Self Management and the Future of Software Design

Peter Van Roy[1,2]

Dept. of Computing Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Abstract

Most software is fragile: even the slightest error, such as changing a single bit, can make it crash. As software complexity has increased, development techniques have kept pace to manage this fragility. But today there is a new challenge. Complexity is increasing rapidly as a result of two factors: the increasing use of distributed systems as a result of the sufficient reliability and bandwidth of the Internet, and the increasing scale of these systems as a result of the addition of many new computers to the Internet (e.g., mobile phones and other devices). To manage this new complexity, we propose an approach based on *self-managing systems*: systems that can maintain useful functionality despite changes in their environment. The paper motivates this approach and gives some ideas on how to build general self-managing software systems. An important part of the approach is to build systems as hierarchies of interacting feedback loops. We give several examples of these systems and we deduce some rules of thumb for their design.

Keywords: Software development, self management, distributed system, complexity

1 Introduction

Software is fragile and highly nonlinear: even a minor error can have catastrophic effects. Major disasters have occurred due to minor errors such as omitted commas in Fortran programs or changed bits because of alpha rays [10]. So far, this has not unduly hampered the quantity of software being developed. As software complexity has increased, software development techniques have kept pace. This situation is analogous to the Red Queen in Alice [9]: we are running as fast as we can in order to stay in the same place. Software development is now facing a new challenge: complexity is ramping up quickly due to the increased use of distributed systems. The reliability and bandwidth of the Internet infrastructure has reached a point where it is feasible to build large distributed applications. Examples of such applications include the wide variety of file sharing programs (Napster, Gnutella, Morpheus,

[1] This work is funded by the European Union in the SELFMAN project (contract 34084), EVERGROW project (contract 001935) and CoreGRID network of excellence (contract 004265). We thank Luis Quesada, Boriss Mejias, Raphaël Collet, and Yves Jaradin for comments on drafts of this paper.

[2] Email: pvr@info.ucl.ac.be

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Freenet, Bit Torrent, etc.), Skype, various role-playing games (World of Warcraft, etc.), and research testbeds such as PlanetLab [11]. Technologies for building such applications now exist, e.g., Web services and Grid software. Distributed systems are also greatly increasing in scale because of the increased use of large numbers of small devices. E.g., mobile phones are now full-fledged computing nodes with Internet connectivity.

How can we address the problem of programming large-scale distributed systems? Such systems have new properties that greatly increase the complexity of programming: scale (large numbers of independent nodes), partial failure (part of the system fails), security (multiple security domains), resource management (resources are localized), performance (harnessing multiple nodes or spreading load), and global behavior (emergent behavior of the system as a whole). Each of these properties has been studied in isolation. For example, the area of distributed algorithms has solutions for handling partial failure in many cases. But the properties have not been looked at together. The purpose of this paper is to give some ideas how this can be done.

Global behavior is particularly relevant for large systems. They must be designed carefully, otherwise the system will not behave well when stressed. Ideally, it should converge rapidly to its desired behavior. But it may instead collapse, oscillate, or show chaotic behavior. Such erratic behavior has been observed for power grids and has resulted in large-scale power outages. One reason for this is because the power grid's behavior was designed for a situation close to equilibrium; it was not studied far from equilibrium.

2 Self-managing systems

To build large-scale distributed systems with good behavior, we need a framework in which to think about them. What should such a framework look like? To reduce the complexity of the system, it should be able to manage its own problems as much as possible. This leads us to propose self-managing systems as a suitable framework. A self-managing system is one that can maintain its functionality despite changes in its environment, in a general sense.

Self-managing systems have recently been brought to the forefront because of IBM's Autonomic Computing initiative [13]. When computer systems become large then the cost of managing them becomes prohibitive. The initiative aims to reduce this cost by removing humans from the management loop. The role of humans is then to manage the policy and not to maintain the mechanisms. This greatly reduces the need for manual intervention.

