Network-Embedded Programmable Storage and Its Applications

Sumeet Sobti[1], Junwen Lai[1], Yilei Shao[1], Nitin Garg[1], Chi Zhang[1], Ming Zhang[1], Fengzhou Zheng[1], Arvind Krishnamurthy[2*], and Randolph Y. Wang[1*]

[1] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA.
{sobti, lai, yshao, nitin, chizhang, mzhang, zheng, rywang}@cs.princeton.edu.
[2] Department of Computer Science, Yale University, New Haven, CT 06520, USA.
arvind@cs.yale.edu

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, that 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

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

N. Mitrou et al. (Eds.): NETWORKING 2004, LNCS 3042, pp. 1143–1155, 2004.

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 [1].

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 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.

We also discuss the requirements of a general infrastructure to support easy and effective access to programmable network-embedded storage, and describe a prototype implementation (Section 4). 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). 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. The overlay links used should approximate the underlying physical topology and the Stones can simply be general-purpose computers. The

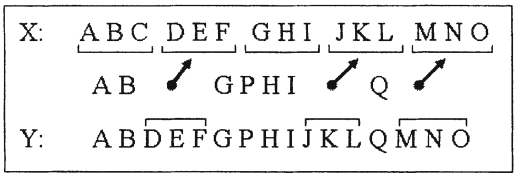

Fig. 1. A simple rsync example.

other potentially more efficient possibility is to co-locate a Stone with a router and the links among the Stones would largely be physical.

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 collaboratively supported platform, such as PlanetLab [2] (www.planet-lab.org), can be turned into a Prognos by loading the participating machines (also referred to as Stones) with the SOS module.

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 [3] (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 co-operation 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.

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.

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. 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 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.

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 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 has 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 pre-determined 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.

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 maintain estimates of the differences between the file versions at different stones and also monitor pair-wise bandwidths. 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.

2.3 Summary of Prsync Experimental Results

We have experimentally validated the Prsync design. The experiments were performed on two testbeds – one constructed in our laboratory, and another implemented on a set of PlanetLab machines distributed across the wide-area. Due to lack of space, we refer the reader to [4] for details.

Here, we only provide a brief summary of the results as they relate to the four aspects of Prsync described in the previous section. (1) The rsync protocol is observed to perform more than 5× better than simpler legacy protocols for

copying files, especially in low bandwidth conditions. (2) Hop-by-hop use of rsync can improve upon end-to-end rsync by an additional factor of 2. These results demonstrate the utility of executing complex service-specific code (e.g., rsync) at the embedded storage elements for functionality and performance gains. (3) The adaptive nature of Prsync allows it to perform well in a diverse range of network conditions. Lack of adaptivity can degrade performance by as much as 2×. (4) In a PlanetLab experiment, the mesh-based method of selecting propagation paths performs 30% better than the tree-based method, which in turn performs about 30% better than a simple end-to-end rsync. These results demonstrate the kind of performance benefits that a service can get by being network-aware, and by intelligently using network information in a service-specific manner.

3 A Customizable Distributed File System

Today, we build cluster-based distributed file systems [5,6,7] that are very different from wide-area storage systems [8,9,10]. 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.

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 [11,8], 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 2 shows the Prognosfs parts in greater detail. The fixed part is similar to that of the Petal/Frangipani systems [6,7]. 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

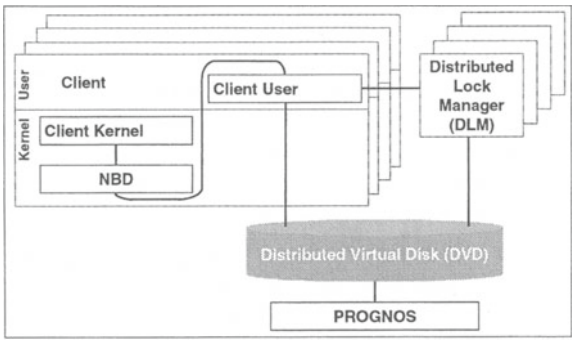

Fig. 2. Components of Prognosfs.

