Network-Embedded Programmable Storage and Its Applications

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

Abstract

As Internet-scale distributed services become an increasingly common fixture of our everyday computing landscape, application-specific storage that is geographically distributed and embedded at strategic locations inside the wide-area network is already a fact of life today. The way such services are launched today typically requires the service providers to reach agreements with data centers to acquire the needed space and storage and then to "hardwire" the acquired storage with application-specific logic. This is a time-consuming and costly process that only the largest service providers can afford to undertake and only occasionally. This state of art imposes a significant barrier to entry to smaller service providers and hinders short-term experimentations.

We propose a network-embedded programmable storage model that, in spirit, is analogous to that of active networks: each packet carries a customized code fragment that is executed at each network-embedded storage element to store, retrieve, manipulate, and/or transmit persistent data. The goal is to allow embedded storage consumers to more quickly deploy and customize new and flexible services. We shall refer to such a system as a PROGNOS (PROGrammable Network Of Storage). As all the computation and storage can be moved into the network, the only functionalities that are left at the edge are "reservoirs" of application-specific intelligence that is ready to be injected into the network core.

We discuss the requirements of operating systems support for a PROGNOS and describe a prototype implementation. We explore two extensive applications built on top of a PROGNOS. One is an incremental file transfer system tailor-made for low-bandwidth conditions. The other is a "meta distributed file system", a file system that can assume very different personalities in different topological and/or workload environments.

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

†Department of Computer Science, Yale University, arvind@cs.yale.edu.

1 Introduction

As Internet-scale distributed services become an increasingly common fixture of our everyday computing landscape, application-specific storage that is geographically distributed and embedded at strategic locations inside the wide-area network is already a fact of life today. Examples include content-distribution networks [21], cooperative web caches [10, 42], active web caches [17], peer-to-peer object storage and retrieval systems [13, 14, 23, 30], web crawlers and search engines [9], and email services [31]. The way such services are launched today typically requires the service providers to reach agreements with data centers to acquire the needed space and storage, and then to "hardwire" the acquired storage with application-specific logic. This is a time-consuming and costly process that only the largest service providers can afford to undertake and only occasionally. This state of art imposes a significant barrier to entry to smaller service providers and hinders short-term experimentations.

In contrast, users gain access to the *network* resources in a fundamentally different way today—a user can easily pay a small fee to access a small slice of the aggregate network resources: she does not "own" any connectivity "real estate" inside the network and does not have to hardwire any embedded resources with her application-specific logic. Indeed, the emerging field of active networking pushes this degree of freedom even further [35, 40]. Ideally, a user who desires to experiment with new services should be able to access embedded *storage* as easily as she would access *networks*.

If we agree that it is desirable to have storage embedded in the network fabric and to allow easy access to this resource so that innovators can quickly launch, experiment with, and tear down new services, the exact mechanism by which users access this storage still needs to be determined and it may be more complex than throwing a bunch of disks into the network and telling users to go at it.

One possibility is to define a fixed set of operations (such as caching) that can be used by any applications to manipulate the network-embedded storage. The abstraction level of such operations would be analogous to that of IP forwarding: the same low-level behavior is hardwired into all network-embedded elements; and it

is up to the applications running at the edge of the network to manufacture packets that carry a small amount of application-specific (or packet-specific) state to take advantage of the fixed and simple services hard-coded inside the network.

This approach would benefit a certain class of applications and we do not discount its usefulness for such applications. The obvious disadvantage of this approach is its lack of flexibility. Due to the extremely diverse needs of the embedded storage consumers, it appears difficult, if not impossible, to arrive at an embedded storage specification that caters well to all present and future application needs.

This concern leads us to a more radical alternative, one that is not unlike active networks in spirit, but has its own implications and challenges. Under the so-called "capsule" approach, each packet may carry a customized code fragment that is executed at each network-embedded storage element to store, retrieve, manipulate, and/or transmit persistent data. By decoupling application-specific intelligence from the network and its embedded storage infrastructure, we hope that this approach will enable embedded storage consumers to more quickly deploy innovative new services than is possible under a vendor-driven standardization process that dictates the exact functionalities of the embedded storage elements. We shall refer to such a system as a PROGNOS (PROGrammable Network Of Storage), each embedded storage element as a programmable STONE (STOrage Network Element), and the embedded operating system as an SOS (STONE Operating System). To summarize, the three key elements of a PROGNOS are: presence of embedded storage, its network-awareness, and its programmability.

As the key enabling technologies of a PROGNOS are maturing, some of the most important remaining issues concern the design of the interfaces visible to an embedded-storage programmer. We do not, however, pretend to know today what these interfaces should be. We believe that the evolution of the design and implementation of a PROGNOS over time needs to be application-driven. We have developed a prototype PROGNOS and two extensive PROGNOS-based applications that can intelligently exploit embedded storage that is both network-aware and programmable. One of them is a peer-to-peer incremental file transfer system tailor-made for low-bandwidth conditions. The other is a "meta distributed file system", a file system that can assume very different personalities in different topological and/or workload environments as we customize its participating STONES to exhibit different behaviors. One of the common themes demonstrated in these applications is that a PROGNOS is not only useful for deploying *different* applications, it also enables sophisticated customization inside the network within the *same* application—indeed, such customization would be difficult, if not impossible, to emulate at the edge of the network.

The remainder of this paper is organized as follows. Section 2 further motivates the need for embedded storage as we explore the relationship between PROGNOS and active networks. Section 3 answers a number of questions concerning the "network-awareness" and programmability properties of a PROGNOS. Section 4 discusses the requirements of operating systems support for a PROGNOS in general and describes our prototype implementation in particular. Section 5 describes the two PROGNOS-based applications. Section 6 describes some of the related work. Section 7 concludes.

