

Grassroots Approach to Self-Management in Large-Scale Distributed Systems^{*}

Ozalp Babaoglu, Márk Jelasity^{**}, and Alberto Montresor

Department of Computer Science, University of Bologna, Italy
e-mail: babaoglu,jelasity,montreso@cs.unibo.it

Abstract. Traditionally, autonomic computing is envisioned as replacing the human factor in the deployment, administration and maintenance of computer systems that are ever more complex. Partly to ensure a smooth transition, the design philosophy of autonomic computing systems remains essentially the same as traditional ones, only autonomic components are added to implement functions such as monitoring, error detection, repair, etc. In this position paper we outline an alternative approach which we call “grassroots self-management”. While this approach is by no means a solution to all problems, we argue that recent results from fields such as agent-based computing, the theory of complex systems and complex networks can be efficiently applied to achieve important autonomic computing goals, especially in very large and dynamic environments. Unlike in traditional compositional design, the desired properties like self-healing and self-optimization are not programmed explicitly but rather “emerge” from the local interactions among the system components. Such solutions are potentially more robust to failures, are more scalable and are extremely simple to implement. We discuss the practicality of grassroots autonomic computing through the examples of data aggregation, topology management and load balancing in large dynamic networks.

1 Introduction

The desire to build fault tolerant computer systems with an intuitive and efficient user interface has always been part of the research agenda of computer science. Still, the current scale and heterogeneity of computer systems is becoming alarming, especially because our everyday life has come to depend on such systems to an increasing degree. There is a general feeling in the research community that to cope with this new situation—which emerged as a result of Moore’s Law, the widespread adoption of the Internet and computing becoming pervasive in general—needs radically new approaches to achieve seamless and efficient functioning of computer systems.

^{*} This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Projects BISON (IST-2001-38923) and DELIS (IST-001907).

^{**} also with RGAI, MTA SZTE, Szeged, Hungary

Accordingly, more and more effort is devoted to tackle the problem of self-management. One of the most influential and widely publicized approach is IBM's *autonomic computing* initiative, launched in 2001 [9]. The term "autonomic" is a biological analogy referring to the autonomic nervous system. The function of this system in our body is to control "routine" tasks like blood pressure, hormone levels, heart rate, breathing rate, etc. At the same time, our conscious mind can focus on high level tasks like planning and problem solving. The idea is that autonomic computing should do just the same: computer systems would take care of routine tasks themselves while system administrators and users would focus on the actual task instead of spending most of their time troubleshooting and tweaking their systems.

Since the original initiative, the term has been adopted by the wider research community although it is still strongly associated with IBM and, more importantly, IBM's specific approach to autonomic computing. It is somewhat unfortunate because the term *autonomic* would allow for a much deeper and more far-reaching interpretation, as we explain soon. In short, we should not only take it seriously *what* the autonomic nervous system does but also *how* it does it. We believe that the remarkably successful self-management of the autonomic nervous system, and biological organisms in general lies exactly in the *way* they achieve this functionality. Ignoring the exact mechanisms and stopping at the shallow analogy at the level of function description misses some important possibilities and lessons that can be learned by computer science.

1.1 The meaning of "self"

The traditional approach to autonomic computing is to replace human system administrators with software or hardware components that continuously monitor some subsystem assigned to them, forming so called *control loops* [9] which involve monitoring, knowledge based planning and execution (see Figure 1). Biological systems however achieve self-management and control through entirely different, often fully distributed and *emergent* ways of processing information. In other words, the usual biological interpretation of self-management involves no managing and managed entities. There is often no subsystem responsible for self-healing or self-optimization; instead, these properties simply follow from some simple local behavior of the components typically in a highly non-trivial way. The term "self" is meant truly in a grassroots sense, and we believe that this fact might well be the reason of many desirable properties like extreme robustness and adaptivity, with the additional benefit of a typically very simple implementation.