Another area that is building self-managing systems is structured overlay networks [1]. These have developed out of the popular applications for peer-to-peer networks. Note that many of the applications mentioned in the introduction are based on peer-to-peer networks. Unlike peer-to-peer networks based on random neighbor communication, structured overlay networks provide both guarantees (information is guaranteed to be found if it exists) and efficiency (broadcast does not flood the network as it does in, e.g., random neighbor networks such as the one used in Gnutella). Structured overlay networks provide primitive self-managing be-

havior: they reorganize themselves in reaction to environmental changes such as failures and overloads, so that their functionality is maintained. Structured overlay networks have led to robust software that is being used in various areas, such as the construction of robust distributed communication networks and robust storage services that continue to provide service despite high node turnover (node "churn").

These two research areas, autonomic systems and structured overlay networks, have attracted attention once again to self-managing systems. But self-managing systems are actually a very old idea. The beginning of the area as a discipline can be dated to the definition by Norbert Wiener of cybernetics in the 1940's [19] and by Ludwig von Bertalanffy of general system theory in the 1960's [4]. The general idea of system theory is to study the concept of a *system*, its properties and design. There are various ways to define the concept of a system [17]. A system can be considered as a set of components (called subsystems) connected together to form a coherent whole. For the purposes of this paper, we consider the system and its subsystems to be concurrent software agents.

System theory is still very much in its early stages. Recent research results have not been systematized into a textbook and the ideas have not been applied to computer science. W. Ross Ashby wrote an introductory textbook in 1956 that is still worth reading today [3]. In the area of computer systems, textbooks exist only for specialized subfields such as distributed algorithms [15]. We consider that it is high time to apply system theory to software construction. In this paper, we motivate this thesis with several examples of realistic systems.

3 Designing self-managing systems

How does one design a self-managing software system? We do not yet have a general set of design techniques, but we can talk about several important aspects: feedback loops, global properties, and the general architectural framework. It turns out that designing with feedback loops is fundamental. We give several examples of systems built with feedback loops to see what they can teach us for the general case.

3.1 Feedback loops

The notion of a *feedback loop* is a basic element of system theory. A feedback loop consists of three elements that together interact with a subsystem: an element that monitors the state of the subsystem, an element that calculates a corrective action, and an element that applies the corrective action to the subsystem. The overall system can be described as a graph of interacting feedback loops. Feedback loops can interact in two main ways. The simplest interaction is where both loops affect interdependent system parameters, i.e., they interact through their environment. This is called stigmergy. A second form of interaction is where a loop manages the next innermost feedback loop, i.e., it continuously adapts the policy implemented by the inner loop. The system's global behavior depends on all the feedback loops taken together.