2 The Role of Embedded Storage

It has been said that "the network *is* the computer." We believe, however, that the network *cannot* be the computer without having *storage* in it.

The original proponents of active networks envisioned an acceleration of the pace of innovation as new network services that are divorced from the underlying hardware are loaded into the infrastructure on demand. This approach has spawned a fertile ground for much creative research over the years. The reality today, however, is that relatively few new services have truly benefited from this approach.

There are many possible explanations for this phenomenon. Most researchers of the active networking efforts to date have consciously avoided tackling persistent storage inside the network—the various existing systems tend to only rely on a small in-memory "soft store". Indeed, the original active networking proposal only envisioned a persistent store that is to be used for purposes such as accounting/auditing logs [35]. As a result of excluding sophisticated persistent data management, the functions of the capsules tend to be limited to those that resemble IP forwarding (albeit more intelligent forwarding), all the active components inside the network remain simply as means of getting from one place to another, and consequently, there is not a great deal that one could accomplish inside the network. We believe that this restriction, while extremely valuable in isolating important research issues and simplifying engineering of existing systems, is one of the main factors that have limited the power of active capsules thus far.

In a PROGNOS, we seek to remove this restriction. While it is still possible to keep persistent data at the edge of the network and to only use the network-embedded nodes, or the STONEs, as transient caches, it is also possible to rely on the STONEs exclusively as an application's *only* persistent store and to have no other form of persistent store at the edge of the network at all. As *all* the computation and storage *can* be moved into the network, the only functionalities that are left at the edge are "reservoirs" of application-specific intelligence

that is ready to be injected into the network core. The network truly becomes the computer. We believe that the PROGNOS approach can vastly multiply the utility and power of the active capsules.

One analogy that we have found useful is to view the STONEs in a PROGNOS as generic "stem cells" that are not pre-hardwired with any special functionalities. When a stem cell receives a signal from the outside world, it transforms itself into a tailor-made building block of a specific organism. The separation of the responsibilities of stem cells from those of outside stimuli is analogous to the decoupling of the embedded storage infrastructure from application-specific intelligence. The stem cells no longer simply make up blood vessels that carry bits from one part of the organism to another, as is still largely the case in today's active networks—instead, the stem cells *are* the organism.

The PROGNOS approach may also foster the development of several industry sectors, each of them specializing and excelling in a more focused role. The first sector would include the equivalent of a Cisco and its mission would be the mass production of STONEs, devices that, to the first degree of approximation, are routers with disks. It is also responsible for loading the STONEs with a basic SOS. The second sector would include the equivalent of an ISP and its mission is to monitor the demand for embedded storage and to properly provision its network by purchasing enough STONEs to embed at strategic locations to meet customer demands. The third sector would include the equivalent of a Hotmail whose task should be mainly the development of innovative software that fills a particular user need. It pays its ISP-equivalent for the right of consuming a slice of the aggregate PROGNOS resources and the Hotmail capsules would subsequently enlist and coordinate a number of STONEs to implement the desired service. Such a triangular relationship already exists today for *network* resources (the Cisco-ISP-Hotmail relationship being an example) and the clean division of responsibilities has led to great advances in each of the three sectors. The PROGNOS approach seeks to extend this relationship to the embedded *storage* resources and we believe that a comparable division of responsibilities may lead to even greater blossom of advances in three parallel sectors.

3 Questions on Network-Awareness and Programmability

What makes the storage on a STONE different from that on a computer at the edge is that it is both network-aware and programmable. In this section, we explain these properties by answering several basic questions.

• ***What does it mean for a STONE to be "inside" the network?***

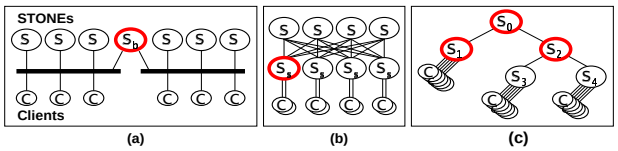

Figure 1: Example topologies connecting client machines with their STONEs that collectively implement a network service.

Being embedded inside the network means that the customized code running on a STONE has access to topology information and other network conditions and the code could exploit such information. Each STONE does not necessarily need to have accurate *global* information: local information might suffice. Without specifically naming the application involved, we illustrate this network-awareness with several examples of Figure 1.

In Figure 1(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 of the left subnet, they might increase the load on the left subnet or its STONEs. 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 on the left subnet: (1) S_b could cache data from the left subnet in its own persistent store. (2) If S_b itself becomes a bottleneck, as reply data flows from the left subnet to a client in the right subnet, S_b could forward a copy of the data to a STONE in the right subnet and this STONE would absorb future reads of the same data. (3) If the desired data in the left subnet is a large file, S_b could stripe its blocks across multiple STONEs in the right subnet. (4) S_b could passively monitor the amount of traffic destined to any STONE and adjust its decision accordingly.

In Figure 1(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 advantage of this architecture is that it minimizes client complexity. 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 the PROGNOS approach enables inside the network. (Unlike a μproxy, the switching fabric is fully programmable, can have its own storage, and is not limited to the NFS protocol.)

In Figure 1(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.

In each of these three topology examples, the function executed by a STONE is intimately associated with its network-awareness. We do not claim that any one of the example application-specific functions executed on a STONE in itself is novel. Indeed, the very fact that such functions have emerged and the extreme diversity of them make the flexibility offered by a PROGNOS attractive.

