Self-* Storage:
Brick-based storage with automated administration

Gregory R. Ganger, John D. Strunk, Andrew J. Klosterman

August 2003

CMU-CS-03-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This white paper describes a new project exploring the design and implementation of "self- storage systems:" self-organizing, self-configuring, self-tuning, self-healing, self-managing systems of storage bricks. Borrowing organizational ideas from corporate structure and automation technologies from AI and control systems, we hope to dramatically reduce the administrative burden currently faced by data center administrators. Further, compositions of lower cost components can be utilized, with available resources collectively used to achieve high levels of reliability, availability, and performance.*

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. Aspects of the self-* storage project are funded by DARPA, AFOSR, NSF, and the PDL Consortium.

Keywords: Self-management, autonomic computing, storage management, data centers

1 Introduction

As computer complexity has grown and system costs have shrunk, system administration has become a dominant factor in both ownership cost and user dissatisfaction. The research community is quite aware of this problem, and there have been well-publicized calls to action [27, 51]. Storage systems are key parts of the equation for several reasons. First, storage is where the persistent data is kept, making it a critical system component. Second, storage represents 40–60% of hardware costs in modern data centers, and 60–80% of the total cost of ownership [1, 32]. Third, storage administration (including capacity planning, backup, load balancing, etc.) is where much of the administrative effort lies; Gartner [17] and others [24] have estimated the task at one administrator per 1–10 terabytes, which is a scary ratio with multi-petabyte data centers on the horizon.

Many industry efforts are working to simplify "storage management" by developing tools to reduce the effort of managing traditional storage system designs. We believe that this approach is fundamentally limited, much like that of adding security to a finished system rather than integrating it into the system design. We believe that what is needed is a step back and a clean-slate redesign of storage in the data center. Without re-architecting storage systems, administration complexity will continue to dominate administrator workloads and ownership cost.

This white paper overviews current storage administration tasks and outlines a storage architecture that integrates automated management functions to simplify them. We refer to such systems as *self-* storage systems* in an attempt to capture many recent buzzwords in a single meta-buzzword; self-* (pronounced "self-star"[1]) storage systems should be self-configuring, self-organizing, self-tuning, self-healing, self-managing, etc. Ideally, human administrators should have to do nothing more than provide muscle for component addition, guidance on acceptable risk levels (e.g., reliability goals), and current levels of satisfaction.

We think in terms of systems composed of networked "intelligent" *storage bricks* [14, 15, 26], each consisting of CPU(s), RAM, and a number of disks. Although special-purpose hardware may speed up network communication and data encode/decode operations, it is the software functionality and distribution of work that could make such systems easier to administer and competitive in performance and reliability. Designing self-*-ness in from the start allows construction of high-performance, high-reliability storage infrastructures from weaker, less-reliable base units; with a RAID-like argument [37] at a much larger scale, this is the storage analogue of cluster computing's benefits relative to historical supercomputers.

Our self-* storage design meshes insights from AI, corporate theory, and storage systems. It borrows the management hierarchy concept from corporate structure [46]: supervisory oversight without micro-management. Each storage brick (a "worker") tunes and adapts its operation to its observed workload and assigned goals. Data redundancy across and within storage bricks provides fault tolerance and creates opportunities for automated reconfiguration to handle many problems. Out-of-band supervisory processes assign datasets and goals to workers, track their performance and reliability status, and exchange information with human administrators. Dataset assignments and redundancy schemes are dynamically adjusted based on observed and projected performance and reliability.

We refer to self-* collections of storage bricks as *storage constellations*. In exploring self-* storage, we plan to develop and deploy a large-scale ($\approx$1 PB) storage constellation, called Ursa

[1] "Self-*" is a play on the wild-card character, '*', used in many programs.

Major, with capacity provided to research groups (e.g., in data mining and scientific visualization) around Carnegie Mellon who rely on large quantities of storage for their work. We are convinced that such deployment and maintenance is necessary to evaluate self-* storage's ability to simplify administration for system scales and workload mixes that traditionally present difficulties. As well, our use of low-cost hardware and immature software will push the boundaries of fault-tolerance and automated recovery mechanisms, which are critical for storage infrastructures.

The remainder of this white paper is organized in three sections. Section 2 discusses storage administration in general. Section 3 describes our self-* storage architecture. Section 4 overviews our current design and plan for building and deploying the first prototype system (Ursa Minor).

2 Storage administration

The human costs of storage administration are estimated to be 4–8 times that of storage hardware and software costs [1, 24, 32], even with today's expensive, special-purpose storage subsystems. With the industry's push towards using commodity-priced, consumer-quality components, administrative overheads will only worsen unless there are changes in approach. This section discusses storage administration functions and efforts to automate some of them.

2.1 Storage administration functions

"Storage administration" and "storage management" are catch-all terms for a large set of tasks. As a result, a solid definition has not been available, making it a vague target for improvement. To help clarify our goals, this section offers one categorization of storage administration tasks.

Data protection: Perhaps the most important role of a storage infrastructure, and thus task source for storage administrators, is ensuring the continued existence and accessibility of stored data in the face of accidents and component failures. Data protection addresses up to four concerns: user mistakes and client software problems that corrupt data (e.g., deletion followed by regret), failures of a small subset of components (e.g., a disk head crash or server failure), site and large-scale failures (e.g., building collapse), and long-term archiving. All four concerns are handled with redundancy (usually replication), with solutions differing by where replicas are kept and how out-of-date they may get [29]. Administrative tasks include creating replicas, tracking their locations, and accessing them when needed. All four concerns can be addressed by the well-known "tape backup" approach, where replicas are copied onto tape media, though with substantial equipment and/or human effort costs.

Performance tuning: The performance of storage systems always seems to be a concern for administrators. Achieving better performance from existing components can involve a variety of load balancing and device (re-)configuration tasks. First, opportunities for improvement must be identified, requiring information about workloads, hot spots, correlated access patterns, and performance goals. Second, decisions about what to change must be made, which requires some insight into how each choice will affect system performance—such insights are particularly difficult to acquire for device-specific tuning knobs. Third, changes must be implemented, which may require restarting components and/or moving large quantities of data—coordinating any down-time and completing data reorganization efficiently can be significant burdens.

Planning and deployment: Capacity planning, or determining how many and which types of components to purchase, is a key task for storage administrators. It arises at initial deployment, as components fail, and as capacity and performance requirements change over time. Effective capacity planning requires estimated workload information, growth predictions, and rough device models. Beyond any paperwork for purchasing new equipment, acquisitions require hardware and software installation and configuration, followed by data migration. In many existing systems, significant work is involved whenever data must be moved to new devices—the process can be as bad as taking the existing storage off-line, configuring and initializing the new components, and then restoring the data onto the new components.

