

Interposed Request Routing for Scalable Network Storage

Darrell C. Anderson, Jeffrey S. Chase, Amin M. Vahdat *

Department of Computer Science

Duke University

{anderson,chase,vahdat}@cs.duke.edu

Abstract. This paper explores interposed request routing in Slice, a new storage system architecture for high-speed networks incorporating network-attached block storage. Slice interposes a request switching filter — called a *μproxy* — along each client’s network path to the storage service (e.g., in a network adapter or switch). The *μproxy* intercepts request traffic and distributes it across a server ensemble. We propose request routing schemes for I/O and file service traffic, and explore their effect on service structure.

The Slice prototype uses a packet filter *μproxy* to virtualize the standard Network File System (NFS) protocol, presenting to NFS clients a unified shared file volume with scalable bandwidth and capacity. Experimental results from the industry-standard SPECsfs97 workload demonstrate that the architecture enables construction of powerful network-attached storage services by aggregating cost-effective components on a switched Gigabit Ethernet LAN.

1 Introduction

Demand for large-scale storage services is growing rapidly. A prominent factor driving this growth is the concentration of storage in data centers hosting Web-based applications that serve large client populations through the Internet. At the same time, storage demands are increasing for scalable computing, multimedia and visualization.

A successful storage system architecture must scale to meet these rapidly growing demands, placing a premium on the costs (including human costs) to administer and upgrade the system. Commercial systems increasingly interconnect storage devices and servers with dedicated Storage Area Net-

works (SANs), e.g., FibreChannel, to enable incremental scaling of bandwidth and capacity by attaching more storage to the network. Recent advances in LAN performance have narrowed the bandwidth gap between SANs and LANs, creating an opportunity to take a similar approach using a general-purpose LAN as the storage backplane. A key challenge is to devise a distributed software layer to unify the decentralized storage resources.

This paper explores *interposed request routing* in Slice, a new architecture for network storage. Slice interposes a request switching filter — called a *μproxy* — along each client’s network path to the storage service. The *μproxy* may reside in a programmable switch or network adapter, or in a self-contained module at the client’s or server’s interface to the network. We show how a simple *μproxy* can virtualize a standard network-attached storage protocol incorporating file services as well as raw device access. The Slice *μproxy* distributes request traffic across a collection of storage and server elements that cooperate to present a uniform view of a shared file volume with scalable bandwidth and capacity.

This paper makes the following contributions:

- It outlines the architecture and its implementation in the Slice prototype, which is based on a *μproxy* implemented as an IP packet filter. We explore the impact on service structure, reconfiguration, and recovery.
- It proposes and evaluates request routing policies within the architecture. In particular, we introduce two policies for transparent scaling of the name space of a unified file volume. These techniques complement simple grouping and striping policies to distribute file access load.
- It evaluates the prototype using synthetic benchmarks including SPECsfs97, an industry-standard workload for network-attached storage servers. The results demonstrate that the

*This work is supported by the National Science Foundation (EIA-9972879 and EIA-9870724), Intel, and Myricom. Anderson is supported by a U.S. Department of Education GAANN fellowship. Chase and Vahdat are supported by NSF CAREER awards (CCR-9624857 and CCR-9984328).

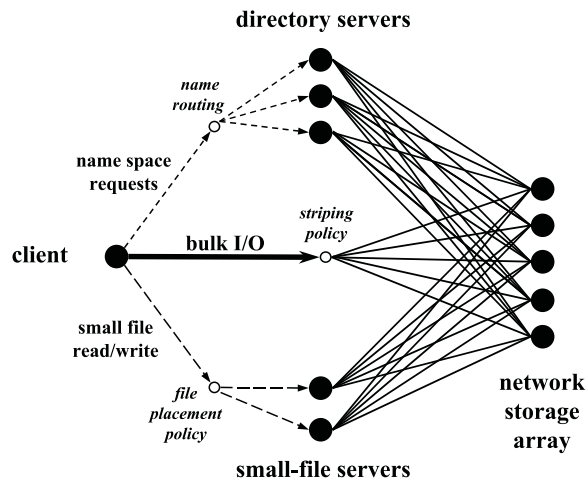


Figure 1: Combining functional decomposition and data decomposition in the Slice architecture.

system is scalable and that it complies with the Network File System (NFS) V3 standard, a popular protocol for network-attached storage.

This paper is organized as follows. Section 2 outlines the architecture and sets Slice in context with related work. Section 3 discusses the role of the μ proxy, defines the request routing policies, and discusses service structure. Section 4 describes the Slice prototype, and Section 5 presents experimental results. Section 6 concludes.

2 Overview

The Slice file service consists of a collection of servers cooperating to serve an arbitrarily large *virtual volume* of files and directories. To a client, the ensemble appears as a single file server at some virtual network address. The μ proxy intercepts and transforms packets to redirect requests and to represent the ensemble as a unified file service.