• ***Is a PROGNOS only useful in the wide-area?***

The PROGNOS approach is applicable to both LAN and WAN environments. Previous cluster-based systems, such as several cluster file systems [6, 24, 36], assume an environment in which all nodes are of the same distance from each other. As soon as the system scales beyond a single subnet, a PROGNOS might become useful. Note that a PROGNOS does not necessarily need to involve a massive number of hosts across the Internet: a small number of sites connected to a small number of strategically located STONEs can benefit from a PROGNOS just as well.

• ***Can the functions executed on STONEs be migrated to the edge of the networks?***

We define edge machines to be those that lack accurate topology and network condition information. In this context, users tend not to have a great deal of influence over where edge machines are placed. To understand the limitation of relying on edge machines, consider the topology shown in Figure 1(c). Suppose we would like to provide site-redundancy so each new data write is sent to two STONEs that are distributed at two sites. In a PROGNOS, a STONE (such as S_1) that receives new data from its clients can be easily programmed to send a replica to a STONE at a neighboring site (such as S_0). To move this functionality to the edge of the network, a client may be forced to send two separate copies into the network: one to S_1, and one to S_0, resulting in obvious inefficiency.

• ***What is a PROGNOS physically made of?***

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 (but loaded with a special SOS). The other potentially more efficient possibility is to co-locate a STONE with a router and the links among the STONEs will be largely physical. An extreme form of this co-location is to couple a router and a STONE in the same physical packaging.

4 Operating System Support

The two key interfaces are: a local per-node SOS interface, and a global network-wide PROGNOS interface. The SOS manages per-STONE resources and allows application code fragments on the STONEs to communicate with their counterparts on other STONEs. The PROGNOS interface allows an edge machine to interact with the PROGNOS network.

In this section, we first discuss how some of the existing technologies can work together to provide resource management and security in a PROGNOS: this part of the discussion is largely a survey that is not implemented in our prototype. We then describe some of the functionalities supported by our prototype SOS and PROGNOS. Few of the components discussed in this section, with the exception of the PHARO location service and the distributed lock manager built on top, are novel. Our aims are: (1) to make a case for the feasibility of the PROGNOS approach; (2) to provide a starting point for the discussion of the operating system interfaces; and (3) to provide a simple vehicle with which we can build several PROGNOS-based applications to demonstrate the potential power of embedded storage that is both network-aware and programmable. We do not pretend to know what the operating system interfaces should be today—we anticipate these interfaces to evolve in an application-driven process, and hopefully, in a research "marketplace".

4.1 Resource Management and Security

The three key players in resource management are: the SOS, an application-specific service running on a PROGNOS, and a 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 implements 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 that 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. Exceptional packets of each service are dispatched to its corresponding daemon and the code fragments are executed within the corresponding address space. Our current SOS disallows services to communicate with each other on a STONE. 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. More efficient alternatives than the process model also exist.

These include software-based fault isolation [39] and safe language-based extensions [8]. More portable options such as Java byte code (potentially complemented by Just-In-Time compiling technologies) may be necessary due to the lack of a uniform hardware STONE platform.

All the participants that collaborate in a PROGNOS to implement a particular service must be able to authenticate each other. These participants may include code fragments running on STONEs allocated to this service and the processes on the edge machines belonging to the service provider. Existing cryptographic techniques for authentication, secure booting, and secure links can be used for this purpose [41, 18]. Existing network-wide resource arbitration mechanisms [11, 12, 34, 43] can be used to account for resources on a PROGNOS-scale.

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 three 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, the way that we envision a PROGNOS to be used by storage consumers is likely to rely more heavily on an access-controlled system than the current free-for-all Internet model. These *storage consumers* are distinct from a 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, 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, 24, 36]: 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 Prototype PROGNOS Functionalities

Our prototype SOS is simply a Linux user-level process. (A stripped-down Linux kernel version that offers a subset of the existing system call interface is being planned for the near future.) One of the chief aims of this exercise is to have a vehicle with which we can experiment with several PROGNOS-based applications to demonstrate the utility of the PROGNOS approach. To this end, we have not started with a potentially more efficient kernel-based implementation, nor have we provided any of the security mechanisms discussed in the previous subsection in this initial prototype.

4.2.1 Code Injection

While our discussion so far might have implied a more radical approach of carrying a code fragment per packet, with the applications that we experiment with, we are satisfied with a more "discrete" approach—the application-specific code fragments are injected into the PROGNOS at service launch time. (Updating code fragments requires re-starting the service.) Subsequent packets are dispatched to the appropriate code fragments based on the application type and packet type.

The injected code is a dynamically-linked library (DLL) in native binary format. Once received by the SOS in a message, the DLL is incorporated into the running SOS process via the Linux dynamic linking loader interface. In a more realistic scenario, the honoring of the injection request would be subject to authorization, and the execution of the injected code would be subject to security enforcement mechanisms described in the last section.

While the SOS interface is responsible for honoring per-STONE code injection requests, the PROGNOS interface is responsible for distributing code fragments to the set of authorized STONEs specified by the service launch request. In our applications, the code fragments injected into individual STONEs might be different because these fragments may be tailor-made for STONEs at different locations in the network.

4.2.2 Persistent Storage

Each application is allocated a partition at service launch time. Due to the diverse needs of embedded storage consumers, however, we have found it difficult to settle on a single SOS storage interface. Instead, we

offer several alternatives and the storage user is free to choose one or even switch among them. The three 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, 32]. 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. (3) A subset of the Linux local file system interface.