Monitoring and record-keeping: System administration involves a significant amount of effort just to stay aware of the environment. For example, someone must keep track of the inventory of on-line and spare resources, the configurations of existing components, the datasets (a.k.a. volumes) and their users, the locations and contents of any off-line backup media, etc. As the number of information tidbits grows, so does the need for strict policies on maintaining such records. For on-line components, there is also an active monitoring task—a watchful eye must be kept on the reliability and performance status of system components, to identify potential trouble spots early.

Diagnosis and repair: Despite prophylactic measures, problem situations arise that administrators must handle. Proactive mechanisms, like on-line redundancy, can mask some failures until they can be handled at an administrator's convenience. But, at some point, a human must become aware of the problem and do something about it (e.g., replace failed components). More work-intensive repairs can involve new acquisitions and restoring data from backup media. More complex situations involve diagnosis: for example, an administrator must often deduce sources of problems from vague descriptions like "performance is insufficient" or "some data corruption is occurring."

The breakdown above leaves out generic but time-consuming tasks, such as reporting and training of new administrators. Also, there are certainly other ways to usefully categorize storage administration tasks, such as by the tools used or by whether they are reactive or proactive activities.

2.2 Efforts to simplify administration

The state-of-the-art for many storage administration tasks is sad, historically relying on an unintegrated, ad hoc collection of tools provided with the individual components or created by the data center's administrators. There are, of course, tools for many of the mechanical tasks, such as controlling backup/restore and migrating data while providing online access. Similarly, monitoring and basic diagnosis are aided by tools for polling and visualizing device-collected statistics. A current industry focus [44, 45] is on standardizing access to these statistics, as is done via SNMP [8] in networks. Most storage administrative functions, however, require planning in the context of the data center's workloads and priorities, and much of it currently happens via active human thinking.

Despite recent "calls to arms" [27, 51], automation of management functions has long been a goal of system designers. For example, 20 years before coining the term "Autonomic Computing," IBM was working on the goal of "system-managed storage" [18]. Early efforts focused on simplifying administration by introducing layers of storage abstraction (e.g., SCSI's linear LBN address space) that we now take for granted in block-based storage systems. Prior to these efforts, low-level device details (e.g., media defects and rotational position sensing) were issues of concern for administrators.

More recent efforts have focused on raising the administrative abstraction further. Most notable is HP Labs's vision of attribute-managed storage [21]. Given high-level workload and device model specifications, the HP Labs researchers have been able to create tools that design cost-effective storage systems to meet those goals [2, 6, 7]. This work paves the way for self-* storage systems by tackling one of the most important problems: goal specification. Unfortunately, it still expects unrealistic prescience on the part of system administrators; specifically, it requires accurate models of workloads, performance requirements, and devices. Device models could be provided by manufacturers or automated extraction tools [4, 40, 48], but workload characterization has stumped storage researchers for decades [16]. Expecting system administrators to do it in practice is a show-stopper.[2] Inside the infrastructure, however, such goal specification and usage mechanisms will be critical.

Another approach to automating administration is to simply reduce the need for it. For example, on-line data redundancy can help with many data protection issues. Most storage infrastructures can mask at least some failures with on-line redundancy across devices; commonly referred to today as "RAID" [9, 37], this basic approach has a very long history. Researchers continue to seek cost-effective ways to increase the fault tolerance window, thereby reducing the pressure on backup/restore activities. In addition, the effort involved with backup/restore has been significantly reduced by the increasing prevalence of on-line *snapshot* support in storage systems [11, 25]. Remote replication of storage [13, 38] to a secondary site can reduce the time and effort of low-downtime disaster recovery. On-line archival systems [10, 12, 31] can simplify long-term storage. In other areas, interposed request routing [3] and smart file system switches [30, 53] can address load balancing and other issues by embedding dynamic routing in the system; these are examples of the much-ballyhooed "virtualization" concept, which boils down to adding a level of indirection. Efforts like those mentioned above provide tools for self-* storage systems, but they address only the mechanism portion of the equation; selecting policies and configuration settings remain tasks for the administrator.

There have been numerous projects exploring various aspects of dynamic and adaptive tuning of storage systems. We will not review them here, but building self-* storage will obviously involve borrowing extensively from such prior work.

2.3 Brick-based storage infrastructures

Today's large-scale storage systems are extremely expensive and complex, with special-purpose disk array controllers, Fibre Channel SANs, robotic tape systems, etc. Much lower cost per terabyte could be realized by instead using collections of commodity PC components. As a result, many share our vision of storage infrastructures composed of large numbers of low-cost, interchangeable building blocks, called *storage bricks* [14, 15, 26]. Each storage brick is a networked, rack-mounted server with CPU, RAM, and a number of disk drives. The per-brick CPU and RAM provide necessary resources for caching and data organization functionality. Example brick designs provide 0.5–5 TB of storage with moderate levels of reliability, availability, and performance. Ideally, high levels of these properties can be achieved by intra- and inter-brick redundancy mech-

[2]Note that this does not mean that I/O workload characterization is no longer worth pursuing. It will play a critical role in self-* storage systems, and continuing research (e.g., [49]) will be needed.

anisms and appropriate data and workload distribution.

Achieving this ideal is the challenge. Making matters worse, replacing carefully-engineered (and internally redundant) monolithic systems with collections of smaller, weaker systems usually increases administrative overhead, as the number of components increases and their robustness decreases. Thus, there is a very real possibility that shifting towards low-cost hardware only serves to increase the total cost of ownership by increasing storage administrative loads. Overall, the desire to shift towards less expensive, brick-based storage infrastructures increases the need for self-* storage systems.

3 Self-* storage architecture

Dramatic simplification of storage administration requires that associated functionalities be designed-in from the start and integrated throughout the storage system design. System components must continually collect information about tasks and how well they are being accomplished. Regular re-evaluation of configuration and workload partitioning must occur, all within the context of high-level administrator guidance.

The self-* storage project is designing an architecture from a clean slate to explore such integration. The high-level system architecture, shown in Figure 1, borrows organizational concepts from corporate structure.[3] Briefly, workers are storage bricks that adaptively tune themselves, routers are logical entities that deliver requests to the right workers, and supervisors plan system-wide and orchestrate from out-of-band. This section describes the administration/organization components and the data storage/access components. It concludes with some discussion of how a self-* storage system can simplify storage administration.

