Network-Embedded Programmable Storage and Its Applications

Sumeet Sobti* Junwen Lai* Yilei Shao* Nitin Garg* Chi Zhang* Ming Zhang*
Fengzhou Zheng* Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

We consider the utility of two key properties of network-embedded storage: programmability and network-awareness. We describe two extensive applications, whose performance and functionalities are significantly enhanced through innovative combination of the two properties. One is an incremental file-transfer system tailor-made for low-bandwidth conditions. The other is a "customizable" distributed file system that can assume very different personalities in different topological and workload environments. The applications show how both properties are necessary to exploit the full potential of network-embedded storage. We also discuss the requirements of a general infrastructure to support easy and effective access to network-embedded storage, and describe a prototype implementation of such an infrastructure.

1 Introduction

For wide-area distributed services, network-embedded storage offers optimization opportunities that are not available when storage resides only at the edges of the network. A prime example of this is content-distribution networks, such as Akamai, which place storage servers at strategic locations inside the network and direct client requests to servers that are "close" to them, thus achieving reduced access latency for the clients and better load balance at the servers.

Given the desirability of network-embedded storage, a natural question to ask is this: What is a good "access model" for network-embedded storage that allows services to realize its full potential? By access model, we mean mechanisms through which diverse services can use the network-embedded storage resources to satisfy their diverse needs.

One simple access model is what can be referred to as the *fixed-interface* model. In this model, each embedded storage element exports a fixed set of high-level operations (such as caching operations). Service-specific code is executed only at edge-nodes. This code manufactures service-specific messages and sends them into the network to manipulate the embedded storage elements through the fixed interface. An example of this model is the Internet Backplane Protocol (IBP) proposed in the "Logistical Networking" approach [8].

Although the fixed-interface model does benefit a certain class of services, it has two main limitations. First, it does not have sufficient flexibility. Due to the extremely diverse needs of distributed services, it may be difficult to arrive at an interface that caters well to all present and future services. Second, the restriction that service-specific code executes only at the edges of the network, and not at the embedded storage elements, imposes a severe limitation, both on the functionalities provided by the services and the optimization opportunities available to them. For example, for application code executing at the edges, it is often difficult to gather information about changes in the load and network conditions around an embedded storage element, and then to respond to such changes in a timely fashion.

These limitations point to the need for the following properties. (1) *Programmability:* the services should be able to execute service-specific code of some form at the embedded storage elements. (2) *Network-awareness:* the code executing at these elements should be able to use dynamic information about the resources at and around them. We do not claim that any of these properties is novel by itself. We, however, do believe that it is the combination of the two that is necessary to realize the full potential of embedded storage.

To support this hypothesis, this paper presents qualitative and quantitative evidence in the form of two applications of network-embedded storage. One is an incremental file-transfer service tailor-made for low-bandwidth conditions (Section 2). The other is a "customizable" distributed file system that can assume very different personalities in different topological and workload environments (Section 3). In these applications, we explicitly point out how the absence of any one of the two properties would significantly limit their power, both in terms of functionality and performance. These applications also show

*Department of Computer Science, Princeton University, Princeton, NJ 08544, USA. Email: {sobti, lai, yshao, nitin, chizhang, mzhang, zheng, rywang}@cs.princeton.edu.

†Department of Computer Science, Yale University, New Haven, CT 06520, USA. Email: arvind@cs.yale.edu.

Krishnamurthy is supported by NSF grants CCR-9985304, ANI-0207399, and CCR-0209122. Wang is supported by NSF grants CCR-9984790 and CCR-0313089.

that the combination of programmability and network-awareness is useful in a diverse set of environments, including both local and wide area networks. A general theme of our work is that in any system configuration or service, if a storage element is in a position to exploit its location advantage intelligently, it should be programmed to do so.

Next in this paper, we consider the question of real-world deployment of services that intend to use network-embedded storage. To launch such a service today, a service provider is typically required to reach agreements with data centers to acquire the needed storage and physical space. This is often an inefficient, time-consuming and costly process, which imposes a significant barrier-to-entry for smaller service providers, and hinders short-term experimentations. A different alternative is to invest effort in a shared, general-purpose infrastructure specifically targetted toward reducing the effort and overhead associated with deploying and customizing services.

We discuss the requirements of a general infrastructure to support easy and effective access to programmable network-embedded storage in Section 4, and describe a prototype implementation of such an infrastructure in Section 5. We refer to such an infrastructure as a *Prognos* (PROGrammable Network Of Storage), and to each embedded storage element in it as a *Stone* (STOrage Network Element).

The resource platform for a Prognos can be either commercially owned, or collaboratively supported as in the PlanetLab project [24] (www.planet-lab.org). As long as the Stones have access to network information, the making of the Stones and the links among them can be quite flexible. One possibility is to construct a Prognos on top of an overlay network [5]. The overlay links used should approximate the underlying physical topology, and the Stones can simply be general-purpose computers. The other potentially more efficient possibility is to co-locate a Stone with a router and the links among the Stones would largely be physical. An extreme form of this co-location is to couple a router and a Stone in the same physical packaging.

We refer to the systems-support module of a Prognos as SOS (Stone Operating System). SOS is responsible for managing the physical resources at the participating Stones, and for allowing services to inject service-specific code into the Stones in a secure fashion. A PlanetLab-like platform, for example, can be turned into a Prognos by loading the participating machines (also referred to as Stones) with the SOS module. We believe that such a collaboratively-supported Prognos can serve as an effective research tool to enable innovators to quickly deploy, experiment with, and tear-down new services.

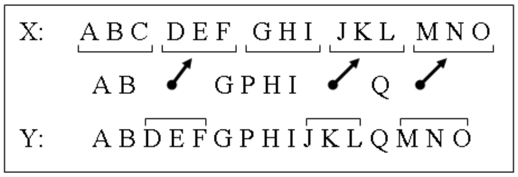

Figure 1: A simple rsync example.

2 Incremental File Transfer

We now describe a service intended to facilitate transfer of incrementally changing, large files. An example usage scenario of this service is one where a producer periodically releases new versions of the Linux kernel file, and multiple consumers update their versions at different times.

The basic idea is to use network-embedded storage elements (or Stones) to optimize these file transfers. As data flows through a sequence of Stones during a file transfer, there is an obvious caching opportunity to benefit subsequent transfers. If, however, the Stones are capable of executing complex service-specific code, more sophisticated optimizations become possible. Our service, which we call "Prognos-based rsync" (or Prsync), programs the Stones to use the `rsync` protocol to propagate files.

2.1 The `rsync` Protocol

The rsync protocol [31] (rsync.samba.org) is a tool for updating an old version of a file with a remotely-located new version. The protocol seeks to reduce network usage by not transferring those portions of the new version that are already present in the old version. A checksum-search algorithm is used to identify such portions when the two versions are not located on the same machine.