While the SOS grants access to per-STONE storage, some may find it desirable to have a network-wide storage interface. We view the PROGNOS-scale storage services more as applications. We will describe a "distributed virtual disk" and a distributed file system in later sections. Some of these storage services may become useful for enough other services that they in effect become part of the PROGNOS interface.

4.2.3 Connectivity

The application code on a STONE communicates with its counterparts on other STONEs, which include both directly connected neighboring STONEs and remote STONEs. A link between two STONEs can be either a virtual overlay link or a physical link. In either case, code running on a sending STONE can explicitly name the receiving STONE so that there is no unnecessary interception or interpretation of the transient packets on intermediate STONEs. In other words, on a STONE that happens to lie in between a pair of communicating parties, the transient packet is not visible to the SOS process: the communication channels are end-to-end. The current SOS implementation enforces no resource arbitration mechanisms such as proportional bandwidth sharing [43], 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. The PROGNOS assembles this information into global topology information, which in turn impacts the application-specific intelligence being injected into individual STONEs.

4.2.4 Location Service

One main challenge is an efficient location service for locating a large amount of persistent data in a PROGNOS. Given an object ID, we need to locate *a* replica for a read request; and we may need to locate *all* obsolete replicas (if any) to invalidate or update them for a write request.

The most recent efforts in building location services for the wide area have produced distributed hash table-based systems such as Chord [33] and Pastry [29]. At least two characteristics of these systems make them unsuitable for the PROGNOS environment. First, in exchange for compact routing table representations on each node, these systems dictate the placement of objects in such a way that the higher-level systems lose the flexibility of making their own placement decisions. In general, these systems require $O(\log N)$ hops for locating an object in an N-node system. A PROGNOS application, in its quest of maximally exploiting the network information and reducing network messages, must be able to control the data placement with pinpoint accuracy.

Second, because these peer-to-peer data location systems were initially motivated by a "Napster-like" read-only environment, it is not clear how the distributed hash table-based approaches can efficiently support read-write use cases. While a read-only use case requires the system to locate *a* replica, a read-write use case requires the system to locate *all* replicas. If the system allows only a fixed number of replicas residing at locations determined by the hashes, locating the copies for invalidation upon writes is easy but read performance may suffer due to lack of caching. On the other hand, if the system allows caching and, therefore, an arbitrary number of replicas, the locations of the replicas are no longer determined by hashes and are therefore difficult to determine.

We have designed a new data location algorithm called PHARO, for Per-Hop Anchor-based ROuting. Unfortunately, we do not have the space to detail the workings of PHARO. Here, we highlight several key features. (1) The routing state for any piece of data is not widely dispersed; yet it is possible to route from any node in the system toward a replica efficiently. This controlled narrow distribution makes the updating of the routing state easy. (2) The algorithm maximally exploits locality in the presence of object reads, writes, and movements. For example, many existing location systems require a designated "manager" to be informed of any object movement; this requirement can be costly if the manager is far away. Under PHARO, only a small number of nearby routing entries may need to be updated. (3) PHARO allows the higher-level systems to retain the freedom of data placement decisions. (4) PHARO is self-synchronizing in that it preserves the integrity and consistency of its data structures during concurrent read and write operations without resorting to an external lock manager. (5) PHARO is self-recovering in that the loss of in-memory routing state on a participating PHARO node only degrades routing performance without negatively affecting correctness and the lost routing state is gradually rebuilt over time as more routing operations are performed. Despite its advantages, we view PHARO only as one of the possible location services.

4.2.5 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, the PROGNOS-based distributed file system, which we describe in a later section, 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 PHARO, the DLM simply becomes an application of PHARO.

5 Applications

The two applications that we now describe share these common themes: (1) they benefit from embedded-storage; (2) they benefit from topology and other network information; and (3) they benefit from the flexibility afforded by a programmable infrastructure. Our goal is to demonstrate that a PROGNOS enables both the quick deployment of new services and sophisticated customization of any single service. We believe that the principles that we demonstrate with the two example applications are generally applicable to virtually all applications that use distributed storage.

5.1 Peer-to-Peer Incremental File Transfer

The "backbone" connecting the STONEs inside a PROGNOS need not necessarily be a high-performance one. In fact, the presence of weak links and variable connectivity quality might make a PROGNOS more compelling. In this section, we explore an application that exploits embedded programmable storage to overcome such weaknesses.

5.1.1 rsync

The inspiration for this application originates in the `rsync` program [37]: it seeks to efficiently synchronize file contents across the network for incrementally changing data. The intuition behind the algorithm is that one should only transmit the differences between an older version and the fresh version. The challenge is to detect and compute the differences when the two files in question are not co-located on the same machine. Suppose node X has an older version of the data and it demands a fresh copy from node Y. Under the rsync algorithm, X communicates to Y the checksum values for every block in its version of the file. Upon receiving these checksums, Y performs a "sliding window" calculation to identify which portions of the latest version are carried over from the older version. It then sends X the contents of the new version as a sequence of literal bytes (for contents that have changed) and block identifiers (for contents that have not changed). A client/server file system for low-bandwidth networks [25] has subsequently used a similar approach.

We have developed a system that we call "Peer-to-Peer PROGNOS-based rsync", or P^{3}rsync. We shall examine three aspects of P^{3}rsync below. First, a PROGNOS enables the rapid deployment of rsync-like innovations, even in cases where one does not have full cooperation of edge machines. Second, while vanilla rsync is end-to-end, P^{3}rsync is peer-to-peer: peer STONEs in the PROGNOS infrastructure core collaborate using pair-wise rsync exchanges to further improve end-to-end performance. Third, while vanilla rsync executes a fixed algorithm, P^{3}rsync adapts to its environment conditions by exploiting the network-awareness of STONEs.

