Appears in Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems. (San Diego, CA, 11 June 2003), pages 1–6. ACM Press. 2003

A Human Organization Analogy for Self-* Systems

John D. Strunk, Gregory R. Ganger
Carnegie Mellon University

ABSTRACT
The structure and operation of human organizations, such as corporations, offer useful insights to designers of self-∗ systems (a.k.a. self-managing or autonomic). Examples include worker/supervisor hierarchies, avoidance of micro-management, and complaint-based tuning. This paper explores the analogy, and describes the design of a self-∗ storage system that borrows from it.

1. INTRODUCTION
A popular research topic these days is the pursuit of "self-∗ systems:" self-organizing, self-configuring, self-tuning, self-repairing, self-managing systems of cost-effective components (e.g., "bricks" or "blades"). Such research is a direct response to the shift from needing bigger, faster, stronger computer systems to the need for less human-intensive management of the systems currently available. System complexity has reached the point where administration generally costs more than hardware and software infrastructure.

In the course of describing self-∗ systems, several analogies have been offered, attempting to draw inspiration from natural systems such as insect collectives or the autonomic nervous system. This paper discusses another analogy source—human organizations, such as corporations or militaries—and the insights it offers. Human organizations successfully combine the efforts of autonomous, imperfect, adaptable entities to achieve a broad range of goals across a broad range of sizes. Compared to natural systems, they accommodate more relevant goals (e.g., customer satisfaction and legal protection vs. survival), and they are more thoroughly understood (e.g., see [10]). Natural systems may well be the best model, when they are better understood, but they provide little guidance to system designers in the meantime.

The theory and practice of human organizations offer several interesting insights for self-∗ system designers. For example, management hierarchies guide and oversee the efforts of workers from outside the critical path. Such management partitions the workload among workers of varying capabilities, provides local goals and policies, and monitors progress and work quality. As another example, human organizations rarely start with detailed performance specifications at any level. Instead, they move forward based on vague expectations and then adjust as they observe the results. Insufficiently timely service produces a clear signal: complaints.

Such insights can be applied directly to the design of self-∗ systems. These systems may have a set of nodes for performing work and a set of supervisory nodes for management tasks. Supervisors could partition work and collect statistics while allowing local optimization to occur unhampered. Human administrators can use complaints to indicate insufficient performance, which is often how they hear about problems, rather than specifying complex service level objectives (SLOs). Supervisor nodes, upon receiving a complaint from the administrator, would translate this abstract input to internal performance goals for various tasks.

As a concrete example, we describe the design of a self-∗ storage system and how it borrows from the analogy of human organizations. In interface and function, storage is simpler than general computation systems. Yet, storage devices' highly non-linear performance characteristics and users' reliability demands make it difficult to perfect. Simultaneously, storage plays a critical role in data centers and seems to require a disproportionate amount of administrative attention (e.g., 1 administrator per 1–2 TB in 2000 [4]). It is clearly a subject worthy of our attention.

2. HUMAN ORGANIZATIONS
As researchers have struggled to develop self-∗ systems, they have looked at existing natural systems for examples with self-∗ properties. This search has produced several popular analogies, such as autonomic computing [6], that evoke images of plumbing that functions, adapts, and repairs itself without requiring attention. Insect collectives (e.g., bee hives and ant colonies) have been another source of inspiration, evoking images of large numbers of expendable, interchangeable individuals autonomously working towards a common good.

On the surface (and for marketing purposes), these analogies look appealing. Beyond these high-level properties, however, it is not clear how much they contribute to self-∗ systems. Most biological systems tend to be inflexible outside of a narrow range, sometimes leading to disastrous results—for example, insects continue to be drawn to bug zapping lamps. Their responses to stimuli are genetically pre-programmed, resulting in very gradual adaptation to new environments with the death of collectives (and mutation of the species) as part of the process. Perhaps most importantly, such systems generally have static, limited policies (e.g., survival). Self-$*$ systems, on the other hand, need to support a broad range of policies across a broad range of environments while being upgradable (e.g., patches) and robust (e.g., death of a collective is unacceptable). Clearly, space remains for inspiration from additional sources.