As a simple example, suppose that nodes X and Y have two versions of a file with contents shown in the top and bottom rows of Figure 1, and X wants to get Y's version. X first partitions its version into fixed size blocks and sends the checksums of those blocks to Y. In the example shown, X sends five checksums to Y. Using the checksums, Y is able to identify portions that are common between the two versions. Y then sends to X a description of its version referencing the blocks at X wherever possible. The middle row of letters shows the description Y sends to X. X is then able to reconstruct Y's version from this description. If the two versions share several blocks, then there is significant saving in the number of bytes transferred.

2.2 Prsync

We examine four aspects of Prsync relating to the programmability and network-awareness of the Stones. First, we show how programmability of Stones enables rapid deployment of Prsync-like services, even when one does not have full cooperation of edge machines. Second, we describe how Stones can themselves use pair-wise rsync exchanges to improve end-to-end performance. Third, we describe how Prsync adapts to its environment by exploiting the network-awareness of Stones. Fourth, we describe how network information can be combined with service-specific state in a service-specific manner to achieve good performance.

2.2.1 Interaction with Legacy Protocols

Consider a scenario where a producer and a consumer want to engage in a file update, but they lack the ability to participate in rsync exchanges. Assume that the Stones have been programmed to cache files, execute checksum-search algorithms, and participate in the Prsync protocol. The system can still be used to transfer files efficiently. The file is first copied from the producer to a nearby Stone using a legacy protocol. The file is then efficiently propagated using Prsync to a Stone that is located close to the consumer. As the last step, the file is copied from this Stone to the consumer using a legacy protocol. This is an example of an end-to-end legacy protocol that benefits from programmable network-embedded storage.

2.2.2 Hop-by-Hop Interaction

In the above scenario, the Prsync protocol is executed between two Stones that are potentially separated by a weak wide-area connection. The performance could be further improved if we were to enlist intermediate Stones to decompose a long-distance rsync into a sequence of short-distance hop-by-hop rsyncs. Here, the performance improvement can come from a combination of two factors. First, intermediate Stones may already have a version that is very close to the fresh version being propagated. In such cases, fewer bytes will have to be transferred along some portions of the path. Second, after a sequence of hop-by-hop rsync exchanges, all the intermediate Stones also end up receiving the fresh version, and therefore, they can satisfy future requests without requiring end-to-end interactions. The hop-by-hop protocol demonstrates that simple caching in particular, or any hardwired storage interface in general, on intermediate Stones is not sufficient—instead, the programmability of Stones is needed to allow them to participate in a sophisticated protocol.

2.2.3 Adapting to Changing Environments

The rsync program employs a computationally expensive checksum and compression algorithm. Its use may in fact be counterproductive in cases of abundant link bandwidth, drastic file content changes, or high CPU load on participating nodes. In order for Prsync to adapt to these environmental factors in a timely fashion, the programmability and the network-awareness properties of Stones become indispensable. When an upstream node X starts to send fresh data to a downstream node Y, the two nodes begin with the checksum-based rsync algorithm. Node X monitors two quantities dynamically: (1) the ratio (r) between the number of bytes that have been actually transferred and the size of the content that has been synchronized, and (2) the physical bandwidth achieved (B). If r exceeds a threshold, which in turn is a predetermined function of B (implemented as an empirical table lookup), then the communicating nodes would abandon the checksum-based rsync, and revert to simply transmitting the literal bytes of the fresh file. Note that such adaptive optimizations need to be performed on a hop-by-hop basis within the network—they are difficult, if not impossible, to replicate at the edge. An additional optimization to further reduce rsync overhead is to compute the per-block checksums off-line, and store them along with the file in the Stone's persistent store.

2.2.4 Selecting Propagation Paths

In scenarios where there exists path diversity and pairs of Stones are connected by multiple paths (as in overlay networks), Prsync can select propagation paths for hop-by-hop synchronization based on application-specific metrics. We have experimented with two specific methods of doing this. In the *tree-based* method, an overlay tree spanning all the Stones is constructed. The tree is constructed using a minimum-spanning tree algorithm on a graph where the nodes are Stones and the edges are weighted with the inverse of pair-wise bandwidth. The tree construction uses heuristics for constraining the node degree and diameter of the resulting tree. The resulting tree thus contains high bandwidth paths between all pairs of Stones, and only these paths are used for hop-by-hop rsync exchanges. The *mesh-based* method maintains an overlay graph in which each Stone is adjacent to a certain number of other Stones to which it has high-bandwidth links. When selecting a path between a pair of Stones, all paths in this overlay graph are considered. Note that the time taken for a pair-wise rsync exchange is determined by the link bandwidth and the difference between the file versions at the two Stones. Prsync can monitor pair-wise bandwidths, and also maintain estimates of the differences between the file versions at different stones. By using these estimates, a *best* path (i.e., one for which the expected time for hop-by-hop propagation of data is minimized) can be selected in the mesh. This is an instance where information about the network characteristics is combined with service-specific state in a service-specific manner to improve performance. It would be difficult to achieve such optimizations without both programmability and network-awareness of Stones.

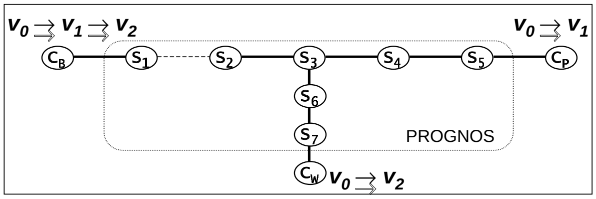

Figure 2: The topology of the Prsync testbed.

Requester	Versions	e-to-e copy (s)	e-to-e rsync (s)	h-by-h rsync (s)
C_P	$V_0 \rightarrow V_1$	97.3	21.0	—
C_W	$V_0 \rightarrow V_2$	97.8	21.5	9.6

Table 1: Prsync performance.

2.3 Prsync Experimental Results

We describe Prsync experiments from two platforms. One platform uses a set of machines in our laboratory that can be operated in a controlled environment. The other consists of a set of PlanetLab machines distributed across the wide-area.

Figure 2 shows the topology of the network constructed in our laboratory. Each node has Dual Intel Pentium III processor, 1GB PC133 ECC SDRAM and a 60GB Maxtor 96147U8 disk. Nodes C_B, C_P, and C_W are considered “edge” machines. The remaining machines make up a Prognos core. C_B serves as the producer of the data. C_P and C_W are requesters.

In the following experiments, we synchronize Linux kernel `tar` files. When we refer to file versions V_0, V_1, and V_2 below, they correspond to “linux.2.0.20.tar”, “linux.2.0.28.tar”, and “linux.2.0.29.tar” respectively. Each of these files is about 25 MB in size. We show results of four experiments, each of which demonstrates one of the aspects detailed in Section 2.2.

