Overcoming Software Fragility with Interacting Feedback Loops

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

April 10, 2008

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. But almost all work in this area has focused on 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. To make this idea practical, a necessary condition is good support for concurrent programming. We find that a message-passing model without shared state works well. Our own work focuses on 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 from the start as interacting feedback loops. We show how to extend them with a distributed transaction layer that keeps their good self-organization properties. We are using this system to build three realistic application scenarios taken from industrial case studies.

1 Introduction

Concurrency lets us build systems with mostly independent parts. This is necessary for large software systems and distributed systems. But it is not sufficient: as systems become larger, the inherent fragility of their construction becomes more and more apparent. Software errors and partial failures become common, even frequent occurrences. One way to overcome these problems 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.

Building systems as multiple interacting feedback loops puts conditions on how the system supports concurrent programming. 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 and by Erlang for building reliable telecommunications systems. More reasons for justifying this model are given in [18].

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.
- Section 3 gives two nontrivial examples of successful systems that consist of multiple interacting feedback loops: the human respiratory sys-

tem and the Transmission Control Protocol.

- Section 4 summarizes our own work in this area. We target three large-scale distributed applications, built using a transactional service on top of a structured overlay network.

Section 5 concludes by recapitulating how feedback loops can overcome software fragility and why all software should be designed with feedback loops.

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 semantics of abstract interpretation [4]) but there is no need for it to be complete.

Many software systems can be expressed in this way. For example, a transaction manager 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 very common.

Feedback loops were studied as a part of Norbert Wiener's cybernetics in the 1940's [23] and Ludwig von Bertalanffy's general system theory in the 1960's [2]. W. Ross Ashby's introductory textbook of 1956 is still worth reading today [1], as is Gerald M. Weinberg's textbook of 1975 explaining how to use system theory to improve general thinking processes [21]. 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).

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 [22]. We postulate that this is also true of software.

2.3 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; we mention five of them. The subsumption architecture of Brooks is a way to implement intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [3]. Structured overlay networks (closely related to distributed hash tables) are inspired by peer-to-peer networks [17]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. IBM's Autonomic Computing initiative aims to reduce management costs of computing systems by removing humans from low-level management loops [9]. Hellerstein *et al* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [8]. Distributed algorithms for fault tolerance handle a special case of feedback where the observer is a failure detector [11].

3 Examples of interacting feedback loops

We give two examples of nontrivial systems that consist of multiple interacting feedback loops (for more examples see [19, 20]). Our first example is taken from biology: the human respiratory system. Our second example is taken from software design: the TCP protocol family.

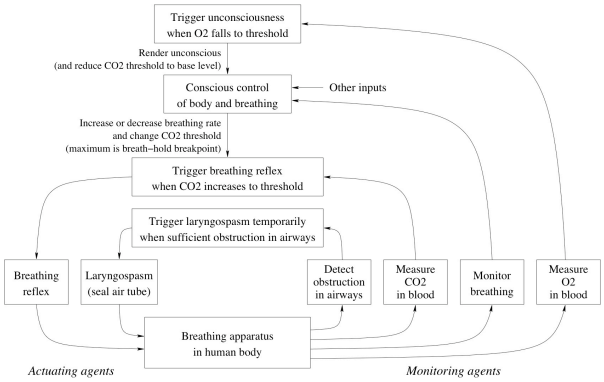

Figure 1: The human respiratory system as a feedback loop structure

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 [24]. 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.

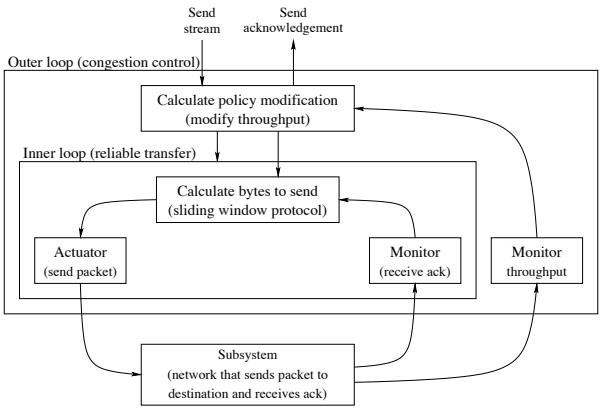

Figure 2: TCP as a feedback loop structure

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 [10]. 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 Distributed transactions on a structured overlay network

Our own work on feedback structures targets large-scale distributed applications [14]. We are building an infrastructure based on a transaction service running over a structured overlay network [13, 20]. 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 [5]. For these scenarios we need distributed transactions.

Structured overlay networks are inspired by peer-to-peer networks [17]. In a peer-to-peer network, all nodes play equal roles. There are no specialized client or server nodes. Structured overlay networks provide two basic services: name-based communication (point-to-point and group) and distributed hash table (also known as DHT) which provides efficient storage and retrieval of (key,value) pairs. Almost all existing structured overlay networks are organized as two levels, a ring complemented by a set of fingers:

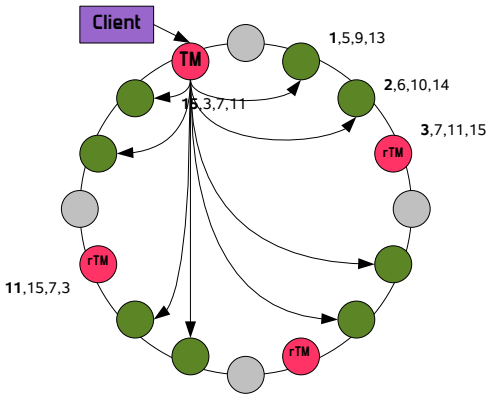

Figure 3: Distributed transactions on a structured overlay network

- *Ring structure.* All nodes are connected in a simple ring. The ring must always be connected despite node joins, leaves, and failures.
- *Finger tables.* For efficient routing, extra 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. We have reimplemented them using a concurrent component model and we are extending them with the hooks and abilities needed for building services and applications. We have devised algorithms for handling imperfect failure detection (false suspicions) [12] and network partitioning (detecting and merging partitions) [15].

Implementing transactions over a structured overlay network 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 [6]. Figure 3 shows an example with a replication factor of four.

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of Paxos [13, 7]. We have shown that majority techniques work well for DHTs [16]: 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 our network merge algorithm.

5 Conclusions

To overcome the fragility of software, one solution is to build it as a set of interacting feedback loops. Each feedback loop monitors and corrects part of the system. No part of the system should exist outside of a feedback loop. We motivate these ideas with real-world designs taken from biology and software. A simple concurrency model that makes these designs easy to implement consists of concurrent components with no shared state, communicating through asynchronous message passing. Using this concurrency model, we have built structured overlay networks that survive in realistically harsh environments (with imperfect failure detection and network partitioning) and we are extending them with transaction management to implement the application scenarios needed by our industrial collaborators.

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 acknowledges all SELFMAN partners for their insights and research results.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

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

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

[12] 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.

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

[14] 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, `www.ist-selfman.org`.

[15] 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).

[16] 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).

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

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