Humans are able to work together in groups to solve a wide variety of problems relatively efficiently. The primary example that we explore here is that of the corporation. Corporations have a number of properties that are desirable for self-$*$ systems. They are able to exist over a wide range of sizes. For example, corporations exist that have from tens of employees to several hundreds of thousands of employees, and both are successful.[1] They are composed of heterogenous components—the company CEO has a far different skill set than the janitor—and they are resilient to failure. The tasks of employees that leave the organization are redistributed to others with similar abilities either on a temporary basis, until a replacement is hired, or on a more permanent basis. Corporations also show a great deal of versatility. While most have the same general goal, increasing the value of their owners' investment, the methods for achieving it vary widely.

2.1 Insights

Of course, we do not propose to endow system components with human intelligence. The value of an analogy is the set of insights into system design that it provides. Human organizations offer a number of ideas that can guide the design of self-$*$ systems.

Hierarchical management without micro-management: Most human organizations have a tree-like structure in which supervisors delegate tasks, along with a set of goals or deadlines, downward to their subordinates. The main tasks of supervisors are partitioning work among their workers, ensuring they are meeting their assigned goals, and taking corrective action to improve efficiency. Although supervisors delegate tasks, they do not dictate how those tasks should be accomplished. Workers are able to perform tasks in their own way. This division works well since each worker has different strengths and characteristics, implying that they themselves are the best judge of how they will most effectively accomplish their assigned tasks.

As we design self-$*$ systems, this management structure provides an architecture for how the different system components should work together. The separation of task distribution from the implementation of those tasks permits devices to locally optimize based on their individual characteristics. By communicating goals along with tasks, the supervisors enable the individual workers to prioritize their activities such that their optimizations are aligned with the high-level goals of the system.

Complaint-based tuning: Humans are not good at precisely specifying what they want, but they are very good at complaining when they are not satisfied. As a result, most service organizations within corporations do not ask users for quantitative expectations. Instead, they create their own rough, educated guesses and refine performance targets based on any complaints received. It is human nature to be outspoken when things are "broken" (i.e., not living up to expectations), while tasks that are being handled properly are generally taken for granted.

[1] The larger examples have clear inefficiencies, yet they still thrive.

At the highest level, any self-$*$ system will need to interact with a human administrator to receive system goals and priorities. It is unreasonable to expect the administrator to provide detailed, meaningful SLOs—even experts pursuing workload characterization research struggle with how to generate them. Instead, for performance metrics, self-$*$ designs should borrow from the human solution: educated guesses refined by feedback.

Risk analysis: All companies must assess and cope with issues of failure and security. As in the real world, trial-and-error with feedback-based refinement is not sufficient. Companies spend a great deal of effort to create policies for managing risk. Some self-$*$ policies (e.g., reliability and availability) involve similar tradeoffs that require similar analysis.

Try it and see: Making predictions about future system configurations is subject to a significant margin of error. Companies counter this by trying new ideas on a small scale before investing large amounts of resources. Self-$*$ systems can use this "try and see" approach as well. Workload characterization and system models introduce inaccuracies into system performance predictions. By trying new configurations on a small scale first,[2] predictions can be refined and many mistakes corrected with minimal impact on the overall system. Continued reexamination, as the new configuration is deployed at ever larger scales, will still be needed to recover from unanticipated non-linearities.

Observe, diagnose, repair loop: Biological systems act via pre-programmed responses to stimuli, and most do not diagnose problems via investigation and reasoning the way humans do. Humans develop and learn from shared repositories of knowledge. They apply the resulting expertise to deduce the sources of problems, often by testing hypotheses with experimentation, and then take corrective action. Such problem solving will be needed in self-$*$ systems, particularly when physical repairs must be explained and justified to the human administrator.

Shifts, sick days, vacations, and sabbaticals: Human organizations plan for and schedule worker downtime. Doing so makes the workers more effective, allowing them to refresh themselves and to step back from their work to gain perspective. To support such downtime, managers must have sufficient workers and properly coordinate their schedules. Self-$*$ infrastructures must plan for similar kinds of individual worker downtime. For example, upgrades require taking individual workers offline. Additionally, occasional worker reboots provide real operational benefits by resetting software state [5].

Negotiation: Parties with independent interests use negotiation to coordinate on common objectives and to exchange resources towards mutual benefit. Negotiation mechanisms will be useful when self-$*$ systems in different administrative domains must interact [7]. Such negotiation may also be useful for avoiding decision-making bottlenecks in the management hierarchy—some dynamic and short-term decisions could be negotiated locally by workers in different branches of the hierarchy.

