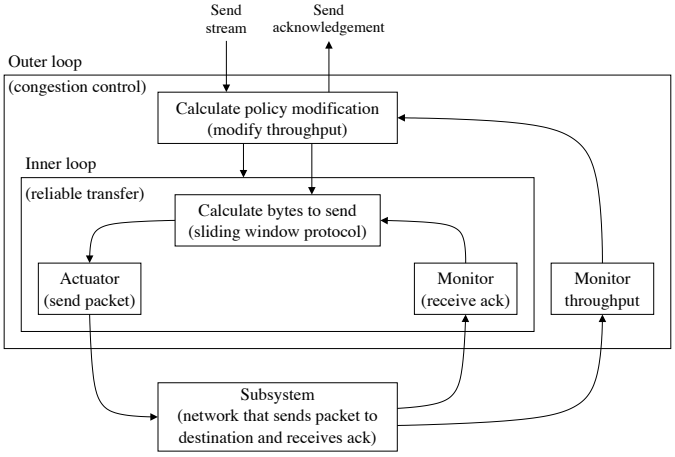

Fig. 4. TCP as a feedback structure

We consider therefore that its design merits close study. We explain the heart of TCP as two feedback loops that interact hierarchically to implement a reliable byte stream transfer protocol with congestion control [15]. The protocol sends a byte stream from a source to a destination node.

Figure 4 shows the two feedback loops as they appear at the source node. The inner loop does reliable transfer of a stream of packets: it sends packets and monitors the acknowledgements of the packets that have arrived successfully. The inner loop implements a sliding window: the actuator sends packets so that the sliding window can advance. The 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.

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

IV. Scalaris key/value store

To show the applicability of our methodology in realistic situations, we have built the Scalaris key/value store [23]. Scalaris is an open-source library providing a self-organizing data management service for Web 2.0 applications [22], [23], [24], [26]. It implements a key/value store with transactions at high performance and strong consistency. Scalaris does not attempt to replace current database management systems with their general, full-fledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed

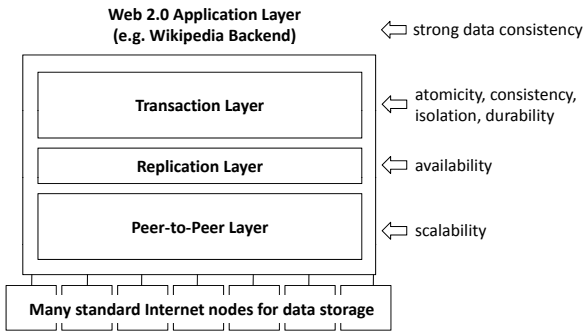

Fig. 5. Scalaris' layered system architecture.

for Internet shopping, banking, or multiplayer online games. We measured transaction performance (read-modify-write) at 4000/second on one node rising to 14000/second on 15 nodes, on a cluster where each node has two dual-core Intel Xeons (4 cores per node) running at 2.66 GHz with 8 GB of main memory, interconnected through GigE [24]. For the IEEE Scalable Computing Challenge 2008, we built a Distributed Wikipedia application on Scalaris and demonstrated it on two platforms: a cluster with 160 cores (tightly coupled) and PlanetLab with 150 peers (loosely coupled) [26].

Figure 5 shows Scalaris in the traditional layered system architecture view:

1) At the bottom, an enhanced structured peer-to-peer 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 [25]. Lexicographical ordering instead of random hashing enables control of data placement which is necessary for low latency access in multi data center environments.
2) The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures. We use symmetric replication, in which the data is replicated symmetrically around the ring.
3) The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses a fast consensus protocol with low communication overhead [24] that has been optimally embedded into the peer-to-peer network.

A bird's-eye view shows that Scalaris is built along these layers: each was designed, implemented and tested separately. But a closer look shows that each layer is made up of weakly interacting feedback structures. The weak interaction through monitoring and actuation allowed us to design the components separately without the need to define explicit interfaces. Thus the WIFS approach does not only link the three layers together, but is also used for orchestration within the layers. It reduced the design complexity and lowered the implementation and testing effort.