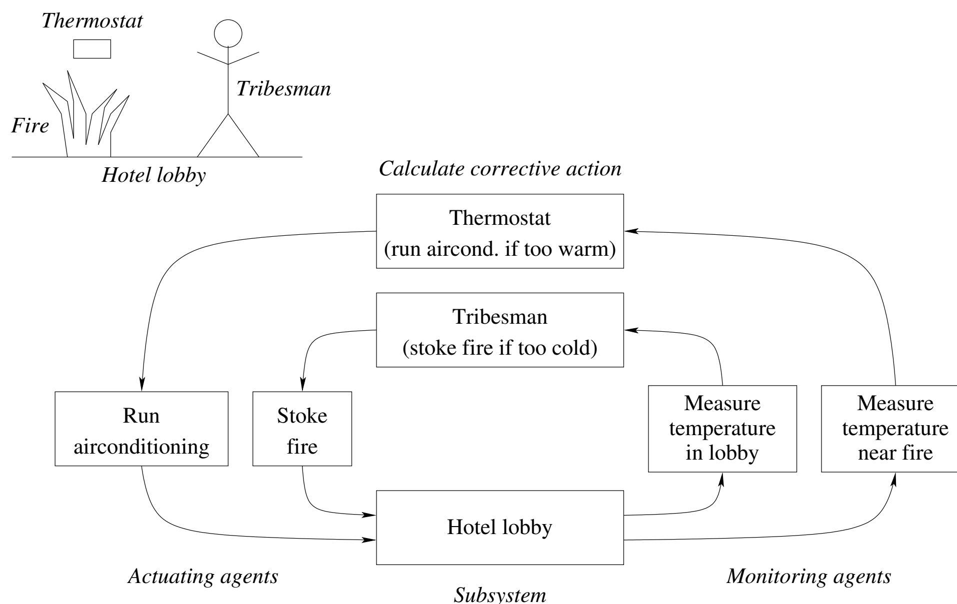

Fig. 1. Wiener's example of two feedback loops interacting through stigmergy

Figure 1 illustrates a a classic example given by Wiener of two interacting feedback loops with counterintuitive global behavior: in an airconditioned hotel, a primitive tribesman attempts to warm himself by starting a fire. This causes the airconditioning to work harder, so the result is that the harder he stokes the fire, the lower the temperature becomes. In this example, the two loops affect system parameters that depend on each other, namely the temperature in different parts of the lobby. In the figure, each block is a concurrent agent continuously sending asynchronous messages to the other agents according to the arrows. Even though each loop taken in isolation uses negative feedback and is stable, [3] the result of both loops is that the system becomes unstable, i.e., the temperature will continue to decrease (until the system reaches a boundary, and then its behavior will change again). A simple rule of thumb is that it is not enough to add a negative feedback loop to a system to ensure stability! The result may well be unstable because of the new loop's interaction with the system. In the next section we will give an example of the second form of interaction, where one loop manages the other.

Designing systems with feedback has been extensively studied in electronics, typically with building blocks such as operational amplifiers and phase-locked loops. These systems exploit the fact that there is a good (piecewise) linear approximation of their behavior. This is a strong condition that can be exploited. But linearity is probably too strong a condition to impose on computer systems, which are highly nonlinear by default, e.g., changing a single bit can have major effects. It may be possible to have a weaker property than linearity that can be satisfied by computer systems and also give a satisfactory design theory. The approach then is to choose first a property that facilitates reasoning about the program and its global behavior, and then to build a program that satisfies the property. This can greatly simplify

[3] In negative feedback, an increase in the monitored value of a system parameter gives a decrease in the corrective action, which results in a decrease in the value of the system parameter. In positive feedback, it results in an increase in the corrective action.

program design. Note that one possible failure mode is that the property itself no longer holds.

One example property is monotonicity or strict monotonicity. In a strict monotonic system, when the input changes in the same direction (e.g., increases, in a general sense), the output will also change in the same direction. Using monotonicity as the basic property is sufficient for designing systems with feedback. A negative feedback amplifier can be built using strict monotonicity. Another property weaker than linearity that may be useful is continuity, but continuity is in general not enough to guarantee stability. We note that two further properties that may be useful in a theory of self management are determinism and confluence.