The first experiment demonstrates the ability of Prognos to overcome a legacy protocol. The results are summarized in the first row of Table 1. Initially, C_P has version V_0, C_B has V_1, and no other machine has any version of the file. There is a weak link of 2.5 Mbps between S_1 and S_2; all remaining links are dedicated (separate) 100 Mbps. Now, C_P desires to upgrade its file to V_1 and it has several options. It could use an existing legacy protocol to copy V_1 end-to-end from C_B to C_P; there is no store-and-forward delay at any intermediate hop. Or it could leverage the Prognos core so that V_1 is first copied from C_B to S_1, then it is rsync’ed from S_1 to S_5, and finally, it is copied from S_5 to C_P.[1] Despite the store-and-forward delay of Prsync, it is almost 5× better than the legacy protocol due to the bandwidth saving on the weak link.

The second experiment demonstrates the usefulness of exploiting intermediate Stones. The results are summarized in the second row of Table 1. In this experiment, initially, C_W has version V_0, C_B has V_2, and S_3 has V_1 (as a result of satisfying a previous request, for example). The link conditions are the same as in the previous experiment. Now C_W desires to upgrade its file to V_2 and it has three options. The first two options are similar to the previous experiment: end-to-end copy from C_B to C_W, or using an end-to-end rsync in the Prognos core from S_1 to S_7. Because the content difference between V_1 and V_2 is small, the performance of these two options is similar to that seen in the first experiment. Option three, however, leverages the V_1 copy stored at S_3, as Prsync performs hop-by-hop rsync within the Prognos core. Only a small amount of data is exchanged across the weak link $S_1 \rightarrow S_2$, thereby improving performance.

The third experiment demonstrates the importance of adapting to environmental conditions. The performance of pair-wise exchange is shown in Figure 3 under different link bandwidth conditions. In this experiment, we attempt to upgrade the kernel file from version 2.0.20 to version 2.0.x, which constitutes the x-axis labels in the figure. We examine four different algorithms injected into two neighboring Stones. “Rsync” refers to the vanilla rsync algorithm. “Copy” refers to transferring the literal bytes. “Rsync-precomp” improves vanilla rsync by precomputing and storing per-block checksums. “Hybrid” adds the adaptive algorithm to “Rsync-precomp.” As expected, rsync performs well when the available bandwidth is scarce or when the file difference is small compared to the file size, and its performance can degrade significantly otherwise. Pre-computing checksums improves rsync by nearly a constant amount, but does not address the severe degradation that rsync can experience. The adaptive algorithm, though not always perfect, performs the best overall.

The fourth experiment is run on an overlay network comprising of 34 PlanetLab nodes. Initially, every node has a copy of version 2.0.21 of the Linux kernel. The file is then updated at the source and some random subset of the nodes synchronize their version with the newly published version. This process is repeated for versions 2.0.22 through 2.0.29. Figure 4 shows the performance of three alternatives: end-to-end rsync, hop-by-hop rsync over fixed paths defined by a tree topology, and hop-by-hop rsync over paths that are dynamically computed over a mesh topology. The tree-based hop-by-hop method shows improvements of more than 30% over the end-to-end rsync. The mesh-based method, combining network information with service-specific state, shows a further 30% improvement over the tree-based method.

[1] Note that in this and all subsequent Prsync experiments, data is always first written entirely to the disks at intermediate Stones (such as S_1 and S_5) before it is forwarded onto the next hop. Of course, this is not necessary and a pipelined version could have worked better.

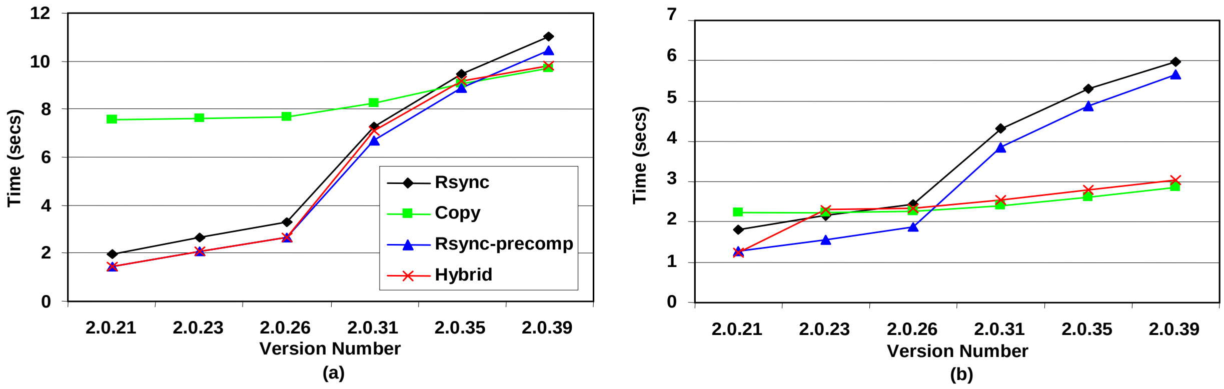

Figure 3: Performance of pair-wise exchange. (a) The link bandwidth is 25 Mbps. (b) The link bandwidth is 100 Mbps.

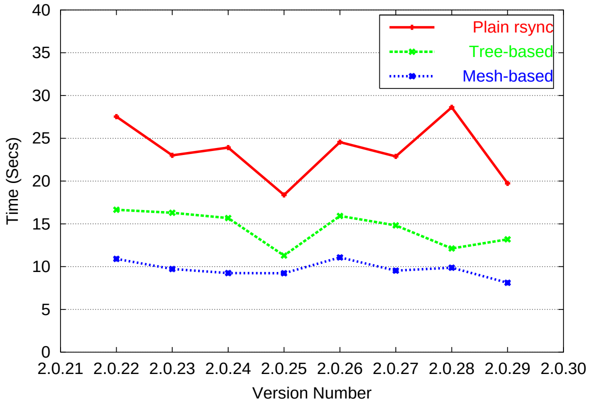

Figure 4: Average cost of upgrading an old copy of the Linux kernel to the current version available at the source.

2.4 Prsync Summary

Prsync demonstrates the utility of executing complex service-specific code (e.g., rsync) *at* the embedded storage elements. In addition, it shows how network-awareness can allow services to adapt their behavior dynamically and flexibly. The results illustrate the performance benefits of programmable network-embedded storage elements that can perform complex tasks, such as participating in hop-by-hop rsync protocols and executing application-specific routing algorithms. Such benefits are difficult to obtain without both programmability and network-awareness of embedded storage.