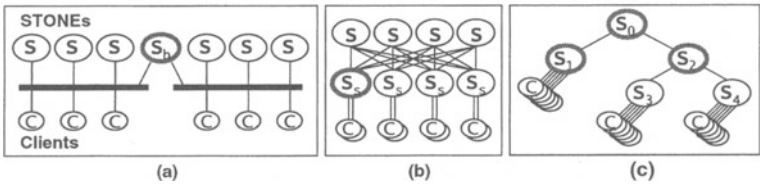

Fig. 3. Example topologies connecting client machines with their Stones.

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 [6]. 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 3 shows several example topologies. In Figure 3(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 3(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 [12]. 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 3(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 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 relies on the combination of 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 4.

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 Summary of Prognosfs Experimental Results

Detailed experimental results from both local area and wide area configurations are described in [4]. Here, we only present some observations from our experiments with the topology of Figure 3(a). The main role of the bridge Stone S_b is to forward blocks from one side to the other. In addition, its behavior can be customized in at least two ways. In one case, it is programmed to "cache" any data blocks that flow through it, so that it may be able to satisfy any subsequent requests for those blocks. In another case, it is programmed to actively "distribute" blocks flowing through it among the Stones on the destination side in a round-robin fashion. These "cache" and "distribute" strategies pay the cost of replication the first time a block flows through S_b for potential benefit during subsequent accesses to that block. This illustrates the fact that the benefits of any given strategy may be highly workload- and application-dependent. Therefore, the ability to dynamically adapt the behavior of embedded storage is often

important. In some cases, it may be possible to execute these functions by issuing commands from the edges of the network, but this often incurs overheads and limits the ability to quickly adapt to the workload.

Prognosfs is an example that illustrates some of the extremely diverse customizations made possible by programmable embedded storage. The example strategies, such as those mentioned in the context of Figure 3, serve to show that a fixed interface for embedded storage may not always be sufficient. Different strategies suit different system configurations, and one needs both programmability and network-awareness of embedded storage to tailor application behavior to prevailing conditions.

4 Prototype Prognos

Resource Management and Security. The three key players in resource management are: the Stone Operating System (SOS), the application-specific 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 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. Code specific to each service is executed within its own separate address space, thus isolating it from other services running concurrently on the same Stone. The service daemons request resources from the SOS, which is currently implemented as a simple Linux user-level process. Prognos could benefit from resource accounting abstractions that are more precise than the process model, such as "resource containers" [13], but our prototype does not support such fine-grained mechanisms. More efficient alternatives than the process model, such as software-based fault isolation and safe language-based extensions, also exist. 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 and/or language-based implementations.

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 [14,15].

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.

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.

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 [16]. 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.

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[17], 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.

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 [5,6,7]. Due to lack of space, we refer the reader to [18] for further details on Canto.

5 Related Work

Many active network prototypes have been built [19,20,21,22]. 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 [23], 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 [1], 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 [24] and media transcoding [25]. 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" [26,27]. 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 [28] 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 [6,7] 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 [5,6,7]. 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.

6 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. Beck, M., Moore, T., Plank, J.S.: An End-to-End Approach to Globally Scalable Network Storage. In: Proc. of ACM SIGCOMM 2002. (2002)