3.1 Administration and organization

Several components work together to allow a self-* constellation to organize its components and partition its tasks. We do not believe that complex global goals can be achieved with strictly local decisions; some degree of coordinated decision making is needed. The supervisors, processes playing an organizational role in the infrastructure, form a management hierarchy. They dynamically tune dataset-to-worker assignments, redundancy schemes for given datasets, and router policies. Supervisors also monitor the reliability and performance status of their subordinates and predict future levels of both. The top of the hierarchy interacts with the system administrator, receiving high-level goals for datasets and providing status and procurement requests. Additional services (e.g., event logging and directory), which we refer to as *administrative assistants*, are also needed.

This subsection describes envisioned roles of the administrative interface, the supervisors, and some administrative assistants.

3.1.1 Administrative interface

Although less onerous to manage, a self-* storage system will still require human administrators to provide guidance, approve procurement requests, and physically install and repair equipment.

[3]Implications of the corporate analogy for self-* systems are explored elsewhere [46].

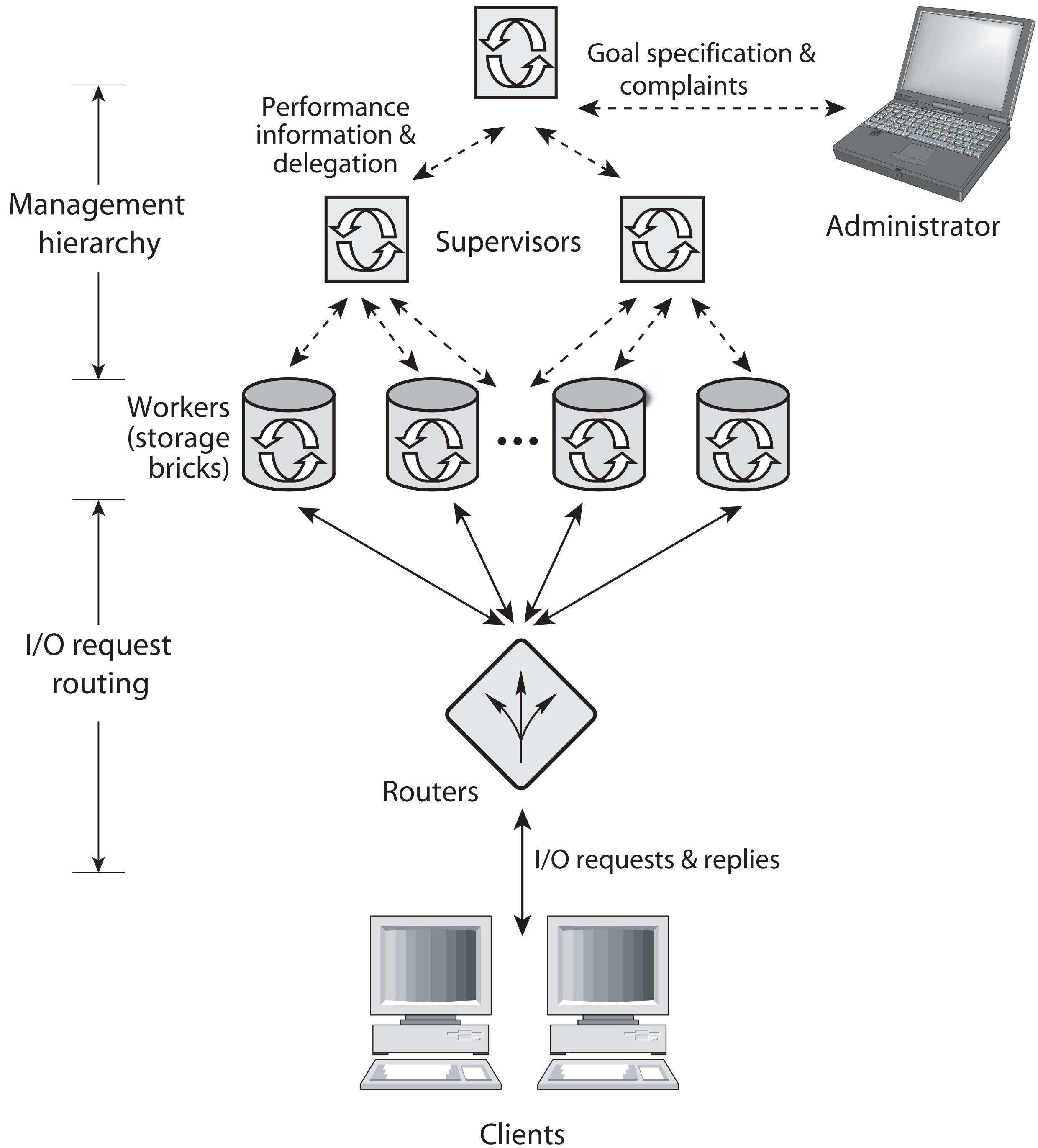

Figure 1: **Architecture of self-* storage.** The top of the diagram is the management hierarchy, concerned with the distribution of goals and the delegation of storage responsibilities from the system administrator down to the individual worker devices. The bottom of the diagram depicts the path of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

Critical to self-* storage's ability to self-manage is having an external entity (the system administrator) specify the high-level goals for the system [21]. Unfortunately, it is unrealistic to expect administrators to be able to provide detailed SLA-style specifications of such goals. There is some hope that availability and reliability goals [28] can be derived from various business continuity and insurance costs (à la risk management). Performance goals, on the other hand, better fit a model of rough initial defaults and in situ refinement based on *complaints* like "X is not fast enough." Such *complaint-based tuning* is intuitive, simple to understand, and should provide the system with sufficient feedback to revise performance targets and priorities for datasets.

When a complaint is registered, the supervisors examine the performance of the flagged dataset(s). If performance meets current goals, then the target values are modified (e.g., towards lower latency or higher throughput). If it does not, then priorities among datasets may be modified, but, also, the administrator must be made aware of the need for additional resources or lower expectations.

The administrative interface must provide information and assistance to the system administrator when problems arise or trade-offs are faced. For example, there is usually a trade-off between performance and reliability. The administrator needs to be made aware of such trade-offs when they complain about performance beyond the point of the current system's ability to tune. In addition to identifying problems, the system needs to help the administrator find solutions. For performance, availability, and reliability goals, the system should tell the administrator how much better it could do with more devices, providing guidance for procurement decisions that involve financial trade-offs.