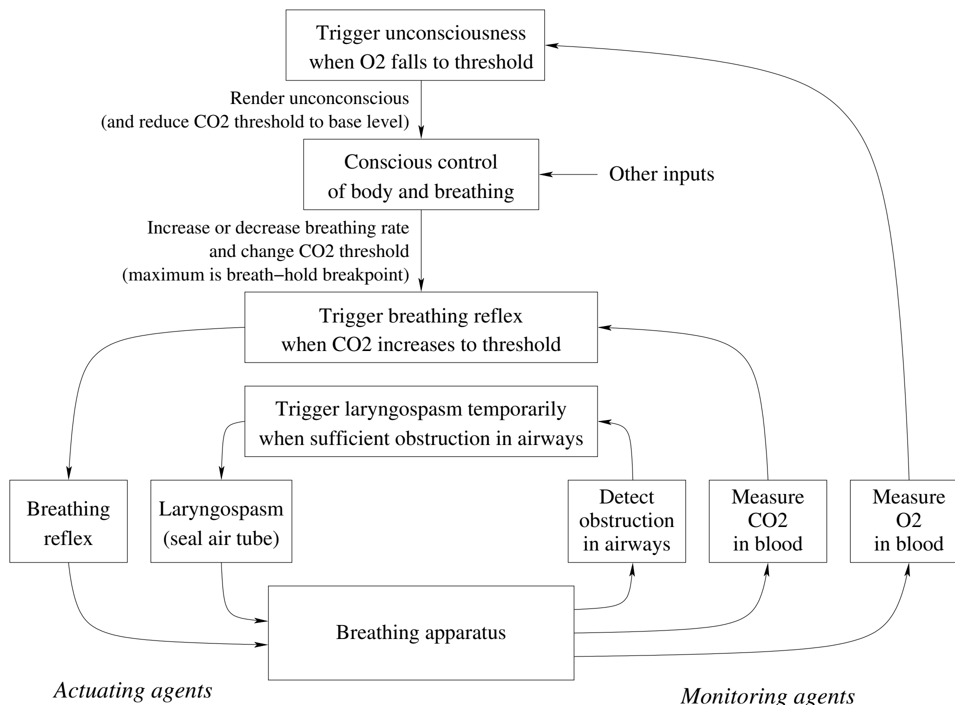

Fig. 2. Feedback loop structure of the human respiratory system

3.2 System design with feedback loops: the human respiratory system

Let us give a detailed example of a system designed with feedback loops. Our example is taken from a biological system, namely the human body. Biological systems have to survive in particularly harsh environmental conditions that are far removed from the comfortable environments that most of our current computers live in. For that reason, we consider that studying biological systems is a useful way to get insight in how to design software for a more complex system. Our example is the human respiratory system, which is a particularly interesting and well-understood biological system. Figure 2 shows the different components of this system and how they interact. We derived this figure from a precise medical description of the system's behavior [21]. The figure is slightly simplified when compared to reality because we have left out interactions with the rest of the body. Nevertheless 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 unconsciousness). 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 and we temporarily cannot breath (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 normal breathing reflex is reestablished.

We can infer some probable design rules of thumb 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 nonlinear and may run both with negative or positive feedback. It is the most complex of the four loops by far. We can justify why it is sandwiched in between two simpler loops. On the one 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 nested feedback loops that implement abstraction. On the other side, the outermost loop overrides the conscious control 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 (but not all) of the body's feedback loop structures. This very power means that it needs a check.