1.2 Trust

There are a few practical obstacles in the way of the deployment of grassroots self-management. One is the entirely different and somewhat un-natural

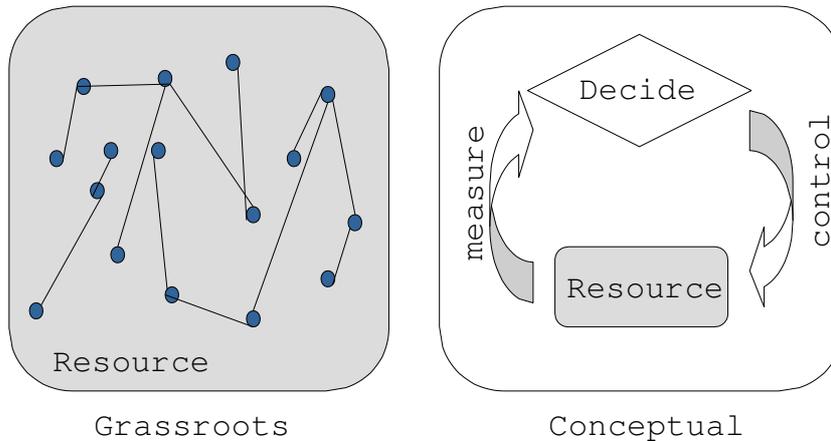


Fig. 1. Models of self-management.

way of thinking and the relative lack of understanding of the principles of self-organization and emergence [12]. Accordingly, *trust delegation* represents a problem: psychologically it is more relaxing to have a single point of control, an explicit controlling entity. In the case of the autonomic nervous system we cannot do anything else but trust it, although probably many people would prefer to have more control, especially when things go wrong. Indeed, the tendency in engineering is to try to isolate and create central units that are responsible for a function. A good example is the car industry that gradually places more and more computers into our cars that explicitly control the different functions, thereby replacing old and proven mechanisms that were based on some, in a sense, self-optimizing mechanism (like the carburetor) and so also sacrificing the self-healing and robustness features of these functions to some degree.

1.3 Modularity

To exploit the power and simplicity of emergent behavior yet to ensure that these mechanisms can be trusted and be incorporated in systems in an informed manner, we believe that a *modular paradigm* is required. The idea is to identify a collection of simple and predictable services as *building blocks* and combine them in arbitrarily complex functions and protocols. Such a modular approach presents several attractive features. Developers will be allowed to plug different components implementing a desired function into existing or new applications, being certain that the function will be performed in a predictable and dependable manner. Research may be focused on the development of simple and well-understood building blocks, with a particular emphasis on important properties like robustness, scalability, self-organization and self-management.

The goal of this position paper is to promote this idea by describing our preliminary experiences in this direction. Our recent work has resulted in a collec-

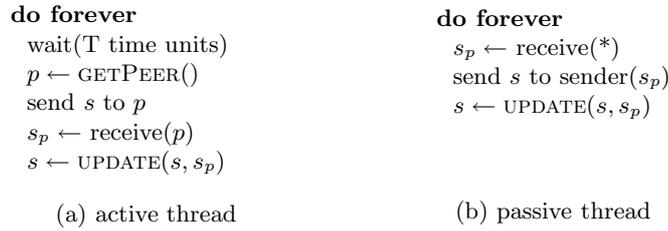


Fig. 2. The skeleton of a gossip-based protocol. Notation: s is the local state, s_p is the state of the peer p .

tion of simple and robust building blocks, which include *data aggregation* [7, 11], *membership management* [6], *topology construction* [4, 10] and *load balancing* [8]. Our building blocks are typically no more complicated than a cellular automaton or a swarm model which makes them ideal objects for research. Practical applications based on them can also benefit from a potentially more stable foundation and predictability, a key concern in fully distributed systems. Most importantly, they are naturally self-managing, without dedicated system components. In the rest of the paper, we briefly describe these components.

2 A Collection of Building Blocks

Under the auspices of the BISON project [1], our recent activity has been focused on the identification and development of protocols for several simple basic functions. The components produced so far can be informally subdivided into two broad categories: *overlay protocols* and *functional protocols*. An overlay protocol is aimed at maintaining application-layer, connected communication topologies over a set of distributed nodes. These topologies may constitute the basis for functional protocols, whose task is to compute a specific function over the data maintained at nodes.