3.1.2 Supervisors

The hierarchy of supervisor nodes controls how data is partitioned among workers, and how requests are distributed. A supervisor's objective is to partition data and goals among its subordinates (workers or lower-level supervisors) such that, if its children meet their assigned goals, the goals for the entire subtree will be met. Creating this partitioning is not easy. Prior to partitioning the workload, the supervisor needs to gain some understanding of the capabilities of each of its workers. Similar to this interaction in human organizations, the information will be imperfect, resulting in some trial-and-error and observation-based categorization.

Supervisors disseminate "tasks" (i.e., datasets to be stored) and goals (e.g., as SLOs in the Rome specification language [50]) to subordinates and assess their success in meeting their goals. By communicating goals down the tree, a supervisor gives its subordinates the ability to assess their own performance relative to goals as they internally tune. Supervisors need not concern themselves with details of how subordinates meet their goals.

We envision supervisors that start with weighted random distributions of tasks to workers and then refine dynamically. Randomness provides robust load distribution and should avoid most pathological configurations. Refinement guidance seeks to escape from any bad luck and to cope with heterogeneity of dataset goals, worker capabilities, and workload distributions. Approaches based on off-line optimization are not our focus because of the difficulty of characterizing workloads and of modeling adaptive storage components. Instead, answers from off-line optimization will be used as hints, with on-line adaptation of configurations as the final arbiter.

Three main types of information from subordinates are desired by supervisors: per-dataset workload information, worker capabilities, and predictions. The workload can be provided in full, as a trace, or in the form of summary statistics. Capabilities will necessarily be rough estimates, given that they depend on workload characteristics, but predictions can often be more accurate. For example, a worker that can simulate itself (e.g., as most disks can for request scheduling purposes) could answer queries like, "How well would you meet your goals if workload X were added?", by simulating the interleaved traces of X and its recent workload. Such support could speed up optimization and prune poor decisions without inflicting a performance penalty on the system via trial-and-error.

3.1.3 Administrative assistants

A self-* constellation requires internal services, beyond those described above, in order to function—critical, and yet often under-appreciated, these are the administrative assistants of a self-* system. For example, a directory service is needed to help the various components translate component/service names to their locations in the network. Similarly, a "first contact" service is needed to connect new components (who use broadcast or hand-configured values to find the directory service) to appropriate supervisors and/or workers. An event logging/notification service is desirable for localizing the various internal information about what is happening in the system; this information is crucial for discovery and diagnosis of current problems and potential problems. When untrusted clients have direct network access to workers, a security service is needed for authentication and access control.

3.2 Data access and storage

Workers store data and routers ensure that I/O requests are delivered to the appropriate workers for service. Thus, self-* clients interact with a self-* routers to access data.

Ideally, a self-* storage system should export a high-level storage abstraction, such as files, to its clients. It should expose a namespace that is much larger than would be needed by the largest system that would be deployed, dynamically mapping live parts of the namespace to physical storage. In particular, namespace boundaries should not be observed even when a system's capacity limit is reached. Traditional block storage systems, which export a linear space of numbered blocks (0 to $N - 1$ for an N block device), create substantial administrative headaches when capacity growth is required.

We envision two types of self-* clients. *Trusted clients* are considered a part of the system, and may be physically co-located with other components (e.g., router instances). Examples are file servers or databases that use the self-* constellation as back-end storage. Such a file server might export a standard interface (e.g., NFS or CIFS) and translate file server requests to internal self-* activity, allowing unmodified systems to access a self-* constellation. *Untrusted clients* interact directly with self-* workers, via self-* routers, to avoid file server bottlenecks [19]. Such clients must be modified to support the self-* constellation's internal protocols, and workers must verify their access privileges on each request. We expect many environments will contain a mix of both client types.

The remainder of this section discusses routers and workers in more detail.

3.2.1 Routers

Routers deliver client requests to the appropriate workers. Doing so requires metadata for tracking current storage assignments, consistency protocols for accessing redundant data, and choices of where to route particular requests (notably, READs to replicated data). We do not necessarily envision the routing functionality in hardware routers—it could be software running on each client, software running on each worker, or functionality embedded in interposed nodes. There are performance trade-offs among these, but the logical functions being performed remain unchanged.

An important part of a self-* router's job is correctly handling accesses to data stored redundantly across storage nodes. Doing so requires a protocol to maintain data consistency and liveness in the presence of failures and concurrency. Many techniques exist for solving these problems, with varying assumptions about what can fail, how, and the implications for performance and scalability.

Self-* routers play a dynamic load balancing role in that they have flexibility in deciding which servers should handle certain requests. In particular, this decision occurs when new data are created and when READ requests access redundant data. The load balancing opportunities can be further enhanced by opportunistic replication of read-mostly data [30] and dynamic querying of workers in low-latency communication environments.

3.2.2 Workers

Workers service requests for and store assigned data. We expect them to have the computation and memory resources needed to internally adapt to their observed workloads by, for example, reorganizing on-disk placements and specializing cache policies. Workers also handle storage allocation internally, both to decouple external naming from internal placements and to allow support for internal versioning. Workers keep historical versions (e.g., snapshots) of all data to assist with recovery from dataset corruption. Although the self-* storage architecture would work with workers as block stores (like SCSI or IDE/ATA disks), we believe they will work better with a higher-level abstraction (e.g., objects or files), which will provide more information for adaptive specializations.

Workers must provide support for a variety of maintenance functions, including crash recovery, integrity checking, and data migration. We envision workers being able to enumerate the set of data items they store, allowing routing metadata to be verified (e.g., for occasional integrity checks and garbage collection) or reconstructed (if corrupted). In addition, when supervisors decide to move data from one worker to another, it should be a direct transfer (via a worker-to-worker COPY request) rather than passing through an intermediary. Further, maintenance functions such as these should happen with minimal effect on real user workloads, requiring careful scheduling of network usage [5, 33] and disk activity [20, 35].

To simplify supervisors' tasks, workers can provide support for tuning and diagnosis. One useful tool will be for each worker to maintain a trace of all requests and their service times. Given this trace, or the right summary statistics, a supervisor can evaluate the loads and effectiveness of its workers. Logs of any internal fault events should also be exposed. Workers could also support a query interface for predicting performance in the face of more or less work.

3.3 Simpler storage administration

Since a primary goal of self-* storage is to simplify storage administration, it is worthwhile to identify ways in which our architecture does so. The remainder of this section discusses specific ways in which tasks in the five categories of storage administration are automated and simplified.