5.1.2 Interaction with Legacy Protocols

Let us consider the following scenario. Suppose a newer file needs to be transferred from `cs.berkeley.edu` to `cs.princeton.edu` but `cs.berkeley.edu` lacks the intelligence to participate in a pair-wise rsync exchange. To circumvent this handicap, a user can locate two STONEs in the PROGNOS core, such as `berkeley.prognos.com` and `princeton.prognos.com`, and injects into these two STONEs intelligence that performs the following steps for each file transfer: (1) copy the file from `cs.berkeley.edu` to `berkeley.prognos.com` using a legacy protocol; (2) rsync the file between `berkeley.prognos.com` and `princeton.prognos.com` across a potentially weak wide-area link; and (3) copy the file from `princeton.prognos.com` to `cs.princeton.edu` using a legacy protocol. This is an example of an end-to-end legacy protocol that can benefit from sophisticated intelligence inside the infrastructure. In addition to serving as scratch space, persistent storage on the two STONEs can be used for caching and/or push. Among the persistent storage interface alternatives, P^{3}rsync chooses the Linux file system.

5.1.3 Peer-to-Peer Interaction

In the above example, the long-distance rsync between `berkeley.prognos.com` and `princeton.prognos.com` could be further improved if we were to enlist more intermediate STONEs to decompose a long-distance rsync into a sequence of short-distance hop-by-hop rsyncs. In a long-distance rsync, although an intermediate STONE may lie on the communication path, it cannot always use the data that passes through it to update a copy stored in its own local persistent store. This is because the data in transit is the difference with respect to the requesting node's version and is not always meaning-

ful with respect to the version stored at an intermediate node. Consequently, the intermediate STONEs may continue to have stale copies and are incapable of servicing future rsync requests routed towards them by themselves. This is unfortunate because if intermediate STONEs were capable of updating their local versions, a version stored by an intermediate STONE is likely to be more similar to the latest version. A long-distance update does not attempt to take advantage of this potential proximity but instead ends up conveying potentially large updates over the entire path.

We now describe the P^{3}rsync protocol. (Due to time constraints, P^{3}rsync is not yet running on top of the PHARO location service so we have made some simplifying assumptions; these assumptions, however, do not impact the main principles that we seek to demonstrate with this application.) We assume a single *producer* of the desired file. (More complex sharing semantics are supported by the second application, the distributed file system.) A file requester first obtains the latest time stamp of the desired file from the producer. The PROGNOS does not participate in this exchange and the working of P^{3}rsync is independent of the mechanism used for locating the producer. The file requester then presents the name and time stamp of the desired file to a nearby STONE that participates in P^{3}rsync. If the fresh file is present on the STONE, the request is satisfied there. If not, the request is forwarded to the next STONE towards the producer. While a request is outstanding, additional requests for the same file on a STONE are queued until the outstanding request is satisfied. The request forwarding process repeats until the request reaches a machine with a fresh copy, at which point the fresh version is propagated back to the requester by having each intermediate STONE rsync with its upstream STONE. Intermediate STONEs are thus updated and could service future requests. Furthermore, if an intermediate STONE has a copy that enjoys substantial similarities with the latest version, it can be updated with relatively low communication costs.

What this peer-to-peer protocol demonstrates foremost is that simple caching in intermediate STONEs is not sufficient—instead, the programmability of PROGNOS enables the STONEs to participate in a sophisticated protocol.

5.1.4 Adapting to Changing Environments

Rsync employs a relatively computationally expensive checksum and compression algorithm. The use of this algorithm 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 the infrastructure to intelligently adapt to these environmental factors, the network-awareness and the programmability of a PROGNOS becomes indispensable.

The heuristics that we have introduced to illustrate the utility of dynamic adaptation is the following. 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 has been actually transferred and the size of the portion 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 pair would abandon the checksum-based rsync algorithm and revert to simply transmitting the literal bytes of the fresh file. More sophisticated heuristics that take latency, loss rates, and the existence of different routes into consideration are also possible. Note that all these 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 that we have introduced is computing offline and storing the per-block checksums along with a file in the STONE (or the requester) persistent store. This precomputation of checksums reduces some of the rsync overhead.

5.1.5 Experimental Results

Figure 2 shows the topology of the network on which we conduct the P^{3}rsync experiments. Table 1 gives some of the experimental platform characteristics. Nodes C_B, C_P, and C_W are considered "edge" machines. The remaining machines make up a PROGNOS core. All the links are dedicated (separate) 100 Mbps physical links. 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 24.5 MB in size. We show results of three sets of experiments, each of which demonstrates one of the aspects detailed in Sections 5.1.2, 5.1.3 and 5.1.4.

The first set of experiments demonstrate the ability of a PROGNOS to overcome a legacy protocol. The results are summarized in the first row of Table 2. In this set, 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 100 Mbps. (The effect of the weak link is achieved by using the `--bwlimit` option of the `rsync` command.) 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-

Processor	Dual Intel Pentium III, 933 MHz
OS	Linux 2.4.10
Memory	1GB PC133 ECC SDRAM
Disk model	Maxtor 96147U8
Capacity	61,471 MB
RPM	5400
Average seek	9 ms
Transfer rate	40.8 MB/s (data sheet)
Bandwidth	26.5 MB/s (measured)
Ethernet Adaptor	Realtek PCI 10/100 Mbps
Network Hub	Linksys Fast Ethernet 10/100 Mbps

Table 1: Platform characteristics.