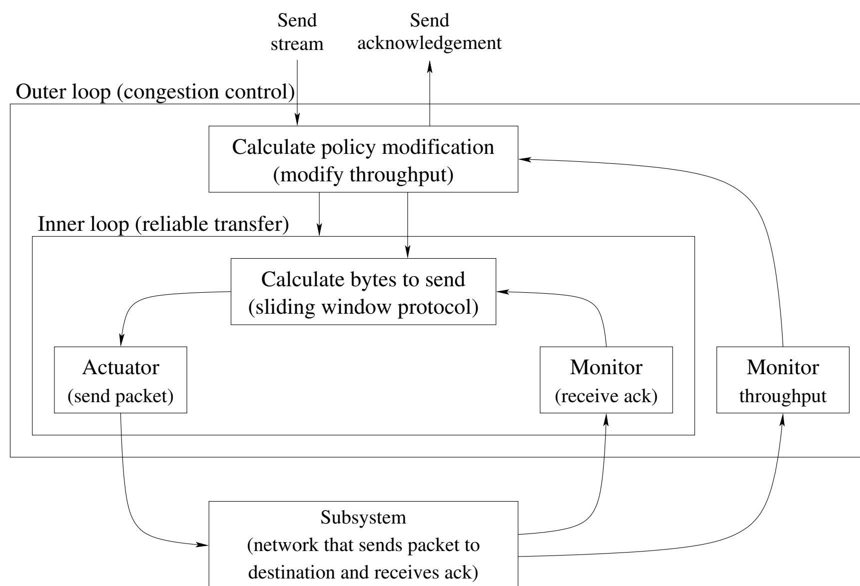

Fig. 3. An example programming pattern with two nested feedback loops

3.3 A new way of designing programs

This style of system design can be applied to programming. Programming then consists of building hierarchies of interacting feedback loops. Let us give a simplified example with two nested feedback loops that implements a reliable byte stream transfer protocol with congestion control (this is a variant of the TCP protocol). The protocol sends a byte stream from a source to a destination node. Figure 3 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 that tell it which packets still need to be sent. The 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 by either changing the policy of the inner loop or by changing the inner loop itself. If the buffered send stream grows too big or the rate of acknowledgements decreases, then it modifies how the inner loop works, for example by reducing the rate of send acknowledgement or the rate of sending. If the transfer stops then the outer loop may terminate the inner loop and abort the transfer.

This structure is a special case of a multi-agent system. Each block in Figure 3 is a single agent acting concurrently with the others and sending messages asynchronously to the others. Because the system is distributed over two nodes, part of the design consists in situating each agent on a node. Each of the two feedback loops implements one task according to a given policy. The policy of the inner loop is determined by the outer loop.

The example of Figure 3 has just two nested feedback loops. In a real system, there will typically be more nested feedback loops. In particular, the outermost loop determines the main interface between the system and its environment.

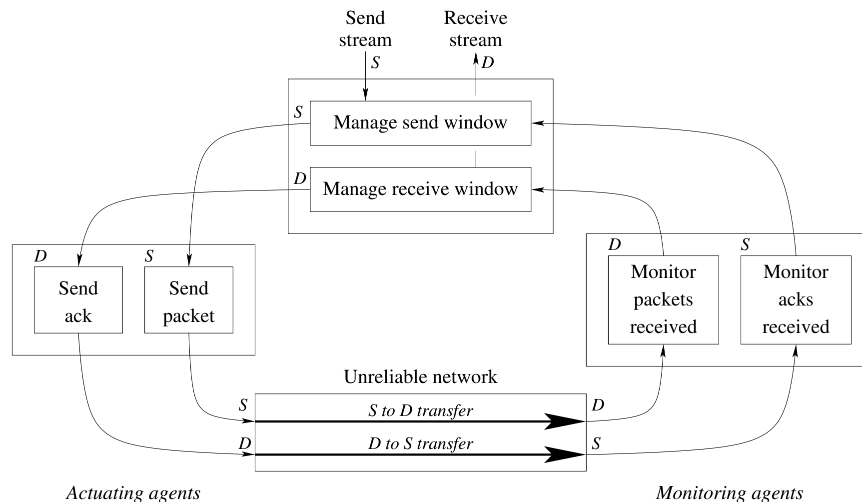

Fig. 4. Inner loop of the reliable byte stream protocol showing distribution

3.4 Interaction between feedback loops and distribution

The protocol of Figure 3 runs on a distributed system consisting of two nodes. Figure 3 only shows what happens at the source node. Figure 4 gives a more complete depiction of the inner loop of Figure 3 that shows the execution on both nodes. In Figure 4, each component is annotated with S or D depending on whether it executes on the source or destination node. We see that the feedback loop is distributed over both nodes. An interesting open question is how to design distributed feedback loops. This is nontrivial because of the interactions between the design of the loop, its distribution, and the partial failures that it is intended to tolerate. Designing these systems is still mostly an open research question. Structured overlay networks are a special case that is being closely investigated [1]. Other special cases include parts of distributed algorithm theory such as self-stabilizing systems [22]. These systems are able to survive large classes of transient faults.