A. Feedback structures in Scalaris

As a complement to the layered presentation of the previous section, we present the architecture of Scalaris as a set of five feedback structures and their interactions:

1) *Connectivity management*. This feedback structure maintains the connectivity of the ring topology using periodic successor list stabilization.
2) *Routing management*. This feedback structure maintains efficient routing tables using periodic finger stabilization.
3) *Load balancing*. This feedback structure balances load by monitoring each node and moving nodes when necessary to distribute load evenly.
4) *Replica management*. This feedback structure maintains the invariant that there will always eventually be f replicas of each data item. Whenever there is a potential new replica, it uses consensus to propose a new replica set.
5) *Transaction management*. This feedback structure uses consensus among replicated transaction managers and storage nodes to perform atomic commit. If the transaction manager fails, then one of the replicated transaction managers takes over. Multiple takeovers are tolerated by consensus.

The Scalaris system specification consists of the conjunction of the five properties implemented by these feedback structures together with the functionality of the key-value store:

$$S_{Scalaris} = S_{key/value} \wedge S_{connectivity} \wedge S_{routing} \wedge S_{load} \wedge S_{replica} \wedge S_{transaction}$$

Interactions between the feedback structures are possible when the perceived set of correct nodes changes, due to nodes joining, leaving, failing, or suspected of failing. This gives a dependency graph between feedback structures:

$$S_{connectivity} \rightarrow S_{routing}$$
$$S_{connectivity} \rightarrow S_{replica}$$
$$S_{routing} \rightarrow S_{replica}$$
$$S_{routing} \rightarrow S_{load}$$
$$S_{replica} \rightarrow S_{transaction}$$

General techniques for handling interactions are explained in Section V-B. For Scalaris we handle the interactions as follows:

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

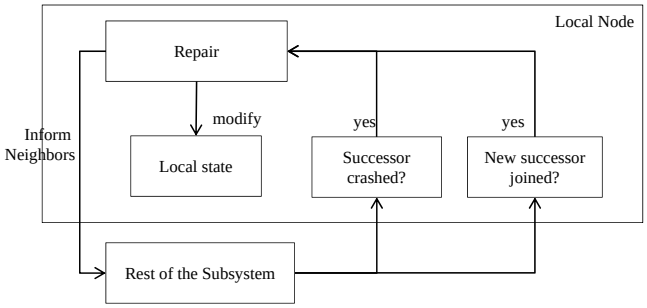

Fig. 6. Scalaris connectivity protocol.

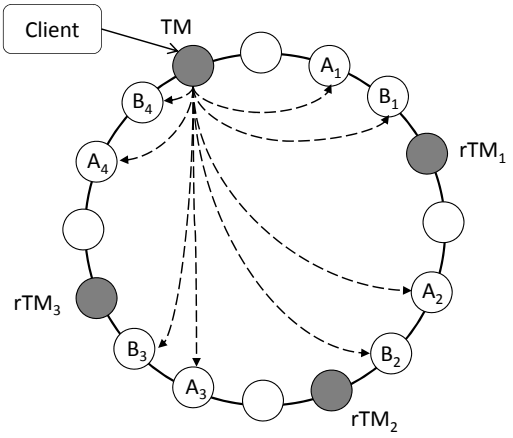

Fig. 7. The Scalaris transaction protocol with replication degree $f = 4$ and two transaction participants A and B

- Replica management can influence the transaction management because the number of replicas can change temporarily. This can cause data inconsistency if there are temporarily more than f replicas, which can occur if there is a false failure suspicion. This is tricky to handle correctly. One solution is to require more than a simple majority in the consensus algorithm of the transaction management. This reduces the probability of inconsistency. But it is not satisfactory because it reduces overall performance just to handle a rare situation. Another solution, which we are working on [24], is to use consensus in the replica management itself to ensure that all nodes agree on the f replicas. The transaction manager then takes a majority only from an agreed set of replicas.
- Covert stigmergy between feedback structures may occur because the network is a shared resource. Connectivity management is the most important property and so it must be done faster than the other managers. Otherwise the overlay network may become disconnected at high loads. To minimize other bad effects due to stigmergy, the management load on the network should be kept as constant as possible. If connectivity management does less work, then routing management takes up the slack.

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

B. Connectivity management

To show how the system design is facilitated using the WIFS approach, we briefly explain the connectivity protocol. Scalaris maintains a unidirectional ring as its basic communication topology. A ring has the property that key ranges can be unambiguously mapped to the nodes in the ring. Each node in the ring maintains a list of successors to its $1^{st}, 2^{nd}, \ldots, k^{th}$ successor, where k is chosen large enough so that the ring is maintained even when some pointers are lost due to churn. The first successor is used for key lookup while the others are needed to recover from node failures.