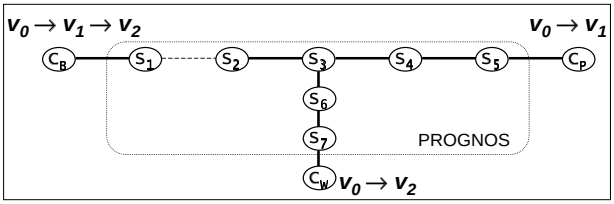

Figure 2: The topology of the P^{3}rsync testbed.

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. Note that in this and all subsequent P^{3}rsync 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 much better.) Despite the store-and-forward delay of the latter P^{3}rsync approach, however, it is almost 5× better than the former due to the bandwidth saving on the weak link.

The second set of experiments demonstrate the usefulness of exploiting intermediate STONEs by injecting them with a customized protocol. The results are summarized in the second row of Table 2. In this set, 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 by some other requester, for example). The link condition is the same as that of the previous set of experiments. Now C_W desires to upgrade its file to V_2 and it has three options. The first two options are similar to the two experiments that we have performed earlier: 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 experiments is similar to that seen in the first set. Option three, however, leverages the V_1 copy

Requester	Versions	e-to-e copy (s)	e-to-e rsync (s)	p-to-p 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 2: P^{3}rsync performance.

stored at S_3, as P^{3}rsync performs peer-to-peer rsync within the PROGNOS core. Only a small amount of data is exchanged across the weak link $S_1 \rightarrow S_2$ and the resulting performance is much better than that of the first two options.

The third set of experiments demonstrate the importance of adapting to environmental conditions. The performance of pair-wise exchange is shown in Figure 3 under different link bandwidth conditions. In these experiments, we attempt to upgrade the Linux kernel `tar` 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 the per-block checksums. "Hybrid" adds the adaptive algorithm explained in Section 5.1.4 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.

In summary, the P^{3}rsync experiments demonstrate the following PROGNOS principles: (1) a PROGNOS infrastructure eases the deployment of new protocols, especially in the absence of edge host support; (2) programmability is not always sufficient by itself without access to embedded storage; (3) embedded storage is not always useful by itself without general programmability; and (4) it is important to exploit network-awareness of the infrastructure in adapting to environmental changes. We believe that these principles are generally applicable to other PROGNOS applications as well.

5.2 A Meta Distributed File System

We now briefly describe a file system that we call "Peer-to-Peer PROGNOS-based File System", or P^{3}FS.

5.2.1 A Framework for Customizable Parts

Today, we build peer-to-peer cluster file systems [6, 24, 36] that are very different from peer-to-peer wide area storage systems [13, 14, 23, 30]. 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 are facing an increasingly diverse continuum. Each of the topologies shown in Figure 1 of Section 3, for example, has a good reason to exist,

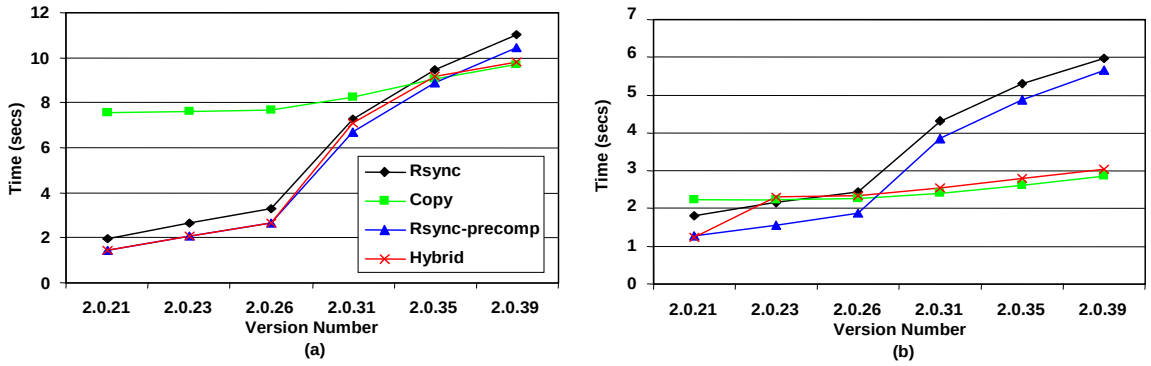

Figure 3: Performance of pair-wise exchange. (a) The link bandwidth is 25 Mbps. (b) The link bandwidth is 100 Mbps.

and each topology demands potentially very different distribution strategies.

A P^{3}FS is a "meta file system" in the sense that its component STONEs can be customized to allow the resulting system to exhibit different personalities in different environments. It has two parts: (1) a fixed framework that is common across all its incarnations, and (2) a collection of injectable components that run on participating STONEs and may be tailor-made for different workloads, and network topologies and characteristics. Simple injectable P^{3}FS parts may even be compiled from high-level specifications of the workload and the physical environment.

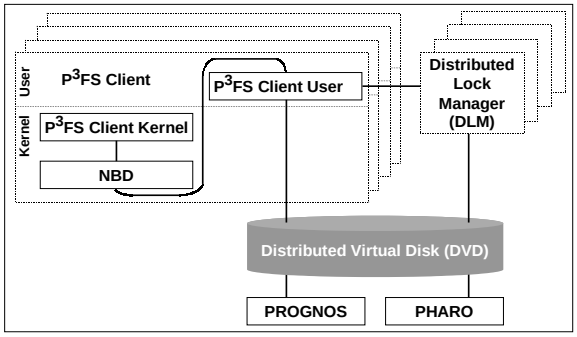

Figure 4: Components of P^{3}FS.