3 A Customizable Distributed File System

Today, we build cluster-based distributed file systems [6, 19, 30] that are very different from wide-area storage systems [14, 18, 27]. Life would be simpler if we only had to build two stereotypical file systems: one for LAN and one for WAN. The reality, however, is more complicated than just two mythical “representative” extremes: we face an increasingly diverse continuum, often with users and servers distributed across a complex interconnection of subnets.

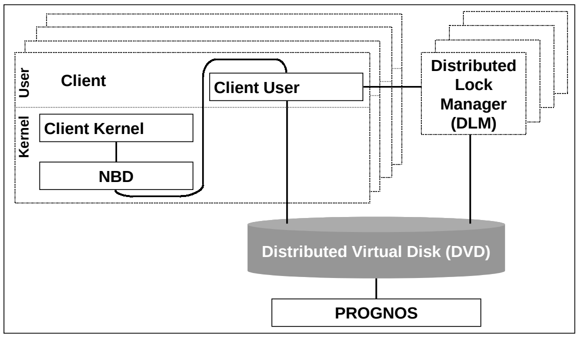

Figure 5: Components of Prognosfs.

Prognosfs is a “meta file system” in the sense that its participating Stones can be customized to allow the resulting system to exhibit different personalities in different environments. Prognosfs software has two parts: (1) a fixed framework that is common, and (2) a collection of injectable components that run on participating Stones and may be tailored for different workloads, and network topologies and characteristics. (In the near future, we envision injectable Prognosfs parts to be compiled from high-level specifications of the workload and the physical environment.)

3.1 Architecture and Component Details

Unlike several existing wide-area storage systems that support only immutable objects and loose coherence semantics [13, 14], Prognosfs is a read/write file system with strong coherence semantics: when file system update operations are involved, users on different client machines see their file system operations strictly serialized. Of course, we are not advocating that this is the only coherence semantics that one should implement—it just happens to be one of the desirable semantics that makes collaboration easy.

Figure 5 shows the Prognosfs parts in greater detail. The fixed part is similar to that of the Petal/Frangipani systems [19, 30]. For each file system call, a Prognosfs client kernel module translates it into a sequence of a lock acquisition, block reads/writes, and a lock release. This sequence is forwarded to a Prognosfs client user module via the Linux NBD pseudo disk driver. The read and write locks provide serialization at the granularity of a user-defined "volume" and they are managed by the Distributed Lock Manager. If a client fails without holding a write lock, no recovery action is required. If a client fails while holding the write lock of a volume, a recovering client inherits the write lock and runs `fsck` on the failed volume. These components of Prognosfs are fixed.

The customizable part of Prognosfs lies within the Distributed Virtual Disk (DVD). Externally, the interface to the DVD is very much like existing distributed virtual disks such as Petal [19]. The difference is that, internally, while all Petal servers are identical, the DVD consists of a number of peer Stones, each of which can run a specialized piece of code to perform functions such as selective caching, active forwarding, replication, and distribution of data to other Stones. These decisions can be made based on network topology, network condition, Stone load, and Stone capacity information that is typically either unavailable or difficult to determine accurately and responsively at the edge.

Figure 6 shows several example topologies. In Figure 6(a), clients on each of the two subnets can read data served by Stones on either subnet. If, for example, the clients of the right subnet repeatedly read data from Stones on the left, they might increase the load on the left subnet. As the "bridge Stone" S_b detects this access pattern, due to its awareness of the topology, S_b can take several possible actions to reduce the load: (1) S_b could cache data from the left subnet in its own persistent store. (2) If S_b itself becomes a bottleneck, S_b could forward a copy of the data to a Stone in the right subnet and this Stone would absorb future reads. (3) As reply data flows from the left subnet to a client in the right subnet, S_b could distribute the data across multiple Stones in the right subnet.

In Figure 6(b), the Stones in the middle layer (S_s) form a "switching fabric"—they accept requests from clients and perform functions such as load-balancing and striping as they forward requests to the next tier Stones. The role played by an S_s is analogous to that played by a μproxy, an NFS interposition agent [4]. Such interposition agents are just an example of the kind of functionalities that Prognosfs can enable. (Unlike a μproxy, the switching fabric is fully programmable, can have its own storage, and is not limited to the NFS protocol.)

In Figure 6(c), we replace a number of wide-area routers with their Stone counterparts. To see the role played by network-awareness, consider an example where S_4, on its clients' behalf, reads data stored at S_1. As data flows back on the path $S_1 \rightarrow S_0 \rightarrow S_2 \rightarrow S_4$, S_0 does not need to cache the data, S_2 may cache the data in the hope that S_3 may demand it later, and S_4 may cache the data in the hope that its own clients may demand it again. Once S_3 does read the cached data at S_2 and caches it itself, S_2 may choose to discard it.

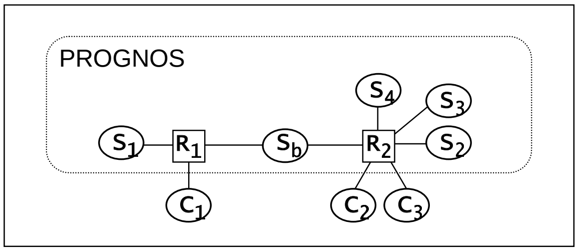

Figure 7: The topology of the Prognosfs testbed. An S_x is a Stone. A C_x is a "client". An R_x is a network switch that houses no disk and is not programmable. R_1 is a Netgear Fast Ethernet Switch FS108. R_2 is an Intel Express 510T Switch. All links are 100 Mbps.

In each of these examples, the function executed by a Stone is intimately associated with its often unique position in the network. Furthermore, although we have described the above Stone functions in the context of Prognosfs, the concepts are more generally applicable to other Prognos applications.

While the Prsync application (Section 2) relies on a known producer to ensure that a requester receives an up-to-date copy of the desired data, the presence of multiple readers and writers and the presence of multiple copies in Prognosfs demand a data location service from the underlying Prognos infrastructure. Given an object ID, the location service is responsible for locating *a* replica for a read request, and for locating *all* obsolete replicas to invalidate (or update) for a write request. This service is briefly described in Section 5.5.

We have implemented an initial prototype Prognosfs, along with a few of its incarnations that are customized to work for some different topologies. Existing applications on multiple Linux client machines are able to transparently read/write-share Prognosfs volumes.

3.2 Prognosfs Experimental Results

We describe results obtained on two platforms. The first is a network topology built inside our laboratory. Figure 7 gives its schematic diagram. Two switches (R_1 and R_2) are connected via a bridge Stone (S_b) and each switch is connected to a number of more Stones and clients. The Stones and the clients have same characteristics as those described in Section 2.3. The second platform consists of a wide-area configuration. Schematically, it looks similar to Figure 7, except that the Stones and the clients are distributed between two different sites. Stone S_1 and client C_1 are located in Arizona and all other nodes are located in Princeton. Communication between the two sites uses a weak wide-area Internet link.