Figure 6 shows the connectivity management protocol from the perspective of a single node. Each node continuously monitors its successor node. If the successor can no longer be reached (e.g. it does not react to a 'ping'), the node repairs the successor by using the second entry from its successor list. It changes its local state accordingly and may notify other nodes in the system (but this is not strictly necessary).

Each node n also periodically checks whether a new node has joined between itself and its successor. It does so by checking whether $pred(succ(n)) = n$. If this is not the case, a new node was inserted between n and $succ(n)$. The original node then updates its successor list by changing its internal state. Again, it may contact other nodes so that they can update their routing tables, but this is not part of the protocol.

Note that the described interactions are performed independently of all others. The use of WIFS facilitates the design, because the feedback loops can be implemented and tested separately. The protocols for the routing table maintenance, the work-load balancing, and the replica management can be derived analogously.

C. Transaction protocol

The transaction protocol in Scalaris is slightly more complicated, because it involves the determination and stable storage of a consensus among distributed nodes. But again, the use of WIFS greatly facilitates the design process.

Figure 7 shows an example of a transaction on a structured peer-to-peer network that has 16 nodes. A client initiates a transaction by asking its nearest node, which becomes a transaction manager (TM). Other nodes that store data are transaction participants (TPs A_i, B_i). Given symmetric replication with degree f (4 in the figure), we have f transaction managers (TM and rTM in the figure) and f replicas for the other participating nodes. A modified version of Lamport's Paxos [20], [13] uniform consensus algorithm is used for node

agreement [24]: each replicated transaction manager (rTM) collects votes from a majority of participants and locally decides on abort or commit. The transaction manager (TM) then collects a majority from the replicated transaction managers and sends its decision to all participants. This algorithm achieves commitment if more than $f/2$ nodes of each replica group are alive.

The algorithm's operation seems simple; things are actually more subtle because it is correct even if nodes can at any time be falsely suspected of having failed. All we know is that after some unknown finite time, the failure suspicions are correct (eventually perfect failure detection [13]). In our experience, this failure detector is adequate for an Internet setting, where nodes may crash and communication may be interrupted.

We found the WIFS design method especially powerful in the implementation of the described transaction protocol [24], because it helps to tame the high degree of concurrency. Each rTM collects the decisions of all involved TPs in a feedback structure and informs the TM. In a separate feedback structure, the TM awaits the answer of a majority of the TMs and computes a decision. In case the TM crashes during this process, the rTMs vote for a substitute TM – thereby incarnating up to $f-1$ separate feedback structures. Note that all feedback structures may be in different (local) states: the self-organization (orchestration) is done via weak interaction.

V. Guidelines for designing with WIFS

Because interactions between the feedback structures are weak, the design process divides naturally into designing single feedback structures and combining them to form the complete system. We first decompose the system specification into separate properties, each of which is assigned to one feedback structure. We then design each feedback structure separately. Finally, we combine these feedback structures to form the complete system. The complete design may need several iterations because it may be necessary to change or simplify feedback structures to avoid undesirable interactions. Section V-A explains how to design one feedback structure and Section V-B explains how to design systems with several feedback structures. More examples and design rules are given in [33].

A. Design of one feedback structure

A feedback structure consists of a set of feedback loops that work together to maintain a global system property. An important design rule is that each feedback loop should target a separate part of this maintenance. In the human respiratory system, the breathing reflex loop implements normal breathing independently of the laryngospasm loop. In the TCP example, the inner loop implements the sliding window and the outer loop does congestion control by changing a parameter of the inner loop. Each feedback loop can then be designed and optimized separately using control theory [14] or discrete systems theory [9]. This works well when the feedback loops are mostly independent.

It often happens that the feedback loops have a tighter interaction. For example, in the case of resource exhaustion, a common behavior is exponential growth at the beginning when resource usage is small, followed by a slowing of the growth toward a finite limit as resources are exhausted. The S-shaped curve that results is called the *logistic curve*.