5.2.2 Architecture and Component Details

Unlike several existing wide-area peer-to-peer storage systems that support only immutable objects and loose coherence semantics [13, 14, 30], P^{3}FS 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* coherence semantics that one should implement—it just happens to be one of the desirable semantics that makes collaboration easy: if Bob writes a file and phones Alice to read it remotely, it would be desirable for Alice to see the version promised by Bob.)

Figure 4 shows the P^{3}FS components in greater detail. The fixed part of P^{3}FS is similar to that of the Petal/Frangipani systems [24, 36] and is hardly novel. For each file system call, a P^{3}FS client kernel module translates it into a sequence of a lock acquisition, block reads and/or block writes, and a lock release. This sequence is forwarded to a P^{3}FS 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 (DLM) described in Section 4.2.5. 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 P^{3}FS are fixed.

The more interesting part of P^{3}FS lies within the Distributed Virtual Disk (DVD). Externally, the interface to the DVD is very much like existing distributed virtual disks such as Petal [24]. 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 those described in Section 3. These decisions can be made on a hop-by-hop basis based on network topology and condition information typically unavailable at the edge. The code running on a STONE uses the (log-structured) logical disk interface exported by the SOS. Another novel aspect of the DVD is its use of the PHARO location service (introduced in Section 4.2.4) to publicize and to locate blocks. PHARO allows the DVD to retain complete control over data placement decisions. PHARO carefully limits the extent to which route update and lookup messages must travel within the PROGNOS network while still guaranteeing a consistent external view of the DVD.

A prototype P^{3}FS has been implemented, 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 P^{3}FS volumes.

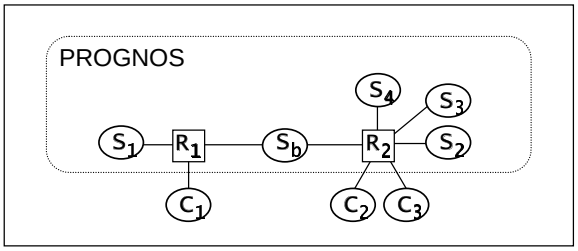

Figure 5: The topology of the P³FS testbed. An S_x is a STONE. A C_x is a "client". An R_x is a network switch that houses no disk and is not programmable. R_1 is a Netgear Fast Ethernet Switch FS108. R_2 is an Intel Express 510T Switch. All links are 100 Mbps.

5.2.3 Experimental Results

Figure 5 shows the topology of the network on which we conduct the P³FS experiments: 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 platform characteristics are the same as those given in Table 1.

We run three experiments. Each experiment runs on a newly initialized P³FS so there is no interaction among the different experiments. Each experiment consists of eight phases. Each phase directly exercises the disks on the STONEs and benefits from no caching at any node. The only difference among the three experiments is that the STONEs (S_b in particular) are injected with different intelligence. Table 3 reports the bandwidth obtained by a client that directly exercises the DVD interface during each phase for each experiment. Table 4 reports the bandwidth obtained at the file system interface. We now describe the details of the different phases and the different intelligence that is injected into the STONEs.

In phase 1, C_1 creates a 100MB file, which is stored on the nearest STONE S_1. In phase 2, C_1 reads the file back. The behavior of these phases are identical for the three experiments. The bandwidth of these phases are limited by the link speed (and software overhead). In phase 3, C_2 reads the file written by C_1 in phase 1.

In each of the three experiments, the STONEs are programmed to respond differently. In the case that we call "Forward", the bridge STONE S_b is programmed to always forward a request from the right switch into the left switch to satisfy C_2's request directly from S_1's disk. The bandwidth experienced by C_2 is similar to that experienced by C_1.

In the case that we call "Cache", S_b is programmed to cache a copy of the data in its local persistent store whenever a client connected to one switch demands data from a STONE connected to a different switch. The bandwidth experienced by C_2 during phase 3 is a little worse due to this extra activity. Note that the PHARO routing state must be updated to reflect the newly cached copies and this activity also contributes to the extra overhead.

In the case that we call "Distribute", in addition to sending the requested data back to C_2, S_b is programmed to forward an additional replica in a round-robin fashion to all the STONEs connected to the right switch: S_b, S_2, S_3, and S_4. The PHARO routing state again needs to be updated to reflect this distribution. All this extra forwarding degrades the bandwidth experienced by C_2 significantly during phase 3.

In phase 4, C_2 reads the same file again. In the "Forward" experiment, 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 is bandwidth is again limited by the link speed. In phase 5, C_3 reads the same file. Its bandwidth is similar to that experienced by C_2.

In phase 6, C_1 and C_2 read the same file simultaneously. In the "Forward" case, the two clients are forced to share a single link to S_1. In the other two experiments, C_1 is able to monopolize the bandwidth from S_1 while C_2 has its request 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 same file 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. Their bandwidth is limited by the internal contention within the switch.

In phase 8, all three clients C_1, C_2, and C_3 read the same file 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.

We report results for an enhanced version of the "Modified Andrew Benchmark" [20, 27], which we call "MMAB". (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 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, resulting in a total of 9,000 small files. 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 5. At the beginning of the experiments, steps one through three are performed on C_1. (The performance of these steps is shown by the three figures delimited by the two for-

Phase no.	1	2	3	4	5	6	7	8
S_b Function	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	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

Table 3: Client bandwidth when exercising the DVD interface.