A single *run* of our experiment has 8 phases. In Phase

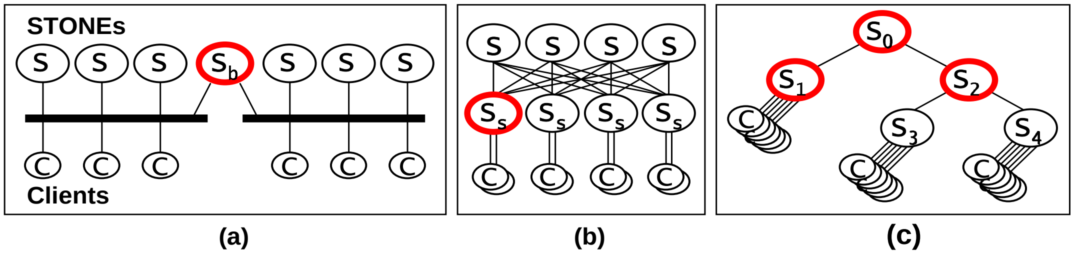

Figure 6: Example topologies connecting client machines with their Stones.

1, client C_1 creates data that is stored at its nearest Stone S_1. In the remaining phases, different sets of clients read the data created in Phase 1. A single *set* of experiments consists of 3 runs. In each of these runs, the bridge Stone S_b is programmed differently. We refer to the three cases as "Forward", "Cache" and "Distribute". In the "Forward" case, S_b simply forwards the data to the target client. For example, when C_2 requests data that resides only on S_1, S_b simply forwards C_2's request to S_1 and S_1's reply back to C_2. In the "Cache" case, S_b is also programmed to cache in its local persistent store any data that it forwards from one switch to the other. In the "Distribute" case, S_b also forwards an additional copy of the data to one of the Stones connected to the target switch in a round-robin fashion. Therefore, when forwarding data to a client connected to R_2, it forwards an additional copy to one of S_b, S_2, S_3 and S_4. Note that in the "Cache" and "Distribute" cases, the Prognos location service is invoked to keep track of the additional copies.

3.2.1 Exercising the DVD Interface

We first discuss the results from the platform built inside our laboratory, presented in the top half of Table 2. In phase 1, 100 MB of data is created by C_1 using the DVD interface. The data is stored on its nearest Stone S_1. In phase 2, C_1 reads the data back. The behavior of these phases are identical for the three runs. The bandwidth of these phases are limited by the link speed (and software overhead). In phase 3, C_2 reads the data. For the "Forward" case, the bandwidth experienced by C_2 is similar to that experienced by C_1. In the "Cache" and "Distribute" cases, however, the extra activity at S_b degrades the bandwidth experienced by C_2.

In phase 4, C_2 reads the data again. In the "Forward" case, the request is still satisfied by S_1 and the bandwidth observed by C_2 remains the same. In the "Cache" case, C_2 is able to read the cached copy at S_b. In the "Distribute" case, C_2 reads data from S_b, S_2, S_3, and S_4 in a striped fashion. In all these cases, C_2's bandwidth is again limited by the link speed. In phase 5, C_3 reads the data. Its bandwidth is similar to that experienced by C_2.

In phase 6, C_1 and C_2 read the data simultaneously. In the "Forward" case, the two clients are forced to share a single link to S_1. In the other two cases, C_1's requests are satisfied by S_1 while C_2 has its requests satisfied by Stone(s) connected to the other switch, so C_1 and C_2 both achieve near wire speed.

In phase 7, C_2 and C_3 read the data simultaneously. In the "Forward" and "Cache" cases, the two clients are forced to share the link to S_b. In the case of "Distribute", the two clients share the striped bandwidth to all the Stones connected to the right switch.

In phase 8, all three clients C_1, C_2, and C_3 read the data simultaneously. In the case of "Forward", all three clients contend for S_1's bandwidth. In the case of "Cache", C_1 monopolizes the bandwidth from S_1, while C_2 and C_3 share the bandwidth from S_b. In the case of "Distribute", all Stones are utilized and the clients achieve the greatest aggregate bandwidth.

The bottom half of Table 2 presents results for the wide-area configuration. The "Cache" and "Distribute" strategies, in addition to distributing the load among multiple Stones, also contribute toward masking the disadvantages of the weak wide-area link between the two collaborating sites. Data traverses the weak-link only once in Phase 3, and subsequent phases are able to finish with local communication only. We use a PlanetLab machine in Arizona as S_1, which apparently has a slower disk. This explains the relatively poor write performance in Phase 1. The performance during the remaining phases is as expected.

3.2.2 Exercising the File-system Interface

We now present results for experiments where clients use the Prognosfs file-system interface to write and read data. We were unable to run the file-system level benchmarks on the wide-area configuration because we lacked root access on the Arizona client machine. Therefore, we only present results for the platform built inside our laboratory. Table 3 reports an experiment where a 100 MB file is created in Phase 1 and read in the remaining phases. The results show trends similar to those in the top half of Table 2, except that the client bandwidth is degraded due to the overheads of going through the in-kernel NBD pseudo disk driver.

Table 4 presents results for a more general file-system

	Phase no.	1	2	3	4	5	6	7	8
		C_1 Write (MB/s)	C_1 (MB/s)	C_2 (MB/s)	C_2 (MB/s)	C_3 (MB/s)	C_1, C_2 (MB/s)	C_2, C_3 (MB/s)	C_1, C_2, C_3 (MB/s)
Laboratory	Forward	10.4	11.1	11.0	11.0	11.0	5.1, 5.1	5.6, 5.7	5.1, 3.5, 3.5
	Cache	10.4	11.1	10.6	11.0	11.0	11.1, 11.1	5.6, 5.6	11.1, 5.6, 5.6
	Distribute	10.4	11.1	6.2	10.9	11.0	11.1, 10.9	7.5, 7.2	11.1, 6.3, 6.3
Wide-Area	Forward	2.0	10.4	2.1	2.1	2.0	10.1, 1.3	1.1, 1.1	9.6, 0.9, 0.9
	Cache	2.0	10.4	1.8	11.0	11.0	10.7, 11.0	5.6, 5.6	10.7, 5.6, 5.6
	Distribute	2.0	10.4	2.1	10.9	11.0	9.6, 10.9	9.6, 9.6	10.1, 10.1, 9.8

Table 2: Client bandwidth when exercising the DVD interface.