Figure 1 depicts the structure of a Slice ensemble. Each client’s request stream is partitioned into three functional request classes corresponding to the major file workload components: (1) high-volume I/O to large files, (2) I/O on small files, and (3) operations on the name space or file attributes. The μ proxy switches on the request type and arguments to redirect requests to a selected server responsible for handling a given class of requests. Bulk I/O operations route directly to an array of *storage nodes*, which provide block-level access to raw storage objects. Other operations are distributed among specialized file managers responsible for small-file I/O

and/or name space requests.

This *functional decomposition* diverts high-volume data flow to bypass the managers, while allowing specialization of the servers for each workload component, e.g., by tailoring the policies for disk layout, caching and recovery. A single server node could combine the functions of multiple server classes; we separate them to highlight the opportunities to distribute requests across more servers.

The μ proxy selects a target server by switching on the request type and the identity of the target file, name entry, or block, using a separate routing function for each request class. Thus the routing functions induce a *data decomposition* of the volume data across the ensemble, with the side effect of creating or caching data items on the selected managers. Ideally, the request routing scheme spreads the data and request workload in a balanced fashion across all servers. The routing functions may adapt to system conditions, e.g., to use new server sites as they become available. This allows each workload component to scale independently by adding resources to its server class.

2.1 The μ proxy

An overarching goal is to keep the μ proxy simple, small, and fast. The μ proxy may (1) rewrite the source address, destination address, or other fields of request or response packets, (2) maintain a bounded amount of soft state, and (3) initiate or absorb packets to or from the Slice ensemble. The μ proxy does not require any state that is shared across clients, so it may reside on the client host or network interface, or in a network element close to the server ensemble. The μ proxy is not a barrier to scalability because its functions are freely replicable, with the constraint that each client’s request stream passes through a single μ proxy.

The μ proxy functions as a network element within the Internet architecture. It is free to discard its state and/or pending packets without compromising correctness. End-to-end protocols (in this case NFS/RPC/UDP or TCP) retransmit packets as necessary to recover from drops in the μ proxy. Although the μ proxy resides “within the network”, it acts as an extension of the service. For example, since the μ proxy is a layer-5 protocol component, it must reside (logically) at one end of the connection or the other; it cannot reside in the “middle” of the connection where end-to-end encryption might hide layer-5 protocol fields.

2.2 Network Storage Nodes

A shared array of network storage nodes provides all disk storage used in a Slice ensemble. The μ proxy routes bulk I/O requests directly to the network storage array, without intervention by a file manager. More storage nodes may be added to incrementally scale bandwidth, capacity, and disk arms.

The Slice block storage prototype is loosely based on a proposal in the National Storage Industry Consortium (NSIC) for object-based storage devices (OBSD) [3]. Key elements of the OBSD proposal were in turn inspired by the CMU research on Network Attached Secure Disks (NASD) [8, 9]. Slice storage nodes are “object-based” rather than sector-based, meaning that requesters address data as logical offsets within *storage objects*. A storage object is an ordered sequence of bytes with a unique identifier. The placement policies of the file service are responsible for distributing data among storage objects so as to benefit fully from all of the resources in the network storage array.

A key advantage of OBSDs and NASDs is that they allow for cryptographic protection of storage object identifiers if the network is insecure [9]. This protection allows the μ proxy to reside outside of the server ensemble’s trust boundary. In this case, the damage from a compromised μ proxy is limited to the files and directories that its client(s) had permission to access. However, the Slice request routing architecture is compatible with conventional sector-based storage devices if every μ proxy resides inside the service trust boundary.

This storage architecture is orthogonal to the question of which level arranges redundancy to tolerate disk failures. One alternative is to provide redundancy of disks and other vulnerable components internally to each storage node. A second option is for the file service software to mirror data or maintain parity across the storage nodes. In Slice, the choice to employ extra redundancy across storage nodes may be made on a per-file basis through support for mirrored striping in our prototype’s I/O routing policies. For stronger protection, a Slice configuration could employ redundancy at both levels.

The Slice block service includes a *coordinator* module for files that span multiple storage nodes. The coordinator manages optional block maps (Section 3.1) and preserves atomicity of multisite operations (Section 3.3.2). A Slice configuration may include any number of coordinators, each managing a subset of the files (Section 4.2).

2.3 File Managers

File management functions above the network storage array are split across two classes of file managers. Each class governs functions that are common to any file server; the architecture separates them to distribute the request load and allow implementations specialized for each request class.

- *Directory servers* handle name space operations, e.g., to *create*, *remove*, or *lookup* files and directories by symbolic name; they manage directories and mappings from names to identifiers and attributes for each file or directory.
- *Small-file servers* handle *read* and *write* operations on small files and the initial segments of large files (Section 3.1).