The simplest feedback structure that exhibits this behavior consists of two interacting feedback loops: a positive feedback loop that causes the initial exponential growth and a negative feedback loop that causes the slowdown and finite limit. In cases such as this, the global behavior can only be determined by analyzing the loops together. Many such feedback loop patterns have been analyzed in the literature (a survey is given in [7]). Studies of feedback loop patterns have been done independently in widely different disciplines, such as business management [27], biology [8], [21], and computer science [14]. A complete classification of all useful patterns and their behaviors does not yet exist as far as we know.

A feedback loop often needs a nontrivial algorithm to achieve its goal. For example, in the case of a large number of agents that collaborate, to achieve the overall goal the best approach is to use a distributed algorithm [13] or a multi-agent system [29]. For Scalaris 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 [13], [24].

B. Design of the complete system

The complete system is designed in two steps, decomposition and orchestration [2]. Decomposition divides the overall management into separate feedback structures. Orchestration handles the interactions between feedback structures.

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

For a successful orchestration, it is crucial to perform the right decomposition. The managers should be independent or interact only in a simple way. The interactions between feedback structures form a dependency graph, which is a directed graph where each node is a feedback structure and each edge indicates an interaction. Because interactions can be subtle (see Section IV-A), it is important to simplify them as much as possible at design time. Section V-B1 below gives design rules to achieve this for the different kinds of interactions. We enumerate all possible interactions and modify the system so they do not result in undesirable behavior. Section V-B2 then

explains how to build the system in the right order, following the dependency graph.

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

- *Stigmergy*. This occurs when managers make changes to a shared subsystem. Each change made by a manager may be sensed by another manager. This is the most common and is often hard to control. It is a powerful way to communicate for managers that otherwise have no direct communication channel, such as the TCP congestion control loops. Since stigmergic communication tends to be noisy, the managers must be designed to tolerate this.
- *Hierarchy*. This occurs when one manager directly controls another. This situation often occurs inside a single feedback structure, when an outer loop controls an inner loop. For example, it occurs inside the TCP structure and in the human respiratory system. To handle this, we choose the control parameter to be a natural parameter of the system being controlled.
- *Direct interaction*. This occurs when two managers interact as peers, giving a cycle in the dependency graph. It does not mean that one manager controls the other, but one manager may interact with another. Direct interaction is sometimes needed since two independent managers affecting the same resource may cause undesired behavior, such as races or oscillation. It must be handled carefully to avoid replacing one kind of undesired behavior by another. Problems can often be avoided by designing each manager around a monotonic function with a limiting value that corresponds to perfect behavior. Each manager then increases its own function in discrete steps.

2) Build the system in the right order: To reduce the interaction of feedback structures it is important to add them in the right order. If done correctly, each new feedback structure can be added in (almost) orthogonal fashion to the system. The acyclic part of the dependency graph can be ordered according to a topological sort. Cycles are handled separately as explained in the previous section. For Scalaris we propose the following order:

- The first property is self healing (connectivity): the structured peer-to-peer network is based on a ring structure and uses feedback loops to repair the ring if a node joins, leaves, or fails, or to repair the system after a temporary network partition. Scalaris will repair this in some cases by using T-Man together with a "dead node cache" maintained at each node [16]. If a dead node reappears, then T-Man will deliver the node to the ring maintenance which then merges the separate rings using a merge algorithm [28].
- We then add self tuning (routing and load balancing). The first step is to add extra routing links to the nodes (called "fingers" in the literature) to make the routing efficient. This is done through a feedback structure that continuously corrects the fingers depending on the changing structure of the ring. The second step is to update the ring dynamically to remove hotspots. This is done through a feedback structure that periodically collects node load information and performs balancing operations in which an unloaded node leaves the ring and rejoins near a loaded node to take over part of its load.
- We add self configuration. Components use the efficient routing to communicate, in particular to inform nodes when to add or remove new components. This is used for applications built on top of Scalaris, such as the Distributed Wikipedia [26].
- We can also add self protection to the system. This is not implemented in Scalaris. We are currently investigating the best approach to do it. It requires several changes, depending on the threats addressed. For example, to cope with certain kinds of collusion we can extend the ring's topology to approach that of a social network, which is resistant to this kind of attack. We can add an observer of node behavior that can eject bad nodes from the ring. We can use an end-to-end protocol to detect malicious nodes during the routing.