Data protection: Two concerns, user mistakes and component failures, are addressed directly by internal versioning and cross-worker redundancy; a self-* storage system should allow users to access historical versions directly (to recover from mistakes), and should automatically reconstruct the contents of any failed worker. System administrators should only have to deal with replacing failed physical components at times of their convenience. The other two concerns, disaster tolerance and archiving, involve a self-* administrative assistant and some mechanism outside of a single self-* constellation. For example, a self-* backup service could periodically copy snapshots to a remote replica or archival system. Alternately, a self-* mirroring service could use the event logging/notification service to identify new data and asynchronously mirror it to a remote site.

Performance tuning: A self-* constellation should do all performance tuning itself, including knob setting, load balancing, data migration, etc. An administrator's only required involvement should be in providing guidance on performance targets not being met. The administrative interface should provide the administrator with as much detail about internals as is desired, which builds trust, but not require manual tuning of any kind.

Planning and deployment: A self-* administrative interface should also help system administrators decide when to acquire new components. To do so, it must provide information about the trade-offs faced, such as "The system could meet your specified goals with X more workers or Y more network bandwidth." The initialization and configuration aspects of adding components should be completely automated. Plugging in and turning on a worker should be sufficient for it to be automatically integrated into a self-* constellation.

Monitoring and record-keeping: A self-* storage system will track and record everything it knows about, which includes the datasets, the components, their status, and their connections. A reasonable system would retain a healthy history of this information for future reference.

Diagnosis and repair: Ideally, a self-* storage system will isolate and re-configure around most problems, perhaps leaving broken components in place to be attended to later. A system administrator will need to physically remove broken components, and perhaps replace them (à la "planning and deployment" above). The self-* system can help in two ways: by giving simple and clear instructions about how to effect a repair, and by completing internal reconfiguration before asking for action (whenever possible). The latter can reduce occurrences of mistake-induced multiple faults, such as causing data/access loss by simply removing the wrong disk in a disk array that already has one broken. Such problems can also be reduced by having worker storage be self-labelling, such that reinstalling components allows them to be used immediately with their current contents.

Overall, a self-* storage system should make it possible for data centers to scale to multiple petabytes without ridiculous increases in administrator head-counts. System administrators will still play an invaluable role in providing goals for their self-* constellation, determining when and what to purchase, physically installing and removing equipment, and maintaining the physical environment. But, many complex tasks will be offloaded and automated.

4 Ursa Minor: prototype #1

We are convinced that one must build and deploy operational systems in order to effectively explore how these ideas simplify administration. We plan to do so, in several iterations of scale, inviting real users from the Carnegie Mellon research community (e.g., those pursuing data mining, astronomy, and scientific visualization). The users will benefit from not being forced to administer or finance large-scale storage, two stumbling blocks for such researchers today. We benefit from the anecdotal experiences—without trying our self-* storage ideas in a large-scale infrastructure with real users, we will remain unable to test them on difficulties faced in real data centers.

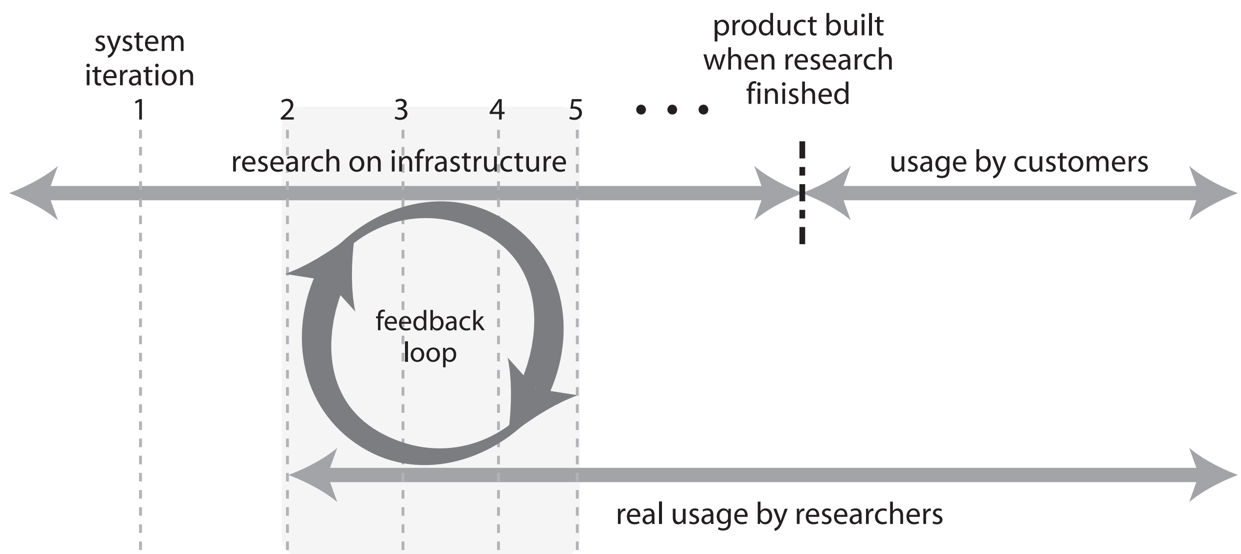

Figure 2: **Deployed infrastructure enhances both collections of researchers.** Experiences from real usage enhances infrastructure research, while availability of infrastructure enhances information processing research. Of course, the infrastructure research must be beyond initial stages in order to provide viable service to its users, but can be iterated and refined over time.

The first prototype, named *Ursa Minor* [4], will be approximately 15 TB spread over 45 small-scale storage bricks. The main goal of this first prototype is rapid (internal) deployment to learn lessons for the design of a second, larger-scale instantiation of self-* storage. *Ursa Major*, the revised system scaled at 100s of terabytes, will be the first deployed for other researchers to use.

4.1 Design of Ursa Minor

From the perspective of their users, our early self-* constellations will look like really big, really fast, really reliable file servers (initially NFS version 3). The decision to hide behind a standard file server interface was made to reduce software version complexities and user-visible changes—user machines can be unmodified, and all bug fixes and experiments can happen transparently behind a standard file server interface. The resulting architecture is illustrated in Figure 3. Each

[4] Ursa Minor is the constellation known as the “Little Dipper” or “Little Bear”. Also, it contains Polaris (the North Star), which provides guidance to travelers even as this first prototype will provide guidance for the self-* storage project.