4 Related work

In several areas of computer science the feedback loop architecture is already being used to some degree. This section gives two examples, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior, and compares them to the feedback loop architecture.

The Erlang system is designed to build distributed systems that survive software and hardware faults [2]. It has been successfully used to build systems of extremely high dependability, for example the AXD301 ATM switch which has a claimed down time of only 30 milliseconds per year [20]. Erlang uses a concept called *supervisor tree* to manage the fault tolerance. Each internal node in the supervisor tree corresponds to a feedback loop in our architecture. An Erlang program is organized as a set of agents that communicate through asynchronous message passing. These agents form the leaves of the supervisor tree. The first internal level in the tree consists of agents that observe parts of the program's execution. If a program agent fails, then an observer agent will restart it and its siblings in a consistent state, using a database to get the consistent state. The second internal level in the supervisor tree consists of a root agent that handles failures of the observer agents. This root agent must be completely reliable. This is possible because the root agent is a very small program.

The subsumption architecture of Rodney Brooks shares features with the feedback loop architecture [6,7]. The subsumption architecture is completely concurrent and distributed. It has been used to successfully implement systems that interact with their environment in a life-like fashion. The subsumption architecture is a way to implement systems by decomposing complex behaviors into layers. The layers are given priorities. Each layer observes the world continuously. If a layer can react, then it disables the lower layers and performs its own actions. In the terminology of Brooks, it suppresses inputs to the lower layers and inhibits outputs from the lower layers. For example, an obstacle-avoiding robot could have three layers: a move forward layer, a turn layer, and an avoiding layer. The default behavior is move forward. If the direction is wrong, then the turn layer disables the move forward layer to turn. If there is an obstacle, then the avoiding layer disables the other two

layers and performs an obstacle avoidance maneuver. This is a simple example that shows the basic principle. There exist more refined versions of the architecture.

In the subsumption architecture, the feedback loops interact through shared system parameters. E.g., in a robot, all the loops detect the robot's position and control the robot's movements. In the feedback loop architecture, feedback loops can also have a policy/mechanism relationship, where each loop modifies the policy that is implemented by the next innermost loop.

5 General architectural framework

Let us now take a step back from the above examples and summarize what a general architectural framework can look like for building a self-managing system. Such a system consists of a possibly large number of interacting parts. The system is organized as a set of concurrent components that communicate through asynchronous message passing. The default is therefore that components are independent. Any synchronous or dependent behavior must be programmed explicitly. This default gives good results for systems intended to handle software faults, such as the Erlang system described above [2], and other cases as well such as security in distributed systems, implemented in the E system [16]. It also matches well with the complex systems approach taken in physics.

Following the examples of Sections 3.2-3.4 and Section 4, the system consists of a hierarchy of interacting feedback loops, where each feedback loop is implemented by several agents and each agent is an instance of a component. Feedback loops interact either through stigmergy or through management.

5.1 Higher-order component model

In a self-managing system, the system is able to monitor and reconfigure itself, that is, install and update parts of itself while it is running. If the system is built as a set of interacting components then it is possible for components to install other components. Components are therefore first-class entities that can be passed as arguments to other components. This is called *higher-order* component programming. We consider the Fractal model as a good starting point [8]. This model is already being used as a framework for building self-managing systems [5]. In a higher-order component model, it takes some care to determine what component is to blame when a subsystem fails. This has been studied by Findler and Blume [12].

5.2 Global properties

An important part of general system theory is to determine the global properties of a system. This aspect is especially important for large-scale computer systems, such as the Internet or distributed systems built on top of the Internet. Some of the important questions are is the system stable, how does it behave when stressed, and can its imminent collapse be detected before it happens? Answers to some of these questions have been given in the study of complex systems in physics. Such systems consist of large numbers of very simple components, but even so they can be a useful approximation for computer systems. For example, Krishnamurthy *et*

al [14] have done an analytic study of the Chord structured overlay network using a master equation approach. Another example is the belief propagation algorithm. This algorithm is defined in terms of message passing between large numbers of simple nodes [23]. It has been used to give solutions to the SAT problem and other problems. Belief propagation is a general technique that can determine global properties of a system in terms of local properties. It can be used for monitoring global properties as part of a feedback loop.