2. Peterson, L., Anderson, T., Culler, D., Roscoe, T.: A Blueprint for Introducing Disruptive Technology into the Internet. In: Proc. First ACM Workshop on Hot Topics in Networking (HotNets). (2002)
3. Tridgell, A.: Efficient Algorithms for Sorting and Synchronization. PhD thesis, Australian National University (1999)
4. Sobti, S., Lai, J., Shao, Y., Garg, N., Zhang, C., Zhang, M., Zheng, F., Krishnamurthy, A., Wang, R.Y.: Network-Embedded Programmable Storage and Its Applications. Technical report, CS Dept., Princeton University (2004)
5. Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., Wang, R.: Serverless Network File Systems. ACM Transactions on Computer Systems **14** (1996)
6. Lee, E.K., Thekkath, C.E.: Petal: Distributed Virtual Disks. In: Conference on Architectural Support for Programming Languages and Operating Systems. (1996)
7. Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: A Scalable Distributed File System. In: Proc. ACM Symposium on Operating Systems Principles. (1997)
8. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-Area Cooperative Storage with CFS. In: Proc. of SOSP. (2001)
9. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: OceanStore: An Architecture for Global-Scale Persistent Storage. In: Proc. of ASPLOS. (2000)
10. Rowstron, A., Druschel, P.: Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In: Proc. of SOSP. (2001)
11. Clarke, I., Sandberg, O., Wiley, B., Hong, T.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. In: Workshop on Design Issues in Anonymity. (2000)
12. Anderson, D., Chase, J., Vahdat, A.: Interposed Request Routing for Scalable Network Storage. In: Proc. of Operating Systems Design and Implementation. (2000)
13. Banga, G., Druschel, P., Mogul, J.C.: Resource Containers: A New Facility for Resource Management In Server Systems. In: Operating Systems Design and Implementation. (1999)
14. Wobber, E., Abadi, M., Burrows, M., Lampson, B.: Authentication in the Taos operating system. ACM Transactions on Computer Systems **12** (1994) 3–32
15. Gibson, G., Nagle, D., Amiri, K., Chang, F., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., Zelenka, J.: File Server Scaling with Network-Attached Secure Disks. In: Proc. of the 1997 SIGMETRICS. (1997)
16. de Jonge, W., Kaashoek, M.F., Hsieh, W.C.: The Logical Disk: A New Approach to Improving File Systems. In: Proc. Symposium on Operating Systems Principles. (1993)
17. Zhang, M., Wang, R.Y., Peterson, L., Krishnamurthy, A.: Probabilistic Packet Scheduling: Achieving Proportional Share Bandwidth Allocation for TCP Flows. In: Proc. IEEE Infocom 2002. (2002)
18. Zhang, C., Lai, J., Garg, N., Sobti, S., Zheng, F., Krishnamurthy, A., Wang, R.: Coherent and Network-aware Tracking of Objects. Technical Report TR-672-03, CS Dept., Princeton University (2003)
19. Alexander, D.S., Shaw, M., Nettles, S., Smith, J.M.: Active Bridging. In: Proc. of ACM SIGCOMM '97. (1997) 101–111
20. Decasper, D., Dittia, Z., Parulkar, G.M., Plattner, B.: Router Plugins: A Software Architecture for Next Generation Routers. In: Proc. of ACM SIGCOMM '98. (1998)

21. Nygren, E.L., Garland, S.J., Kaashoek, M.F.: PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In: Proc. of OpenArch'99. (1999)
22. 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)
23. Nagle, D.: Active Storage Nets. http://www.ece.cmu.edu/~ asn/old/pubs/-Active%20Storage%20Nets%20Intro.pdf (1998)
24. Cao, P., Zhang, J., Beach, K.: Active Cache: Caching Dynamic Contents on the Web. In: Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing. (1998)
25. Amir, E., McCanne, S., Katz, R.H.: An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In: Proc. of ACM SIGCOMM '98. (1998)
26. Acharya, A., Uysal, M., Saltz, J.: Active Disks: Programming Model, Algorithms and Evaluation. In: Proc. of ASPLOS. (1998)
27. Riedel, E., Gibson, G.A., Faloutsos, C.: Active Storage For Large-Scale Data Mining and Multimedia. In: Proc. of International Conference on Very Large Data Bases. (1998)
28. Muthitacharoen, A., Chen, B., Mazieres, D.: A Low-bandwidth Network File System. In: Proc. ACM Symposium on Operating Systems Principles. (2001)