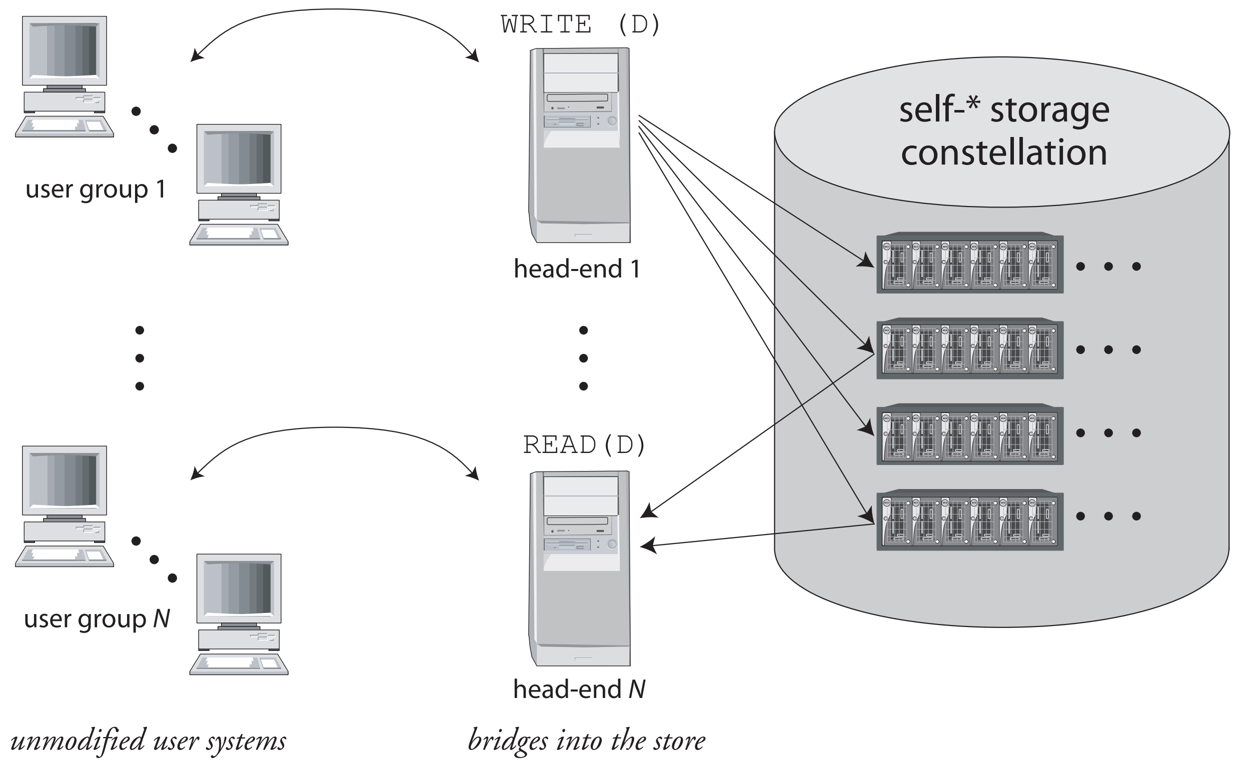

Figure 3: **"Head-end" servers bridge external clients into the self-* storage constellation.**

user community will mount a single large file system, as though they were the only users, by interacting with a specific (virtual) *head-end system*. As trusted clients, head-end systems act as bridges into the self-* storage infrastructure, translating NFS commands into the infrastructure's internal protocols.

Internally, Ursa Minor will be an object[5] store, in the "object-based storage" mold [19, 36]. Each NFS file and directory will be stored as a *self-* object*, identified by a unique system-wide identifier (an object ID, or OID). Each self-* object's contents (data and attributes) will be encoded into some number of *worker objects* (e.g., replicas or erasure code chunks), each stored as an object on a distinct worker. Each worker knows which worker objects it has and supports access to them by their OIDs. A metadata service will translate OIDs and offsets to sets of workers holding portions of desired objects.

Several simplifications fall out of an architecture based on explicit head-ends for different mount points.[6] First, no sharing of NFS structures across machines is necessary; a fail-over strategy, which we will base on a simple watchdog/start-new approach and NFS's statelessness, is needed, yet all active processing is centralized. Second, the metadata service can be partitioned

[5]A storage object is essentially a generic file in a flat namespace. It has device-managed attributes (e.g., size), opaque attributes (for use by higher-level software), and data. Object-based storage systems are expected by many to scale more effectively and to be more easily managed than block-based storage systems.

[6]Note that this architectural short-cut is not our desired end-state. We expect that head-ends will present bottlenecks that are difficult to address (much like current file servers). Also, when we seek to add stateful file system protocols, such as AFS and NFSv4, the simple fail-over feature will be lost.

among head-ends by partitioning the OID namespace; this partitioning simultaneously enhances locality and eliminates the need for decentralized management of complex metadata structures like B-trees. Third, head-ends can play the role of the routers in our self-* storage architecture; they can do dynamic request routing for their requests and execute consistency protocols.

4.2 Building blocks

CMU's Parallel Data Lab (PDL) has been developing relevant technologies for several years, which allows us to put Ursa Minor together relatively quickly. Our focus will be on implementing the data protection aspects, embedding instrumentation, and enabling experimentation with performance and diagnosis. The remainder of this section briefly discusses our plan for incorporating those pieces to quickly realize the main components of Ursa Minor:

Worker: A worker is responsible for storing objects and their historical versions, for keeping a trace of requests, for tuning itself to observed usage patterns, and for efficiently supporting maintenance functions like integrity checks and data migration. For Ursa Minor, the most important aspects will be versioning and request tracing, as these affect correctness and the ability to diagnose. The self-securing storage project [47] has implemented an object store (called *S4*) that provides exactly these functions: it keeps a trace of all requests, and its internal file system (called *CVFS*) can efficiently keep every version of every object [43]. We expect to be able to adapt S4 for Ursa Minor with minimal effort.

Disk bandwidth for maintenance functions will be provided by *freeblock scheduling* [35], which we have implemented [34] and refined into a working system. For most access patterns, it gives background disk maintenance activities 15–40% of the disk head's time with near-zero impact on foreground activity (i.e., real user requests). Intra-worker tuning mechanisms, such as on-disk data reorganization [39], will be an ongoing research and implementation activity.