[2] Small-scale tests can also be done with traces applied to either extra resources or simulators.

Fire drills and audits: When correctness and safety are focal points, human organizations use practice and verification. Fire drills consist of simulated or safely recoverable faults being injected into an operational environment, allowing workers to practice response actions. Audits and other self-checking mechanisms examine intermediate and final results, from real operation or fire drills, to identify errors and problems. These concepts will be needed in self-∗ systems to proactively identify problems, exercise recovery mechanisms, and identify corner cases that should be avoided. They will also be needed to keep administrators engaged and up-to-date so that they are prepared if serious problems arise. Experiences with high-reliability computing systems and automation systems (e.g., auto-pilot and anti-lock brakes) confirm the importance of this latter aspect [11].

Training and experience: People's abilities improve as they learn, creating patterns and identifying shortcuts for particular tasks. Because of this, as well as personal comfort, people do not like to have their tasks changed frequently. At the same time, supervisors must reassess worker capabilities over time to utilize their evolving capabilities more fully. To avoid problems in the early stages of the development process, many organizations put new workers through a training program, either offline or online with close oversight, before putting them into the field. Self-∗ components that adapt to offered loads will likely suffer during rapid reassignment and improve with training and experience. In fact, one measure of quality for self-∗ components will be how effectively and quickly they can tune themselves to the characteristics of the workload and environment.

Entrance exams: Many human organizations use entrance exams to characterize a new person's skill set. Such testing is valuable for setting initial expectations and verifying that his abilities are above a minimum threshold. In self-∗ systems, this can act as a form of on-line benchmarking, giving automated task assignment engines information to guide decisions. Like in the real world, the amount of pre-deployment benchmarking will depend in part on how critically help is needed.

Expecting inefficiencies: Human organizations are often mocked for their inefficiencies—perhaps there is a lesson here for self-∗ designers. Scalable systems only work when complexity is managed, and large human organizations manage complexity in part by tolerating inefficiency and not attempting to fully utilize every resource.

3. SELF-* STORAGE

We are exploiting insights from the corporate analogy in the design of self-∗ storage, a self-tuning, self-managing storage system. This section describes our system architecture and discusses the key interfaces and components, identifying the insights we exploit.

3.1 System architecture

The high-level system architecture, shown in Figure 1, consists of three types of components: *supervisors*, *workers*, and *routers*. These three components work together to configure and tune the storage system based on administrator-provided goals.

Supervisors: The supervisors form a management hierarchy. The top of the hierarchy receives high-level goals for data items from the system administrator. These directives are partitioned at each level of the tree structure according to the capabilities of workers

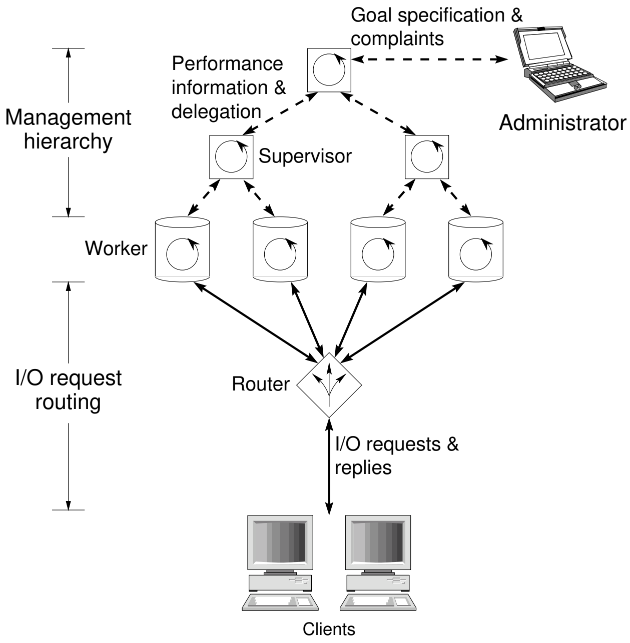

Figure 1: Architecture of self-∗ storage – This figure shows the high-level architecture of self-∗ storage. The top of the diagram is the management hierarchy, concerned with the distribution of goals and the delegation of storage responsibilities from the system administrator down to the individual worker devices. The bottom of the figure depicts the path of I/O requests in the system from clients, through routing nodes, to the workers for service. Note that the management infrastructure is logically independent of the I/O request path.

in that branch. Each supervisor knows the goals and data that were assigned to it as well as how it has partitioned this work to its subordinates (workers or lower-level supervisors).