Our current bag of protocols includes: (i) protocols for organizing and managing structured topologies like super-peer based networks (SG-1 [10], grids and tori (T-MAN [4])); (ii) protocols for building unstructured networks based on the random topology (NEWSCAST [6]); (iii) protocols for the computation of a large set of aggregate functions, including maximum and minimum, average, sum, product, geometric mean, variance, etc [7, 11]; and (iv) a load balancing protocol [8].

The relationships between overlay and functional protocols may assume several different forms. Topologies may be explicitly designed to optimize the performance of a specific functional protocol (this is the case of NEWSCAST [6] used to maintain a random topology for aggregation protocols). Or, a functional protocol may be needed to implement a specific overlay protocol (in superpeer networks, aggregation can be used to identify the set of superpeers).

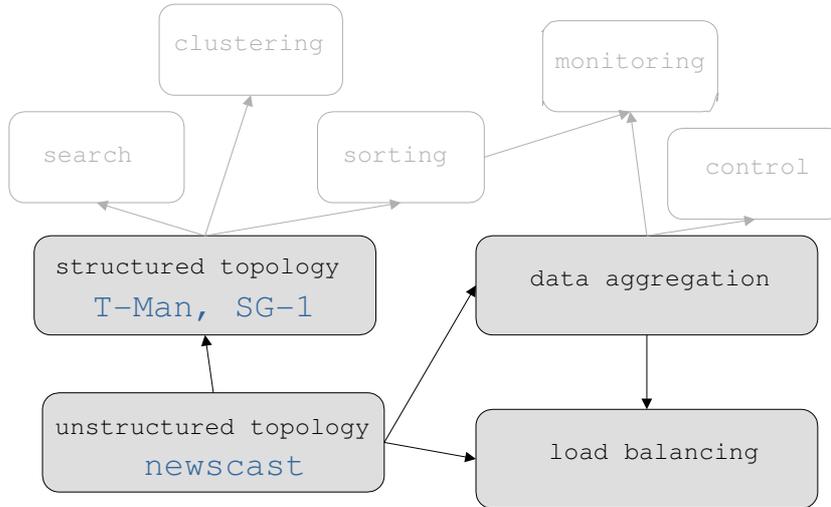


Fig. 3. Dependence relations between components.

All the protocols we have developed so far are based on the gossip-based paradigm [2, 3]. Gossip-style protocols are attractive since they are extremely robust to both computation and communication failures. They are also extremely responsive and can adapt rapidly to changes in the underlying communication structure, just by their nature, without extra measures.

The skeleton of a generic gossip-based protocol is shown in Figure 2. Each node possesses a local state and executes two different threads. The *active* one periodically initiates an *information exchange* with a peer node selected randomly, by sending a message containing the local state and waiting for a response from the selected node. The *passive* one waits for messages sent by an initiator and replies with its local state.

Method `UPDATE` builds a new local state based on the previous local state and the state received during the information exchange. The output of `UPDATE` depends on the specific function implemented by the protocol. The local states at the two peers after an information exchange are not necessarily the same, since `UPDATE` may be non-deterministic or may produce different outputs depending on which node is the initiator.

Even though our system is not synchronous, it is convenient to talk about *cycles* of the protocol, which are simply consecutive wall clock intervals during which every node has its chance of performing an actively initiated information exchange.

In the following we describe the components. Figure 3 illustrates the dependence relations between them as will be described in the text as well.

2.1 Newscast

In NEWSCAST [6], the state of a node is given by a *partial view*, which is a set of peer descriptors with a fixed size c . A *peer descriptor* contains the address of the peer, along with a *timestamp* corresponding to the time when the descriptor was created.

Method GETPEER returns an address selected randomly among those in the current partial view. Method UPDATE merges the partial views of the two nodes involved in an exchange and keeps the c freshest descriptors, thereby creating a new partial view. New information enters the system when a node sends its partial view to a peer. In this step, the node always inserts its own, newly created descriptor into the partial view. Old information is gradually and automatically removed from the system and gets replaced by new information. This feature allows the protocol to “repair” the overlay topology by forgetting dead links, which by definition do not get updated because their owner is no longer active.