VI. Conclusions and Prospects

To tame the complexity of Internet applications, we propose to build them using weakly interacting feedback structures. Each feedback structure consists of a set of interacting feedback loops that together monitor and correct one global system property. A feedback structure is able to manage a wide variety of operating conditions by activating the appropriate subset of its feedback loops. When correctly designed, feedback structures interact weakly and in a well-defined way. Since a feedback structure only ensures that the system property holds for the subsystem it watches over, it is important that no part of the system exist outside of a feedback structure.

We find that many existing systems can be seen as a collection of WIFS. We give examples from biology and from computing, namely the human respiratory system and TCP. In our own work in the SELFMAN project [23], [32], we have built structured peer-to-peer networks that survive in realistically harsh environments (with imperfect failure detection and network partitioning). The Scalaris architecture consists of five feedback structures whose interactions are carefully controlled.

A. A complete and justified methodology

We have motivated why it is useful to design systems using WIFS and we have presented our own techniques in this area. It is important to justify the methodology formally so that there is no unexpected behavior. Systems can show unexpected behavior that becomes clear only through formal analysis. For example, techniques from theoretical physics have been used to show that structured overlay networks exhibit phase transitions when the network becomes slow [18], [19].

To our knowledge, there exists no formally justified methodology for designing with feedback structures. We consider this justification as one of the most important tasks for software development as the Internet continues to grow in complexity.

We propose a research agenda to create such a methodology. The first step is to study existing feedback loop systems to build a library of patterns and rules. One goal of this research is a complete classification of all possible patterns and their behaviors. The second step is to translate the patterns and rules into a process calculus. The translation should correctly cover the intuitive behavior of the pattern, e.g., it can respect the conditions of abstract interpretation [10]. The third step is to prove the relevant properties of the patterns and rules. Important properties include global correctness, stability, compositionality, and phase behavior. (We note that the phase behavior is predictable and can be exposed to the application as an API.) In the final step, we use the formal treatment to justify the original patterns and rules and to characterize the situations in which they can be used. A software developer can then rely on the proofs without using them explicitly.

VII. Acknowledgments

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). We thank Thorsten Schütt for technical discussions on Scalaris.

References

[1] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. *Distributed Control Loop Patterns for Managing Distributed Applications*. Workshop on Decentralized Self Management for Grids, P2P, and User Communities (part of SASO 2008), Oct. 21, 2008, pp. 260–265.

[2] Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi. *A Design Methodology for Self-Management in Distributed Environments*. Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, Aug. 29-31, 2009, pp. 430–436.

[3] Genrich Altshuller. "The Innovation Algorithm: TRIZ, Systematic Innovation, and Technical Creativity". Translated from the Russian by Lev Shulyak and Steven Rodman. Technical Innovation Center, Inc. 1999.

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

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

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

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

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

[9] Christos G. Cassandras and Stéphane Lafortune. "Introduction to Discrete Event Systems". Second Edition. Springer-Verlag, 2008.

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

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

[12] Seth Gilbert and Nancy Lynch. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. ACM SIGACT News 33(2), June 2002, pp. 51–59.

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

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

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

[16] Márk Jelasity and Özalp Babaoglu. *T-Man: Gossip-based overlay topology management*. Proceedings of 3rd International Workshop on Engineering Self-Organising Systems (ESOA 2005), Springer-Verlag LNCS volume 3910, 2006, pp. 1–15.

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

[18] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. *A statistical theory of Chord under churn*. Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, New York, Feb. 2005, pp. 93–103.

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

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

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

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

[23] Scalaris open-source software library. Zuse Institute Berlin, 2008, `code.google.com/p/scalaris`.

[24] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. "Enhanced Paxos Commit for Transactions on DHTs". 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), May 17-20, 2010, Melbourne, Australia.

[25] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. "Range Queries on Structured Overlay Networks". Computer Communications 31(2), Feb. 2008, Elsevier, pp 280-291.

[26] Thorsten Schütt, Monika Moser, Stefan Plantikow, Florian Schintke, and Alexander Reinefeld. "A Transactional Scalable Distributed Data Store: Wikipedia on a DHT". 1st prize at the IEEE International Scalable Computing Challenge, SCALE 2008, Lyon, May 2008.

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

[28] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*. Journal of Peer-to-Peer Networking and Applications (PPNA) 2(4), 2009, pp. 334–347.

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

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

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

[32] Peter Van Roy, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. *Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project*. Springer-Verlag LNCS volume 5382, 2008, pp. 153–178. Revised postproceedings of FMCO 2007, Amsterdam, The Netherlands, Oct. 2007.

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

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