Workers: The workers, typically small storage arrays, are responsible for storing data and servicing I/O requests for clients. The data that a worker stores and its goals are assigned by that worker's direct supervisor. However, the supervisor does not dictate any configuration parameters for worker devices. Each worker internally refines cache, scheduling, and layout policies based on its characteristics, capabilities, and observed workload.

Routers: Routers are logical entities whose functionality may be implemented in client systems, smart network switches, or worker front ends. They ensure that I/O requests are delivered to the appropriate worker nodes for service. Since the core job of the routers is transmitting requests and replies between workers and clients, they implement distribution decisions. While worker nodes have a level of autonomy from the supervisors, routers do not—supervisors dictate the routing policy to routers. This is necessary since the supervisors determine inter-worker data and workload distribution, and the choice of routing policy greatly affects the workload that is presented to a given worker.

3.2 Administrative interface

Critical to self-$*$ storage being able to self-manage is having an external entity (the system administrator) specify the high-level goals for the system. There seems to be some consensus that system configuration, from the administrator's perspective, is better handled by specifying goals rather than mechanisms [1]. While moving from specifying mechanisms to goals is a step in the right direction, it is unclear whether administrators are properly equipped to specify detailed goals. There is some hope that availability and reliability goals can be derived from various business and insurance costs (related to risk management), but the proper way to set performance targets is a different problem. Workload characterization can provide a first estimate of performance requirements (when replacing an existing system) and can even be used to guide storage system design [2, 3], but further tuning by the administrator will be necessary. Providing an easy to use interface for this refinement is a necessity. We believe that using a system of complaints from the administrator can provide just such an interface.

Complaint-based tuning will use complaints about the performance of specific data items as feedback to revise performance targets and priorities between data items. This allows the administrator to revise the system's goals using a very intuitive interface. Humans are very good at voicing displeasure when something fails to meet their expectations even if they cannot enumerate those expectations. In particular, an administrator (or even a normal user) can usually identify when the system is not performing well enough.

When an administrator voices a complaint about the current service level of the storage system, he provides two key pieces of information. First, the complaint is a statement that current service levels are not sufficient. Second, the data identified in the complaint is the "most noticeable" offender. These two items can be used by the system to guide performance tuning.

The first case to consider for tuning is one in which the system is able to meet its current performance targets (i.e., there are sufficient resources for the demand currently placed on the system). When a complaint is received, we know that the current performance level is inadequate, and since it is already meeting its target, that target is incorrect and should be modified. Complaints could also have a qualitative "strength" to guide the magnitudes of the adjustments, but it may be possible to infer this information from the pattern of complaints. By iteratively receiving complaints and adjusting performance goals, the system can "zero in" on the proper settings.

The second case of interest occurs when the system is unable to meet the current performance targets and a complaint is received. In this case, we know that the current resources are insufficient for the observed workload, and a message to that effect can be sent to the administrator. The real insight in this situation is that the administrator chose to complain about this particular data item when there are likely many that are not meeting their targets. The way to interpret this is that it is more important for the specified data item to meet its target than for other data items to meet theirs, so the system should adjust not the performance targets, but the relative importance of meeting them.

3.3 Supervisor interface

For the system as a whole to meet the externally supplied goals, supervisors must communicate with their subordinates to ensure all of the sub-parts are performing satisfactorily. Supervisor-to-subordinate communication serves two functions. First, it disseminates the "tasks" and goals to lower-level nodes, either to workers or other supervisors. Second, it allows a supervisor to assess the performance of its subordinates to ensure they are meeting their goals.

Along the downward path, the supervisor assigns data and specifies associated goals (e.g., as SLOs in Rome [13]); it does not specify mechanisms. This use of goals allows the higher level supervisor to ignore the details of lower-level mechanisms. Additionally, it allows *intent* to be passed as part of the assignment. By communicating intent down the tree, lower nodes gain the ability to assess their performance relative to goals as they internally optimize. In addition to communicating assignments, supervisors may also transmit potential future assignments. This could be used to evaluate "what if…" scenarios, allowing supervisors to tune based on performance predictions from its workers.