Router/Head-End: The head-ends are central points of contact for particular NFS trees. For its trees, a head-end translates NFS calls and provides the router functionality for worker interactions. A local instance of the metadata service provides functions for determining how self-* objects are encoded and determining which workers store which worker objects. This same instance handles location updates due to object creation, migration, or reconstruction; supervisors tell it when to do so. To simplify crash recovery, such updates will be via transactions implemented using a dedicated write-ahead log.

Substantial inter-worker redundancy will be used, with each self-* object translating into several worker objects—the common case will be one replica and several code shares (m-of-n erasure coding can space-efficiently tolerate $n - m$ worker failures). The PASIS project [52] has developed a flexible client library with support for several encoding schemes, integrity check mechanisms, and consistency protocols. The PASIS consistency protocols [22, 23] exploit server-side versioning (as implemented by S4) to efficiently tolerate large numbers of failures, making it a good match for our early prototype and its "inexperienced" workers.

The metadata structures used by head-ends can differentiate between object versions in time; older versions may map to different locations and a different encoding scheme. This will be used for a form of on-line back-up: historical snapshots will be written back into the system with a much more failure-tolerant parameterization of the erasure codes. Occasional snapshots may also be written over the network to "disaster recovery partners."

Supervisors and administrative interface: Supervisors are meant to automate administrative decisions and provide a useful interface for human administrators. Both will be prototyped and explored in Ursa Minor, but neither is critical to most of the lessons we hope to learn from it. Exceptions are system startup and shutdown, worker discovery and integration, and collection/organization of event log and request trace information. The logs and traces, in particular, will be crucial to diagnosis and efforts to understand how well things work in different circumstances. For Ursa Major, of course, the supervisors and administrative interface must evolve towards the self-* storage ideal.

The event logging/notification service will trigger and inform internal maintenance and diagnosis agents. Workers will have the ability to accept requests to log events, such as "any changes to object OID," "capacity exceeds watermark Y," or worker-specific (and, thus, externally opaque) changes to internal settings. Interested self-* components can subscribe to events from the event logging service as a form of notification trigger [41][42, p. 636].

The supervisor processes can run on selected workers or dedicated nodes. Although they centralize decision-making logic, supervisors are not central points of failure in the system, from an availability standpoint. Routers and workers continue to function in the absence of supervisors. Further, supervisor processes store their persistent data structures as self-* objects on workers via the normal mechanisms; replacement processes can be restarted on some other node once the original is determined to be off-line.

Testing and the NFS Tee: In addition to benchmarks and test scripts, we plan to test Ursa Minor under our group's live storage workload; the PDL has about 3 TB of on-line storage for traces, prototype software, result files, etc. Not immediately prepared to completely trust our data to it, we will instead duplicate every NFS request to both our real (traditional file server) infrastructure and the Ursa Minor prototype, comparing every response to identify bugs and tune performance. The request duplication mechanism, which we call an *NFS Tee*, will also be used for experimenting with new versions of the system in the live environment. As illustrated in Figure 4, an NFS Tee functions by interposing between clients and servers, duplicating the workload to the currently operational version of the system and an experimental version implemented on a distinct set of storage bricks. Occasionally, when a new version of the system is ready, data will be fully migrated to it, and the new version will become the current version. Such live testing will be used both for verifying that it is safe to change to the new version and to conduct experiments regarding the relative merits of different designs.

5 Summary

Self-* storage has the potential to dramatically reduce the human effort required for large-scale storage systems, which is critical as we move towards multi-petabyte data centers. Ursa Minor will be the first instance of the self-* storage architecture, and will help us move toward truly self-* storage infrastructures that organize, configure, tune, diagnose, and repair themselves.

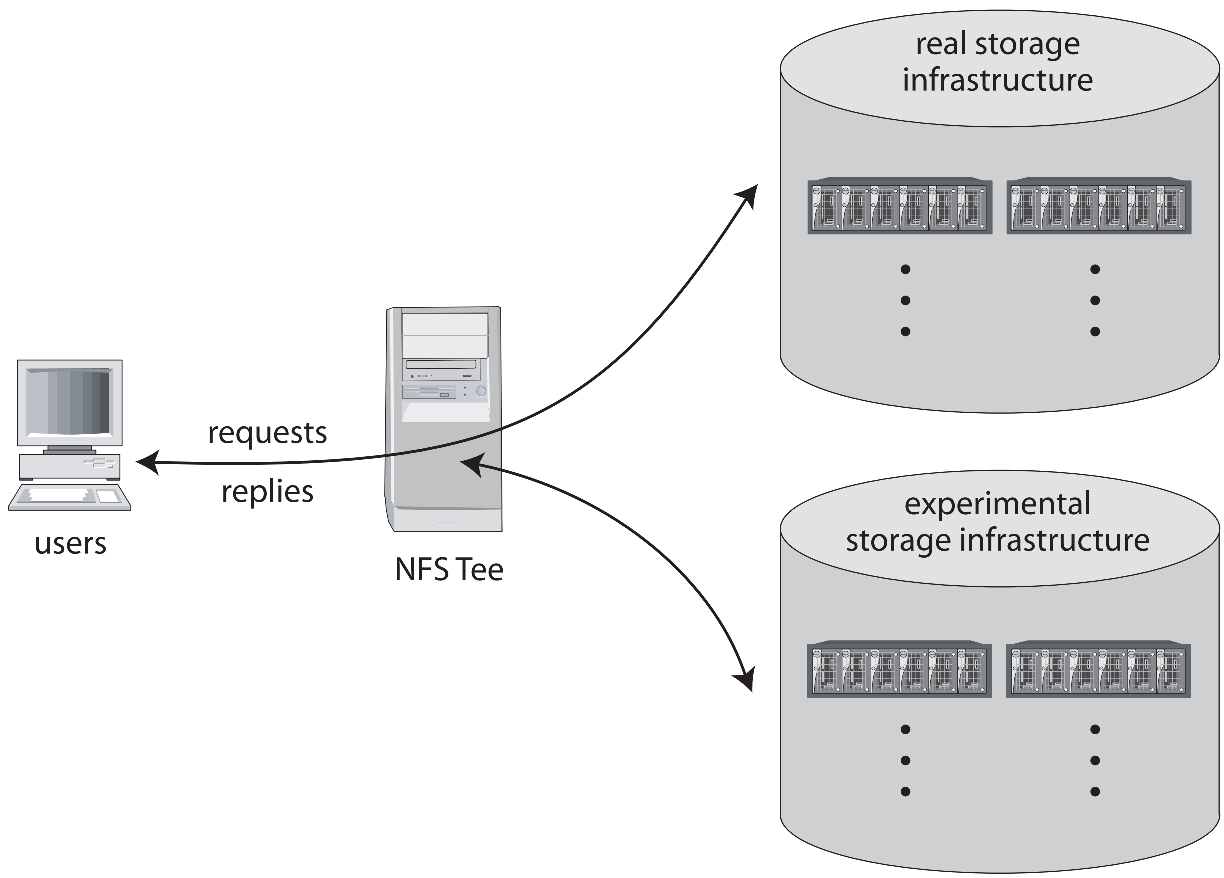

Figure 4: **NFS Tees allow experimental versions of infrastructure to be tested on live workload transparently.**

Acknowledgements

These ideas have benefited from many discussions with luminaries in the field of storage systems. 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. Aspects of the self-* storage project are funded by DARPA, AFOSR, NSF, and the PDL Consortium.