Phase no.	1	2	3	4	5	6	7	8
	C_1 Write (MB/s)	C_1 (MB/s)	C_2 (MB/s)	C_2 (MB/s)	C_3 (MB/s)	C_1, C_2 (MB/s)	C_2, C_3 (MB/s)	C_1, C_2, C_3 (MB/s)
Forward	7.6	7.6	8.4	8.4	8.4	4.6, 4.6	5.4, 5.4	3.7, 3.1, 3.1
Cache	7.7	7.7	7.0	8.6	8.7	8.4, 8.6	5.5, 5.5	8.4, 5.5, 5.5
Distribute	7.3	7.3	5.6	8.4	8.4	8.4, 8.4	6.4, 6.5	8.4, 6.5, 6.5

Table 3: Client bandwidth when exercising the file system interface.

level benchmark called “MMAB”. It is a modified version of the “Modified Andrew Benchmark” [23]. (We modified the benchmark because the 1990 benchmark does not generate much I/O activity by today’s standards.) MMAB performs five steps—the first three are write steps and the last two are read-only steps. The first step creates a directory tree of 3,000 directories, in which every non-leaf directory has ten subdirectories. The second step creates one large file of size 50 MB. The third step creates three small files of size 4 KB in each of the directories. Step four computes disk usage of the directory tree by invoking `du`. The final step reads the files by performing a `wc` on each file. We present the results from running MMAB on our testbed in Table 4. In phase 1, the first three MMAB steps are performed on C_1. (The performance of these steps is shown by the three figures delimited by the two colons in each entry for phase 1 in Table 4.) Each of the remaining phases performs steps four and five. (The performance of these two steps is shown by the two figures delimited by the one colon in each entry from phase 2 to 8 in Table 4.) Again, the “Cache” and “Distribute” strategies pay the cost of replication in phase 3 for potential benefits in later phases.

3.3 Prognosfs Summary

Prognosfs is an example that illustrates some of the extremely diverse customizations made possible by programmable embedded storage. The example strategies, such as “Cache” and “Distribute”, and others mentioned in the context of Figure 6, serve to show that a fixed interface for embedded storage may not always be sufficient. Different strategies suit different system configurations, and even in a given configuration, the benefits of a given strategy are highly workload-dependent. Therefore, the ability to dynamically adapt the behavior of embedded storage is often important. In some cases, it may be possible to execute the functions mentioned above by issuing commands from the edges of the network, but this often incurs overheads and lacks the ability to quickly adapt to the workload.

4 Prognos Discussion

As mentioned in Section 1, a Prognos can be built on top of an overlay network, or even a set of wide-area routers. The Prognos approach, however, is equally applicable to both LAN and WAN environments. Previous cluster-based systems, such as several cluster file systems [6, 19, 30], assume an environment in which all nodes are at the same distance from each other. But, as soon as the system scales beyond a single subnet, as is the case in the Prognosfs example, a Prognos may become useful. Also, in the wide-area case, a Prognos does not necessarily need to involve a large number of hosts across the Internet: a small number of sites connected to a small number of strategically located Stones may benefit from a Prognos as well. This is the case for the Prsync example where a small number of Stones enlisted at strategic locations can allow novel services to be deployed without edge node cooperation.

In addition to the applications described above, we are continuing to research many other Prognos-based applications, including a network-embedded web crawler and a search engine. In this section, we generalize from these application studies and discuss some properties of the underlying Prognos. One objective of this section is show how most concerns related to resource management, security and reliability can be met by putting together several existing techniques.

Phase no.	1	2	3	4	5	6	7	8
	C_1 Write (s)	C_1 (s)	C_2 (s)	C_2 (s)	C_3 (s)	C_1, C_2 (s)	C_2, C_3 (s)	C_1, C_2, C_3 (s)
Forward	12:11:33	5:31	8:34	7:34	8:33	9:35, 14:39	11:37, 11:36	16:45, 26:51, 25:50
Cache	11:8:32	5:27	8:34	3:20	3:20	5:27, 3:20	4:26, 4:26	5:28, 4:26, 4:26
Distribute	11:13:33	5:30	33:73	3:21	3:21	5:31, 3:21	4:25, 4:25	5:30, 4:25, 4:25

Table 4: Results from the MMAB file-system level benchmark.

4.1 Resource Management and Security

The three key players in resource management are: the Stone Operating System (SOS), the service running on a Prognos, and the user of the service. In general, the user trusts the service, which in turn trusts the SOS. The SOS must protect different services from each other on a Stone; the distributed participants implementing the same service on multiple Stones must be able to authenticate each other; and the service must implement its own application-specific protection to protect its users from each other. We discuss each of these issues in turn.

One simple way of insulating the multiple services, which run on a Stone simultaneously, from each other is to employ one process per service per allocated Stone. Such a daemon is present as long as the service is up. Code specific to each service is executed within its own separate address space. Alternatives that are more efficient than the process model also exist. These include software-based fault isolation [32] and safe language-based extensions [9]. A Stone persistent storage partition is allocated exclusively to the service at service launch time. All other resources on a node must be accounted for as well. Resource accounting abstractions that are more precise than the process model, such as "resource containers" [7], may be needed. Existing network-wide resource arbitration mechanisms [11, 12, 29, 36] can be used to account for resources on a Prognos-wide scale.

All the participants that collaborate in a Prognos to implement a particular service, such as Stones allocated to this service and the processes on edge machines belonging to the service provider, must be able to authenticate each other. Existing cryptographic techniques for authentication, secure booting, and secure links can be used for this purpose [34, 17].

The codes that implement different services can choose their own means of authenticating their users. Application-specific access control and resource management is entirely left to individual services.

In practical terms, we understand that many may point at the absence of a single truly secure operating system today and be skeptical about the prospect of service providers vesting enough trust in a Prognos infrastructure. We believe that there are at least four reasons to be more optimistic. First, while programmable, the amount of functionality supported by an SOS is likely to be far more restrictive than that of a general operating system. We therefore conjecture that it is likely easier to engineer a secure SOS.

Second, we envision a Prognos to be administered in a more access-controlled manner than the current free-for-all Internet. The *storage consumers,* who are the direct clients of a Prognos, are distinct from the more general public who are the *service consumers*. Abusive behaviors might be more tractable when identities of the storage consumers are tracked. Such an access control system, however, need not impact the generality or flexibility of a Prognos.

Third, the Prognos approach does not necessarily imply time-sharing the Stones among multiple services. It is possible to have a restricted resource allocation policy that allocates dedicated Stones to services, thus avoiding the complexities, overheads and pitfalls associated with time-sharing.

Fourth, there are more restrictive deployment models of a Prognos that may further reduce its security risks. One example is a small-scale deployment that is managed by a single administrative domain where accesses to the network resources can be more strictly controlled and monitored. Another possibility is the use of a separate dedicated Prognos backbone network that is not available for public consumption. This backbone in effect becomes a "backplane" connecting a set of "core" Stones. The general public, or the service users, connect to the core via a distinct public network using a distinct service consumer interface. Of course, the service implementors are still responsible for "correctly" implementing their services and policing their service users; but at least the service users are prevented from committing mischief directly on the backplane. This is in spirit similar to how several cluster file systems can turn themselves into scalable legacy file servers [6, 19, 30]: a set of core cluster machines are connected by a secure private network that shoulders the intra-cluster protocol traffic while legacy clients connect to the core using a legacy protocol (such as NFS) on a different public network.