There are three types of information that we desire from the upward communication path. First, lower-level nodes need to provide workload information to their supervisor. This information encapsulates not only workload characteristics, but also how they (possibly a subtree of nodes) are performing relative to the desired goals. Second, some information regarding the capabilities of the subtree should be provided. Due to the workload-dependent nature of such a specification, this is likely only an approximation, but may be helpful to the supervisor node as it attempts to tune. Third, it is desirable for lower-level nodes to be able to provide predictions about potential configurations. If supervisors are able to ask, "How well would you meet your goals if I added workload X?", optimization can likely be sped up considerably. In addition to speeding up the process, poor decisions can be pruned without inflicting a performance penalty on the system via trial-and-error.

3.4 Supervisor internals

The hierarchy of supervisor nodes controls how data is partitioned onto each worker and how the incoming workload is distributed. A supervisor's main objective is to correctly partition data and goals onto its subordinate nodes, such that if its children meet their assigned goals, the goals for the entire subtree managed by the supervisor will be met. Creating this partitioning without wasting many resources is not easy. Prior to partitioning the workload, the supervisor needs to gain some understanding of the capabilities of each of its workers. Much like this interaction in human organizations, the information will be imperfect. For instance, the workers may provide best-case performance numbers to the supervisor. This is of some use, but far from ideal.

One of the major obstacles for the supervisor to overcome is finding a way to evaluate a sufficient portion of the possible configurations. To handle this, the supervisor is likely to use a combination of coarse statistics, simulation, and trial-and-error. The coarse statistics for workload characteristics and worker capabilities may allow large numbers of possibilities to be evaluated quickly, but with a significant margin of error. Prospective partitionings could be further refined by asking workers to simulate them given the workload goals and historical traces collected by workers. Finally, the promising ones from this stage can be implemented, first on a small scale, then applied to larger sets of devices and data.

Over the course of tuning, supervisors will develop profiles of each of its workers. These profiles will contain not only coarse performance metrics, but also information about a worker's ability to internally optimize and accurately predict its performance. Having an accurate profile of the worker nodes will greatly assist tuning. For example, workers that are better able to optimize should be assigned the workloads that have the most potential for optimization, while workers that are ineffective at internally optimizing can be given workloads that appear purely random.

3.5 Worker internals

Workers store and service requests for assigned data in whatever manner they deem most effective. By allowing the workers to independently optimize, they can internally reorganize data layouts [12], adapting caching policies [8, 9], and scheduling requests based on both priorities and device characteristics.

Clearly, if workers are given a level of autonomy, there are certain minimum requirements for these devices (beyond mere persistent storage). First, the worker must be able to handle block allocation internally. The mechanism for handling this can be simple, but it is necessary since the supervisor does not dictate intra-device data placement strategies. Second, the worker must provide a way for the supervisor to assess its performance. The most straightforward method for this is for workers to record a trace of all requests and their service times. Given these traces, or proper statistics, the supervisor can evaluate the effectiveness of its workers.

An additional ability that would be helpful, although not strictly necessary, is for the worker to be able to provide predictions about how well it could achieve its goals given a potential configuration. In this scenario, the supervisor could provide a potential configuration and an associated workload for the worker to analyze via analytic models or internal simulation. While the idea of embedding a simulator in a storage device may sound overly optimistic, one must remember that many already have internal timing models that are used for scheduling.

4. OPEN QUESTIONS

This section discusses some issues that we believe are critical to the success of this architecture for self-$*$ storage.

Supervisor's ability to optimize: The ability of supervisors to successfully coordinate many sub-devices, each of which internally optimize, is critical to the success of this architecture. One of the main properties that make this supervisor/worker organization attractive is that the supervisors abstract away the internals of their workers. However, hiding the details makes optimization more difficult. Worse, workers can adapt to their workload, making their behavior more difficult to predict.

To ensure that supervisors make timely progress, we plan to use a two-phase approach. First, it is important for the system to quickly move away from configurations that are "abnormally bad", and into a state that is "reasonable." In this first phase, we want to ensure that the system is able to get itself out of performance holes, but we are not concerned with optimality. In the second phase, it is not necessary to quickly converge upon an optimal configuration, only that the system continually attempts to improve. We believe that the number of configurations (in practice) that exhibit acceptable performance is likely to be large relative to those that exhibit either abnormally good or bad characteristics. By combining both coarse metrics and simulation (in the first and second phases, respectively), the system should be able to make timely progress.