In NEWSCAST, the overlay topology is defined by the content of partial views. We have shown in [6] that the resulting topology has a very low diameter and is very close to a random graph with out-degree c . According to our experimental results, choosing $c = 20$ is already sufficient for very stable and robust connectivity.

We have also shown that, within a single cycle, the number of exchanges per node can be modeled by a random variable with the distribution $1 + \text{Poisson}(1)$. The implication of this property is that no node is more important (or overloaded) than others.

2.2 T-Man

Another component is T-MAN [4], a protocol for creating a large set of topologies. The idea behind the protocol is very similar to that of NEWSCAST. The difference is that instead of using the creation date (freshness) of descriptors, T-MAN applies a ranking function that ranks any set of nodes according to increasing distance from a base node. Method GETPEER returns neighbors with a bias towards closer ones, and, similarly, UPDATE keeps peers that are closer, according to the ranking.

Figure 4 illustrates the protocol, as it constructs a torus topology. In [4] it was shown that the protocol converges in logarithmic time also for network sizes as large as 2^{20} and for other topologies as well including the ring and binary tree topologies. With the appropriate ranking function T-MAN can be also applied to sort a set of numbers.

This component, T-MAN, relies on another component for generating an initial random topology which is later evolved into the desired one. In our case this service is provided by NEWSCAST.

2.3 SG-1

SG-1 [10] is yet another component based on NEWSCAST, whose task is to self-organize a *superpeer-based* network. This special kind of topology is organized

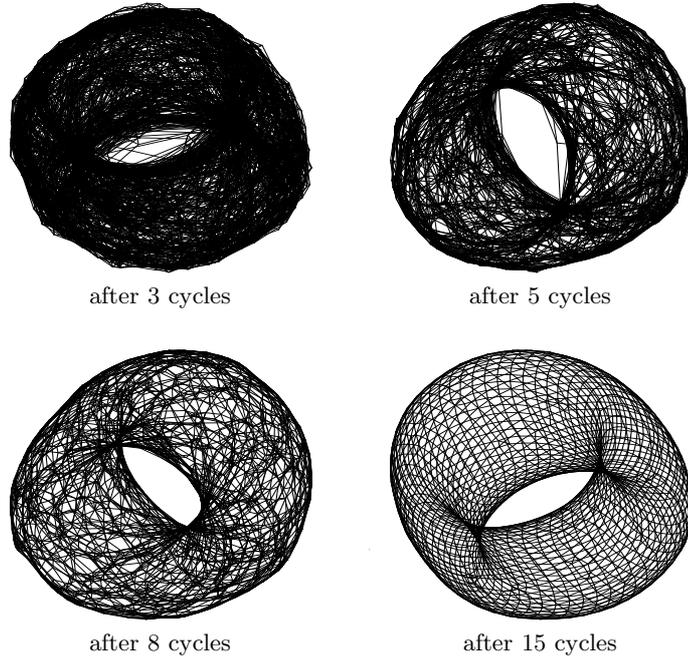


Fig. 4. Illustrative example of T-MAN constructing a torus over $50 \times 50 = 2500$ nodes, starting from a uniform random topology with $c = 20$. For clarity, only the nearest 4 neighbors (out of 20) of each node are displayed.

through a two-level hierarchy, as illustrated in Figure 5: nodes that are faster and/or more reliable than “normal” nodes take on server-like responsibilities and provide services to a set of clients. The superpeer paradigm allows decentralized networks to run more efficiently by exploiting heterogeneity and distributing load to machines that can handle the burden. On the other hand, it does not inherit the flaws of the client-server model, as it allows multiple, separate points of failure, increasing the health of large-scale networks.

In our model, each node is characterized by a *capacity* parameter, that defines the maximum number of clients that can be served by a node. The task of SG-1 is to form a network where the role of superpeers is played by the nodes with highest capacity. All other nodes become clients of one or more superpeers. The goal is to identify the minimal set of superpeers that are able to provide the desired quality of service, based on their capacity.