4.2 Reliability of Embedded-Storage

One question that the implementor of a Prognos must face is: What reliability guarantee does the system provide for the embedded persistent data (and whether the Stones must be backed up by tapes)? There are several possible answers to the question.

The first possible answer is not unlike the one proposed in a recent position paper [8]: it proposes that network-embedded storage may provide a "best effort" service whose reliability can only be characterized statistically and it is the responsibility of the edge storage consumers to cope with the potential loss of embedded persistent data in a way that is in spirit similar to how packet losses in a network are dealt with today. The authors pose the question of whether such limited-duration storage is useful. We believe that the answer is yes and the Prsync application is an example: the loss of any version of data stored on a Stone is not catastrophic and the edge producer is the last resort of any data.

The second possible answer is that it is the responsibility of the application-specific code injected into the Stones to provide redundancy (if any) inside a Prognos in a way that is under the exclusive control of the individual applications. The injected code would determine, for example, what redundancy scheme to use and which Stones to store the redundancy information on. The application may choose to treat different types of data differently and different Stones differently.

The third possible answer moves the responsibility of ensuring a certain degree of reliability into a "middleware" layer above Prognos. One example of such a system is an incarnation of the Prognosfs file system that, for example, always maintains replicas on at least two Stones or at two sites. By sacrificing some flexibility available at the Prognos layer, an application that runs on top of the file system layer may enjoy greater ease of programming.

In general, we believe that a Prognos should allow the storage consumers to pay the price of reliability only when they need it, and in a way of their own choosing.

5 Prognos Prototype

In this section, we describe a simple prototype Prognos on which the described applications run.

5.1 The SOS

Our prototype implementation uses the "process model" to run multiple services concurrently—on each Stone, code for a service is run in a separate daemon process. The service daemons request resources from the SOS, which is implemented as a simple Linux user-level process. One of the chief aims of building this prototype is to have a vehicle with which we can experiment with several Prognos-based applications and demonstrate the utility of the Prognos approach. To this end, we have not started with a potentially more efficient kernel-based or language-based implementation, nor have we provided any of the security mechanisms discussed in the previous section. We also anticipate the SOS interface to evolve in an ongoing application-driven process.

5.2 Code Injection

Service-specific code is injected into the Prognos at service launch time. Updating code requires re-starting the service. The Prognos supports an interface to allow services to inject code in native binary format. The code fragments injected into different Stones might be different because they may be tailor-made for Stones at different locations in the network.

5.3 Persistent Storage

Each service is allocated a separate storage partition on each participating Stone at service launch time. At each Stone, storage is available in three alternative forms, and a service is free to choose one or even switch among them. The alternatives are: (1) A raw disk partition interface that is essentially the Linux `/dev/raw/` interface. (2) A logical disk interface that is similar to several existing ones [15]. A user of this interface can read and write blocks that are keyed by their 64-bit logical addresses. This interface is useful for those who desire a block-level interface but do not care to explicitly manage their own storage layout. Our implementation is log-structured. Prognosfs uses this interface. (3) A subset of the Linux local file system interface. Prsync uses this interface.

5.4 Connectivity

The communication links between Stones can be either physical or virtual. The current SOS implementation enforces no resource arbitration mechanisms such as proportional bandwidth sharing [36], which we plan to add. The SOS also needs to be able to provide local connectivity information in the form of, for example, the set of neighboring Stones, and estimates of pair-wise bandwidth, latency and loss-rate.

5.5 Location Service

Our prototype includes an efficient, network-aware object location service to track copies of objects in a set of participating Stones. We refer to it as Canto (Coherent And Network-aware Tracking of Objects). Canto is heavily used by Prognosfs. It is designed as a network-aware generalization of the manager-based approach commonly used in cluster-based systems [6, 19, 30]. In these systems, each object has a designated manager to track all the copies of the object. This approach works well when all nodes are at equal distance from each other, as in a cluster-based system. When the network grows larger, or when the topology becomes more complex, the simplistic manager-based approach becomes inefficient.

Canto maintains a topology-sensitive tree of nodes. Object location requests are always routed along the edges of this tree. As copies of an object are created, state is added to the tree to keep track of the copies. Canto requires per node routing state that can be proportional to the product of the number of objects and the number of neighboring nodes. For this reason, Canto stores most of this routing state on disks and uses a memory buffer for caching and write-behind. In effect, Canto trades disk storage of routing state for reduced usage of wide-area networks and better performance.

Canto allows services (like Prognosfs) to make their own arbitrary object placement decisions. This is often difficult to achieve in location services based on distributed hash tables (DHTs) [28, 26], where hashing algorithms dictate the placement of objects, thereby allowing a DHT-based system to scale to extremely large number of nodes. Prognos has more modest scalability requirements: a small number of Stones enlisted at strategic locations is sufficient to deploy novel services such as Prsync. Canto works well for such modest-sized systems.

Another important property of Canto is that it is *self-synchronizing*: it preserves the integrity and consistency of its data structures during concurrent read and write operations without resorting to any external locking mechanism or fixed serialization points. Due to lack of space, we refer the reader to [35] for further details.

5.6 Lock Service

Another generic service that is likely to be useful for more than one Prognos-based applications is a distributed lock manager (DLM). For example, Prognosfs uses the DLM to synchronize its access to distributed storage. The DLM provides multiple-reader/single-writer locks to its clients. Locks are sticky so a client retains the lock until some other client requests a conflicting one. Interestingly, the mechanism for caching and invalidating lock state on distributed nodes is a special case of caching and invalidating generic objects inside the Prognos. Since caching and invalidation are handled by Canto, the DLM simply becomes an application of Canto.

6 Related Work

Many active network prototypes have been built [2, 16, 22, 33]. Prognos shares their goal of allowing new services to be loaded into the infrastructure on demand. Most active networking efforts to date, however, have consciously avoided tackling persistent storage inside the network. This decision typically limits the injected intelligence to those related to low-level forwarding decisions. By embracing embedded storage, Prognos makes it possible for services to inject high-level intelligence that is qualitatively different and more sophisticated.

In a DARPA proposal [21], Nagle proposes "Active Storage Nets," which are active networks applied to network-attached storage. In this proposal, active routers may implement storage functions such as striping, caching, and prefetching of storage objects, and quality-of-service responsibilities of I/O operations. "Logistical Networking", a system proposed in a recent SIGCOMM position paper [8], argues for an IP-like embedded storage infrastructure that allows arbitrary packets to manipulate the embedded storage using a fixed low-level interface. In our experience, applications such as Prsync and Prognosfs can fully benefit from the embedded storage only when application-specific intelligence, which could be more sophisticated than conventional caching of objects, is co-located with embedded storage.