6 Conclusions

This paper motivates that a good approach for building large-scale distributed systems is to consider them as general self-managing systems. We propose to build self-managing systems as sets of concurrent agents interacting through asynchronous message passing and implemented using a component model that realizes first-class components and component instances. In this framework, self-managing systems are built as hierarchies of interacting feedback loops. Feedback loops interact through two mechanisms, stigmergy (shared environment parameters) or management (one loop controls another). The global behavior of the system depends on all feedback loops taken together. We relate this proposal to two other architectures, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior. These ideas are being realized in SELFMAN, a project in the European 6th Framework Programme that started in June 2006 [18]. We intend to elaborate these ideas into a programming methodology.

References

[1] Aberer, K., L. Onana Alima, A. Ghodsi, S. Girdzijauskas, M. Hauswirth, and S. Haridi, *The essence of P2P: A reference architecture for overlay networks*, 5th International Conference on Peer-to-Peer Computing (P2P 05), IEEE Computer Society, 2005.

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

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

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

[5] Bouchenak, S., F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quéma, and J.-B. Stefani, *Architecture-Based Autonomous Repair Management: Application to J2EE Clusters*, 2nd International Conference on Autonomic Computing (ICAC'05), 2005, pp. 369–370.

[6] 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.

[7] Brooks, Rodney A., *Intelligence without representation*, Artificial Intelligence 47, 1991, pp. 139–159.

[8] Bruneton E., V. Quéma, T. Coupaye, M. Leclercq, and J.-B. Stefani, *An Open Component Model and its Support in Java*, Proceedings 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Springer LNCS 3054, 2004.

[9] Carroll, Lewis, "Through the Looking-Glass and What Alice Found There," 1872 (Dover Publications reprint 1999).

[10] Ceruzzi, Paul E., "Beyond the Limits: Flight Enters the Computer Age," MIT Press, Cambridge, MA, 1989.

[11] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, *PlanetLab: An Overlay Testbed for Broad-Coverage Services*, ACM SIGCOMM Comp. Comm. Review, 33(3), 2003.

[12] Findler, Robert Bruce, and Matthias Blume, *Contracts as Pairs of Projections*, FLOPS 2006, April 24-26, 2006.

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

[14] Krishnamurthy, S., S. El-Ansary, E. Aurell, and S. Haridi, *A statistical theory of Chord under churn*, The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), 2005.

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

[16] Miller, Mark, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, May 2006.

[17] Principia Cybernetica Web. Entry "system," August 2006. Internet: `http://pespmc1.vub.ac.be/ASC/SYSTEM.html`.

[18] Van Roy, Peter, Ali Ghodsi, Seif Haridi, Jean-Bernard Stefani, Thierry Coupaye, Alexander Reinefeld, Ehrhard Winter, and Roland Yap, *Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components*, CoreGRID Technical Report TR-0018, Dec. 14, 2005.

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

[20] Wiger, Ulf, *Four-fold increqse in productivity and quality* industrial-strength functional programming in telecom-class products, Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems, 2001.

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

[22] Wikipedia, the free encyclopedia. Entry "self-stabilization," August 2006. Internet: `http://en.wikipedia.org/wiki/Self-stabilization`.

[23] Yedidia, J.S., W.T. Freeman, and Y. Weiss, *Understanding Belief Propagation and Its Generalizations*, Exploring Artificial Intelligence in the New Millennium, Chap. 8, Jan. 2003. Also MERL Technical Report TR-2001-22, Jan. 2002.