References

[1] N. Allen. Don't waste your storage dollars: what you need to know, March, 2001. Research note, Gartner Group.

[2] Guillermo A. Alvarez, John Wilkes, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, and Alistair Veitch. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, **19**(4):483–518. ACM, November 2001.

[3] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems*, **20**(1):25–48. ACM Press, February 2002.

[4] Eric Anderson. *Simple table-based modeling of storage devices*. SSP Technical Report HPL–SSP–2001–4. HP Laboratories, July 2001.

[5] Eric Anderson, Joe Hall, Jason Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. *International Workshop on Algorithm Engineering* (Arhus, Denmark, 28–31 August 2001). Published as *Lecture Notes in Computer Science*, **2141**:145–158. Springer-Verlag, 2001.

[6] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 175–188. USENIX Association, 2002.

[7] Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez, and John Wilkes. Selecting RAID levels for disk arrays. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 189–202. USENIX Association, 2002.

[8] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *A simple network management protocol*. RFC–1067. Network Working Group, August 1998.

[9] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.

[10] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival Intermemory. *ACM Conference on Digital Libraries* (Berkeley, CA, 11–14 August 1999), pages 28–37. ACM, 1999.

[11] EMC Corp. EMC TimeFinder product description guide, 1998. http://www.emc.com/products/product_pdfs/pdg- /timefinder_pdg.pdf.

[12] EMC Corp. EMC Centera: content addressed storage system, August, 2003. http://www.emc.com/products/systems/centera.jsp?openfolder=-platform.

[13] EMC Corp. SRDF with Celerra file server, September, 2000. http://www.emc.com/pdf/products/celerra_file_server/09072000_cfs_srdf_wp.pdf.

[14] EqualLogic Inc. PeerStorage Overview, 2003. http://www.equallogic.com/pages/products_technology.htm.

[15] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: enterprise storage systems on a shoestring. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 133–138. USENIX Association, 2003.

[16] Gregory R. Ganger. Generating representative synthetic workloads: an unsolved problem. *International Conference on Management and Performance Evaluation of Computer Systems* (Nashville, TN), pages 1263–1269, 1995.

[17] Gartner Group. Total Cost of Storage Ownership — A User-oriented Approach, February, 2000. Research note, Gartner Group.

[18] J. P. Gelb. System–managed storage. *IBM Systems Journal*, **28**(1):77–103. IBM, 1989.

[19] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[20] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 201–212. USENIX Association, 1995.

[21] Richard Golding, Elizabeth Shriver, Tim Sullivan, and John Wilkes. Attribute-managed storage. *Workshop on Modeling and Specification of I/O* (San Antonio, TX), 26 October 1995.

[22] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Efficient consistency for erasure-coded data via versioning servers*. Technical Report CMU–CS–03–127. Carnegie Mellon University, March 2003.

[23] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Decentralized storage consistency via versioning servers*. Technical Report CMU–CS–02–180. Carnegie Mellon University, September 2002.

[24] Jim Gray. A conversation with Jim Gray. *ACM Queue*, **1**(4). ACM, June 2003.

[25] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[26] IBM Almaden Research Center. Collective Intelligent Bricks, August, 2003. http://www.almaden.ibm.com/StorageSystems/-autonomic_storage/CIB/index.shtml.

[27] International Business Machines Corp. Autonomic Computing: IBM's Perspective on the State of Information Technology, October 2001. http://www.research.ibm.com/autonomic/manifesto/.

[28] Kimberly Keeton and John Wilkes. Automating data dependability. *ACM-SIGOPS European Workshop* (Saint-Emilion, France, September 2002), pages 93–100. ACM, 2002.

[29] Kimberly Keeton and John Wilkes. Automatic design of dependable data storage systems. *Algorithms and Architectures for Self-Managing Systems* (San Diego, CA, 11–11 June 2003), pages 7–12. Storage Systems Department Hewlett Packard Laboratories, 2003.

[30] Andrew J. Klosterman and Gregory Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[31] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaten, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):190–201, 2000.

[32] E. Lamb. Hardware spending sputters. *Red Herring*, pages 32–33, June, 2001.

[33] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: online data migration with performance guarantees. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 219–230. USENIX Association, 2002.

[34] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[35] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[36] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[37] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[38] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 117–129. USENIX Association, 2002.

[39] Brandon Salmon, Eno Thereska, Craig A. N. Soules, and Gregory 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.

[40] Jiri Schindler and Gregory R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[41] SNMP Working Group. *A convention for defining traps for use with the SNMP*. RFC–1215. March 1991.

[42] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[43] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Greg Ganger. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–57. USENIX Association, 2003.

[44] Storage Networking Industry Assoc. "Bluefin" a common interface for SAN management, 13 August, 2002. http://www.snia.org/.

[45] Storage Networking Industry Assoc. SMI Specification version 1.0, 2003. http://www.snia.org/.

[46] John D. Strunk and Gregory R. Ganger. A human organization analogy for self-* systems. *Algorithms and Architectures for Self-Managing Systems* (San Diego, CA, 11 June 2003), pages 1–6. ACM, 2003.

[47] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[48] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Cincinnati, OH, 15–18 August 2001), pages 183–192. IEEE, 2001.

[49] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the spatio-temporal behavior of real traffic data. *IFIP WG 7.3 Symposium on Computer Performance* (Rome, Italy, 23–27 September 2002). Published as *Performance Evaluation*, **49**(1–4):147–163, 2002.

[50] John 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.

[51] John Wilkes. Data services – from data to containers, March 31 – April 2, 2003. Keynote address at File and Storage Technologies Conference (FAST'03).

[52] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.

[53] Z-force, Inc. The Z-force File Switch, August, 2003. http://www.zforce.com/.