Active technologies have been successfully applied to applications such as web caching [10] and media transcoding [3]. We hope to generalize these approaches for a wider array of applications that can benefit from network-embedded programmable storage. Active technologies have also been successfully realized in the context of "Active Disks" [1, 25]. One important difference between Active Disks and Prognos is that the intelligence in the former is at the "ends" of the network while in the latter case, it is embedded "inside" the network.

The applications, Prsync and Prognosfs, represent extensions to previous work that is either limited to client-server settings or lacks customizability. LBFS [20] is a client/server file system that employs a checksum-based algorithm to reduce network bandwidth consumption in a way that is analogous to rsync. By using the Prognos infrastructure, Prsync extends this approach to fully exploit multiple peer Stones and their network-awareness. Prognosfs is similar to Petal/Frangipani [19, 30] in its break down of the file system into three components: clients, a distributed lock manager, and a distributed virtual disk (DVD), but it improves upon existing cluster file systems that possess little network awareness [6, 19, 30]. The most novel part of Prognosfs lies within its DVD—the DVD consists of a number of peer Stones, each of which can be customized for a specific environment.

7 Conclusion

We describe two applications that gain significant performance and functionality benefits by using a clever combination of the programmability and network-awareness of network-embedded storage. These applications qualitatively and quantitatively show that such combination is necessary to exploit the full power of embedded storage. They are also evidence to support our belief that the benefits of such combination are not limited to content-distribution networks, but extend to many conventional applications too. The applications run on our prototype Prognos system that currently works on LAN clusters and wide-area PlanetLab-like overlay networks.

References

[1] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active Disks: Pro-

gramming Model, Algorithms and Evaluation. In *Proc. of ASPLOS* (1998).

[2] Alexander, D. S., Shaw, M., Nettles, S., and Smith, J. M. Active Bridging. In *Proc. of ACM SIGCOMM '97* (1997), pp. 101–111.

[3] Amir, E., McCanne, S., and Katz, R. H. An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In *Proc. of ACM SIGCOMM '98* (1998).

[4] Anderson, D., Chase, J., and Vahdat, A. Interposed Request Routing for Scalable Network Storage. In *Proc. of Operating Systems Design and Implementation* (2000).

[5] Anderson, D. G., Balakrishnan, H., Kaawhoek, M. F., and Morris, R. Resilient Overlay Networks. In *Proc. of the Eighteenth Symposium on Operating Systems Principles* (October 2001).

[6] Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., and Wang, R. Serverless Network File Systems. *ACM Transactions on Computer Systems 14*, 1 (1996).

[7] Banga, G., Druschel, P., and Mogul, J. C. Resource Containers: A New Facility for Resource Management In Server Systems. In *Operating Systems Design and Implementation* (1999).

[8] Beck, M., Moore, T., and Plank, J. S. An End-to-End Approach to Globally Scalable Network Storage. In *Proc. of ACM SIGCOMM 2002* (August 2002).

[9] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[10] Cao, P., Zhang, J., and Beach, K. Active Cache: Caching Dynamic Contents on the Web. In *Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing* (1998).

[11] Clark, D. Internet Cost Allocation and Pricing. In *Internet Economics* (Cambridge, MA, 1997), L. McKnight and J. Bailey, Eds., MIT Press, pp. 95–112.

[12] Clark, D., and Fang, W. Explicit allocation of best effort packet delivery service. *ACM Transactions on Networking 6*, 4 (August 1998), 362–273.

[13] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity* (2000).

[14] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-Area Cooperative Storage with CFS. In *Proc. of SOSP* (2001).

[15] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. Symposium on Operating Systems Principles* (1993).

[16] Decasper, D., Dittia, Z., Parulkar, G. M., and Plattner, B. Router Plugins: A Software Architecture for Next Generation Routers. In *Proc. of ACM SIGCOMM '98* (1998).

[17] Gibson, G., Nagle, D., Amiri, K., Chang, F., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., and Zelenka, J. File Server Scaling with Network-Attached Secure Disks. In *Proc. of the 1997 SIGMETRICS* (June 1997).

[18] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ASPLOS* (2000).

[19] Lee, E. K., and Thekkath, C. E. Petal: Distributed Virtual Disks. In *Conference on Architectural Support for Programming Languages and Operating Systems* (1996).

[20] Muthitacharoen, A., Chen, B., and Mazieres, D. A Low-bandwidth Network File System. In *Proc. ACM Symposium on Operating Systems Principles* (2001).

[21] Nagle, D. Active Storage Nets. http://www.ece.cmu.edu/˜ asn/old/pubs/-Active%20Storage%20Nets%20Intro.pdf, 1998.

[22] Nygren, E. L., Garland, S. J., and Kaashoek, M. F. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *Proc. of OpenArch'99* (1999).

[23] Ousterhout, J. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. of the 1990 Summer USENIX* (June 1990).

[24] Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First ACM Workshop on Hot Topics in Networking (HotNets)* (2002).

[25] Riedel, E., Gibson, G. A., and Faloutsos, C. Active Storage For Large-Scale Data Mining and Multimedia. In *Proc. of International Conference on Very Large Data Bases* (1998).

[26] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (November 2001), pp. 329–350.

[27] Rowstron, A., and Druschel, P. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of SOSP* (2001).

[28] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM* (2001).

[29] Stoica, I., Shenker, S., and Zhang, H. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. *Proceedings of SIGCOMM* (Aug. 1998), 118–130.

[30] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A Scalable Distributed File System. In *Proc. ACM Symposium on Operating Systems Principles* (1997).

[31] Tridgell, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[32] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proc. of the ACM Fourteenth Symposium on Operating Systems Principles* (December 1993).

[33] Wetherall, D. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *Proc. of the ACM Seventeenth Symposium on Operating Systems Principles* (1999).

[34] Wobber, E., Abadi, M., Burrows, M., and Lampson, B. Authentication in the Taos operating system. *ACM Transactions on Computer Systems 12*, 1 (February 1994), 3–32.

[35] Zhang, C., Lai, J., Garg, N., Sobti, S., Zheng, F., Krishnamurthy, A., and Wang, R. Coherent and Network-aware Tracking of Objects. Tech. Rep. TR-672-03, CS Dept., Princeton University, 2003.

[36] Zhang, M., Wang, R. Y., Peterson, L., and Krishnamurthy, A. Probabilistic Packet Scheduling: Achieving Proportional Share Bandwidth Allocation for TCP Flows. In *Proc. IEEE Infocom 2002* (June 2002).