In SG-1, NEWSCAST is used in two ways. First, it provides a robust underlying topology that guarantee the connectivity of the network in spite of superpeer failures. Second, NEWSCAST is used to maintain, at each node, a partial view containing a random sample of superpeers that are currently *underloaded* with respect to their capacity. At each cycle, each superpeer s tries to identify a superpeer t that (i) has more capacity than s , and (ii) is underloaded. If such

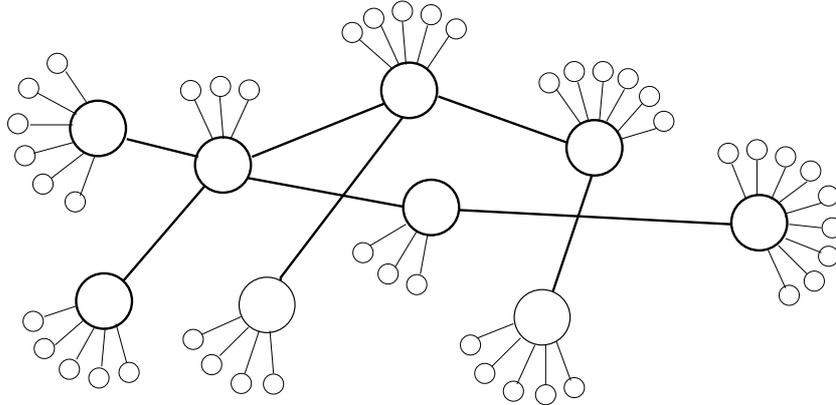


Fig. 5. A superpeer topology. Superpeers (thick circles) are connected together through a random network, while clients (thin circles) are associated to a single superpeer.

superpeer exist and can be contacted, s transfers the responsibility of parts of its clients to t . If the set of clients of s ends to be empty, s becomes a client of t .

Experimental results show that this protocol converges to the target superpeer topology in logarithmic time for network sizes as large as 10^6 nodes, producing very good approximation of the target in a constant number of cycles.

2.4 Gossip-Based Aggregation

In the case of gossip-based aggregation [7, 11], the state of a node is a numeric value. In a practical setting, this value can be any attribute of the environment, such as the load or the storage capacity. The task of the protocol is to calculate an aggregate value over the set of all numbers stored at nodes. Although several aggregate functions may be computed by our protocol, in this paper provide only the details for the average function.

In order to work, this protocol needs an overlay protocol that provides an implementation of method `GETPEER`. Here, we assume that this service is provided by `NEWSCAST`, but any other overlay could be used.

To compute the average, method `UPDATE(a, b)` must return $(a + b)/2$. After one state exchange, the sum of the values maintained by the two nodes does not change, since they have just balanced their values. So the operation does not change the global average either; it only decreases the variance over all the estimates in the system.

In [7] it was shown that if the communication topology is not only connected but also sufficiently random, at each cycle the empirical variance computed over the set of values maintained by nodes is reduced by a factor whose expected value is $2\sqrt{e}$. Most importantly, this result is independent from the size of the network, showing the extreme scalability of the protocol.

In addition to being fast, our aggregation protocol is also very robust. Node failures may perturb the final result, as the values stored in crashed nodes are lost; but both analytical and empirical studies have shown that this effect is generally marginal [11]. As long as the overlay network remains connected, link failures do not modify the final value, they only slow down the aggregation process.

2.5 A Load-Balancing Protocol

The problem of load balancing is similar, to a certain extent, to the problem of aggregation. Each node has a certain amount of load and the nodes are allowed to transfer some portions of their load between themselves. The goal is to reach a state where each node has the same amount of load. To this end, nodes can make decisions for sending or receiving load based only on locally available information. Differently from aggregation, however, the amount of load that can be transferred in a given cycle is bounded: the transfer of a unit of load may be an expensive operation. In our present discussion, we use the term *quota* to identify this bound and we denote it by Q . Furthermore, we assume that the quota is the same at each node.

A simple, yet far from optimal idea for a completely decentralized algorithm could be based on the aggregation mechanism illustrated above. Periodically, each node contacts a random node among its neighbors. The loads of the two nodes are compared; if they differ, a quantity q of load units is transferred from the node with more load to the node with less load. q is clearly bounded by the quota Q and quantity of load units needed to balance the nodes.