Goal tradeoffs: While it seems clear that the system should be able to make tradeoffs among high-level goals, it is not obvious how to implement them nor how they should be controlled by the administrator. For instance, the system may have goals for reliability, availability, performance, and power consumption. One approach is to create a strict ordering: reliability, availability, performance, then power down anything that remains. This leads to unsatisfying scenarios. For instance, performance could possibly be greatly improved by accepting a small reduction in availability. In order to recognize and properly handle this, the system must have a continuous model for its goals, not just a black-and-white threshold.

The introduction of interactions between goals complicates the administrator's job because he must understand those interactions as the workload and system characteristics change. Finding a method to simply communicate this information to the administrator is a necessity. Using the system's ability to make predictions may prove helpful. For example, the system can provide evaluations of "what if" scenarios to help the administrator understand how the configuration would change as the workload increases, devices are upgraded, or components fail. Additionally, the use of "safety stops" may be useful in the event the administrator makes a serious tuning mistake. For example, minimum thresholds for the various parameters could serve as hard limits that the system would not cross while it makes internal tradeoffs.

Configuration complexity: Even in the architecture we propose, there are a number of tunable parameters (e.g., per data set goals and their relative importance). Finding a method for the administrator to easily set each of them is a challenging task. While the complaint-based tuning provides a simple interface, it will be tedious to use outside of the occasional performance "tweak."

For managing this complexity, there are two aspects that work in our favor. First, it is likely that most data will fit a generic (though site-specific) template, and only the exceptions would need to be hand-configured. Second, tuning the system can be a gradual process. When first installed, it is likely that the system will be over-provisioned, allowing the system to perform well even if the goals not properly tuned. Only when the workload increases, causing the system to become resource constrained, will the actual performance thresholds become important. When the frequency of performance tweaks reaches an unacceptable level, more resources should be added. This highlights another tradeoff. Additional resources can be used to reduce goal specification effort. By adding more resources, the system can be moved back toward the over-provisioned state, and, due to the self-$*$ properties of the system, administrative effort to configure the new resources is not necessary. Thus, there is a tradeoff between money spent for new resources and money spent on administrative (tuning) effort.

5. SUMMARY

Human organizations offer useful insights for the design of self-$*$ systems. This paper describes the architecture of a self-$*$ storage system that borrows from this analogy. For example, it uses a supervisor/worker hierarchy to distribute work and manage complexity while freeing components to independently optimize based on their individual characteristics and workloads. It also uses complaint-based tuning for performance goal specification.

Acknowledgments

These ideas have benefited from discussions with many people. In particular, we thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support.

6. REFERENCES

[1] G. Alvarez, K. Keeton, A. Merchant, E. Riedel, and J. Wilkes. Storage systems management. *Tutorial presentation at International Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, June 2000). Hewlett-Packard Labs, June 2000.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, **19**(4):483–518. ACM Press, November 2001.

[3] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. *Ergastulum: an approach to solving the workload and device configuration problem*. Technical report HPL–SSP–2001–05. HP Labs, 2001.

[4] Gartner Group. Total cost of storage ownership — a user-oriented approach, February, 2000. Research note, Gartner Group.

[5] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. *International Symposium on Fault-Tolerant Compter Systems* (Pasadena, CA, 27–30 June 1995), pages 381–390. IEEE Computer Society Press, 1995.

[6] IBM Corp. Autonomic computing: IBM's perspective on the state of information technology, October 2001. `http://www.research.ibm.com/autonomic/manifesto/`.

[7] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, **36**(1):41–50. IEEE, January 2003.

[8] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. *Workshop on Input/Output in Parallel and Distributed Systems* (San Jose, CA), pages 57–67. ACM Press, December 1997.

[9] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003). USENIX Association, 2003.

[10] New England Complex Systems Institute. `http://www.necsi.org/`.

[11] I. Peterson. *Fatal defect: chasing killer computer bugs*. Vintage Books, Inc., 1996.

[12] B. Salmon, E. Thereska, C. A. N. Soules, and G. R. Ganger. A two-tiered software architecture for automated tuning of disk layouts. *Algorithms and Architectures for Self-Managing Systems* (San Diego, CA, 11 June 2003), pages 13–18. ACM, 2003.

[13] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *International Workshop on Quality of Service* (Berlin, Germany, 6–8 June 2001). Published as *Lecture Notes in Computer Science*, **2092**:75–91. Springer-Verlag, 2001.