Phase no.	1	2	3	4	5	6	7	8
S_b Function	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 4: Client bandwidth when exercising the file system interface.

ward slashes in each entry for phase 1 in Table 5.) We then measure the cost of executing steps four and five at different sets of clients with different injected bridge STONE functions. (The performance of these two steps is shown by the two figures delimited by the one forward slash in each entry from phase 2 to 8 in Table 5.)

The "Cache" and "Distribute" strategies pay the cost of replication in phase 3 for potential benefits in later phases. Whether these strategies will pay off, of course, is highly workload- and topology-dependent. And other strategies, such as aggressive data pushing, also exist. The example intelligence that we have described in this section shows the extreme diverse and sophisticated customization that is enabled by the PROGNOS framework.

6 Related Work

Many active network prototypes have been built [2, 16, 26, 40]. PROGNOS shares their goal of decoupling network services from the underlying hardware and allowing new services to be loaded into the infrastructure on demand. Most existing active networking efforts to date, however, have consciously avoided tackling persistent storage inside the network. This decision limits the typical active net intelligence to those related to forwarding decisions. By embracing embedded storage, PROGNOS makes it possible to truly compute inside the network. The sophistication of the applications enabled by PROGNOS is qualitatively different. We view active networking as an enabling technology for PROGNOS.

Active technologies have been successfully applied to more specific applications such as web caching [10] and media transcoding [3]. Storage systems have successfully exploited a greater level of intelligence in the context of secure storage [18, 19] and file system interposition agents [4]. 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 applied to high-performance LAN environments in the context of parallel programming [38] and parallel processing embedded inside "Active Disks" [1, 22, 28]. One important difference between Active Disks and PROGNOS is that the intelligence in the former is at the "ends" of the network while the intelligence in the latter is embedded "inside" the network. PROGNOS focuses on the exploitation of more general network awareness instead of just the exploitation of parallelism.

LBFS [25] 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 [37]. By using the PROGNOS infrastructure, P^{3}rsync seeks to extend this approach for a peer-to-peer context while exploiting the network awareness of these peer components.

Existing cluster file systems possess little network awareness [6, 24, 36]. P^{3}FS is similar to Petal/Frangipani [24, 36] in its break down of the file system into three components: clients, a distributed lock manager, and a distributed virtual disk (DVD). The most novel part of P^{3}FS lies within its DVD—the P^{3}FS DVD consists of a number of peer STONEs, each of which can be customized for a specific environment.

7 Conclusion

Active networking started a movement of decoupling infrastructure from application-specific intelligence to foster the development and deployment of network services. We believe that it is the logical next step to elevate embedded storage to the status of a first class citizen in the drive of active capsule research. This would be a natural progression since networks and storage are two complementary technologies: the judicious use of one can reduce an application's demand on the other. A network-embedded programmable storage can not only

Phase no.	1	2	3	4	5	6	7	8
S_b Function	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 5: MMAB results.

facilitate the rapid deployment of a larger class of new applications, it can also enable sophisticated customization of existing applications.

References

[1] Acharya, A., Uysal, M., and Saltz, J. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)* (October 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), pp. 178–189.

[4] Anderson, D., Chase, J., and Vahdat, A. Interposed Request Routing for Scalable Network Storage. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 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 (Feb. 1996), 41–79.

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

[8] 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 *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[9] Brin, S., and Page, L. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW7/Computer Networks 30*, 1-7 (1998), 107–117.

[10] Cao, P., Zhang, J., and Beach, K. Active Cache: Caching Dynamic Contents on the Web. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)* (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. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. ICSI Workshop on Design Issues in Anonymity* (July 2000).

[14] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-Area Cooperative Storage with CFS. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001), pp. 202–215.

[15] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[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), pp. 229–240.

[17] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.

[18] 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).

[19] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)* (October 1998).

[20] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[21] Karger, D. R., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web Caching with Consistent Hashing. *WWW8/Computer Networks 31*, 11-16 (1999), 1203–1213.

[22] Keeton, K., Patterson, D. A., and Hellerstein, J. M. A Case for Intelligent Disks (IDISKs). *SIGMOD Record 27*, 3 (August 1998).

[23] 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 *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2000)* (November 2000).

[24] Lee, E. K., and Thekkath, C. E. Petal: Distributed Virtual Disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 84–92.

[25] Muthitacharoen, A., Chen, B., and Mazieres, D. A Low-bandwidth Network File System. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001).

[26] 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* (March 1999).

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

[28] Riedel, E., Gibson, G. A., and Faloutsos, C. Active Storage For Large-Scale Data Mining and Multimedia. In *Proc. of the 24th International Conference on Very Large Data Bases* (August 1998).

[29] 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.

[30] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001).

[31] Saito, A., Bershad, B., and Levy, H. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 298–332.

[32] Sobti, S., Garg, N., Zhang, C., Yu, X., Krishnamurthy, A., and Wang, R. Y. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proc. of First Conference on File and Storage Technologies* (January 2002).

[33] 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* (August 2001).

[34] 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.

[35] Tennenhouse, D. L., and Wetherall, D. Towards an Active Network Architecture. In *Proc. Multimedia Computing and Networking 96* (January 1996).

[36] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of the ACM Sixteenth Symposium on Operating Systems Principles* (Oct. 1997).

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

[38] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. E. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (May 1992), pp. 256–266.

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

[40] Wetherall, D. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the ACM Seventeenth Symposium on Operating Systems Principles* (December 1999).

[41] 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.

[42] Wolman, A., Voelker, G. M., Sharma, N., Cardwell, N., Karlin, A., and Levy, H. M. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the ACM Seventeenth Symposium on Operating Systems Principles* (December 1999).

[43] 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).