If the network is connected, this mechanism will eventually balance the load among all nodes. Nevertheless, it fails to be optimal with respect to load transfers. The reason is simple: if the loads of two nodes are both higher than the average load, transferring load units from one to the other is useless. Instead, they should contact nodes whose load is smaller than the average, and perform the transfer with them.

Our load-balancing algorithm is based exactly on this intuition. The nodes obtain an estimate of the current average load through the aggregation protocol described above. This estimate is the target load; based on its value, a node may decide if it is overloaded, underloaded, or balanced. Overloaded nodes contact their underloaded neighbors in order to transfer their excess load and underloaded nodes contact their overloaded neighbors to perform the opposite operation. Nodes that have reached the target load stop participating in the protocol. Although this was a simplified description, it is easy to see that this protocol is optimal with respect to load transfer, because each node transfers exactly the amount of load needed to reach its target load. As we show in [8], the protocol is also optimal with respect to speed under some conditions on the initial load distribution.

3 Notes on Combining the Building Blocks

The combination of the building blocks is done in the traditional way: a building block has a local interface (within one node) towards the other components, and it has a protocol and an implementation associated to it. The implementation can differ over different nodes, just like the local interface, so we focused on the protocols in the discussion above. Nodes using the same protocol (for example, aggregation) form a “layer” in the system. This is however not a layer in the usual sense: we allow for arbitrary dependency relations between the building blocks. In general, the directed graph that describes the dependencies (such as the example shown in Figure 3) will not contain cycles, however, this is not strictly required. Two “layers” can mutually depend on each other’s services, for instance, in a bootstrapping phase when they can catalyze each other’s performance.

Having said that, we need to mention one exception. There is a “lowest layer” in our framework, which in a sense represents the group abstraction: the set of nodes that form the domain of the other components. This layer is the random network component, which provides the *peer sampling service* (see Section 2.1 and [5]). The main requirement for this service is that it must return all nodes with equal probability, in particular, no nodes are allowed to have zero probability of being returned. The peer sampling service is used to support and bootstrap other services like aggregation or structured topologies.

Finally, the aspect of *time-scale* should also be noted. The protocols are all based on the scheme given in Figure 2, however, the waiting time T can be different for different building blocks. This degree of freedom allows for certain architectures that otherwise would not be possible, for example, if an up-to-date aggregate value is needed instantly according to the timescale of a relatively slow layer, like load balancing, then we can simply apply aggregation at a relatively faster timescale.

4 Conclusions

In this paper, we presented examples for simple protocols that exhibit self-managing properties without any explicit management components or control loops; in other words, without increased complexity. We argued that a modular approach might be the way towards efficient deployment of such protocols in large distributed systems. To validate our ideas, we have briefly presented gossip based protocols as possible building blocks: topology and membership management (T-MAN, SG-1 and NEWSCAST), aggregation, and load balancing.

References

1. The Bison Project. <http://www.cs.unibo.it/bison>.
2. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database management. In *Proceedings of the 6th Annual ACM Symposium on*

- Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, August 1987. ACM.
3. Patrick Th. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, May 2004.
 4. Márk Jelasity and Ozalp Babaoglu. T-Man: Fast gossip-based construction of large-scale overlay topologies. Technical Report UBLCS-2004-7, University of Bologna, Department of Computer Science, Bologna, Italy, May 2004. <http://www.cs.unibo.it/techreports/2004/2004-07.pdf>.
 5. Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Hans-Arno Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
 6. Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, November 2003.
 7. Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 102–109, Tokyo, Japan, 2004. IEEE Computer Society.
 8. Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In Giovanna Di Marzo Serungendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising Systems*, volume 2977 of *Lecture Notes in Artificial Intelligence*, pages 265–282. Springer, 2004. invited paper.
 9. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.
 10. Alberto Montresor. A robust protocol for building superpeer overlay topologies. In *Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing (P2P'04)*, pages 202–209, Zurich, Switzerland, August 2004. IEEE Computer Society.
 11. Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN)*, pages 19–28, Florence, Italy, 2004. IEEE Computer Society.
 12. Julio M. Ottino. Engineering complex systems. *Nature*, 427:399, January 2004.