Slice file managers are *dataless*; all of their state is backed by the network storage array. Their role is to aggregate their structures into larger storage objects backed by the storage nodes, and to provide memory and CPU resources to cache and manipulate those structures. In this way, the file managers can benefit from the parallel disk arms and high bandwidth of the storage array as more storage nodes are added.

The principle of dataless file managers also plays a key role in recovery. In addition to its backing objects, each manager journals its updates in a write-ahead log [10]; the system can recover the state of any manager from its backing objects together with its log. This allows fast failover, in which a surviving site assumes the role of a failed server, recovering its state from shared storage [12, 4, 24].

2.4 Summary

Interposed request routing in the Slice architecture yields three fundamental benefits:

- *Scalable file management with content-based request switching.* Slice distributes file service requests across a server ensemble. A good request switching scheme induces a balanced distribution of file objects and requests across servers, and improves locality in the request stream.
- *Direct storage access for high-volume I/O.* The μ proxy routes bulk I/O traffic directly to the network storage array, removing the file managers from the critical path. Separating requests in this fashion eliminates a key scaling barrier for conventional file services [8, 9]. At the same time, the small-file servers absorb and

aggregate I/O operations on small files, so there is no need for the storage nodes to handle small objects efficiently.

- *Compatibility with standard file system clients.* The μ proxy factors request routing policies out of the client-side file system code. This allows the architecture to leverage a minimal computing capability within the network elements to virtualize the storage protocol.

2.5 Related Work

A large number of systems have interposed new system functionality by “wrapping” an existing interface, including kernel system calls [14], internal interfaces [13], communication bindings [11], or messaging endpoints. The concept of a *proxy* mediating between clients and servers [23] is now common in distributed systems. We propose to mediate some storage functions by interposing on standard storage access protocols within the network elements. Network file services can benefit from this technique because they have well-defined protocols and a large installed base of clients and applications, many of which face significant scaling challenges today.

The Slice μ proxy routes file service requests based on their content. This is analogous to the HTTP content switching features offered by some network switch vendors (e.g., Alteon, Arrowpoint, F5), based in part on research demonstrating improved locality and load balancing for large Internet server sites [20]. Slice extends the content switching concept to a file system context.

A number of recent commercial and research efforts investigate techniques for building scalable storage systems for high-speed switched LAN networks. These systems are built from disks distributed through the network, and attached to dedicated servers [16, 24, 12], cooperating peers [4, 26], or the network itself [8, 9]. We separate these systems into two broad groups.

The first group separates file managers (e.g., the name service) from the block storage service, as in Slice. This separation was first proposed for the Cambridge Universal File Server [6]. Subsequent systems adopted this separation to allow bulk I/O to bypass file managers [7, 12], and it is now a basic tenet of research in network-attached storage devices including the CMU NASD work on devices for secure storage objects [8, 9]. Slice shows how to incorporate placement and routing functions essential for this separation into a new filesystem structure for network-attached storage. The CMU NASD

project integrated similar functions into network file system clients [9]; the Slice model decouples these functions, preserving compatibility with existing clients. In addition, Slice extends the NASD project approach to support scalable file management as well as high-bandwidth I/O for large files.

A second group of scalable storage systems layers the file system functions above a network storage volume using a *shared disk* model. Policies for striping, redundancy, and storage site selection are specified on a volume basis; cluster nodes coordinate their accesses to the shared storage blocks using an ownership protocol. This approach has been used with both log-structured (Zebra [12] and xFS [4]) and conventional (Frangipani/Petal [16, 24] and GFS [21]) file system organizations. The cluster may be viewed as “serverless” if all nodes are trusted and have direct access to the shared disk, or alternatively the entire cluster may act as a file server to untrusted clients using a standard network file protocol, with all I/O passing through the cluster nodes as they mediate access to the disks.

The key benefits of Slice request routing apply equally to these shared disk systems when untrusted clients are present. First, request routing is a key to incorporating secure network-attached block storage, which allows untrusted clients to address storage objects directly without compromising the integrity of the file system. That is, a μ proxy could route bulk I/O requests directly to the devices, yielding a more scalable system that preserves compatibility with standard clients and allows per-file policies for block placement, parity or replication, prefetching, etc. Second, request routing enhances locality in the request stream to the file servers, improving cache effectiveness and reducing block contention among the servers.

The shared disk model is used in many commercial systems, which increasingly interconnect storage devices and servers with dedicated Storage Area Networks (SANs), e.g., FibreChannel. This paper explores storage request routing for Internet networks, but the concepts are equally applicable in SANs.

Our proposal to separate small-file I/O from the request stream is similar in concept to the Amoeba Bullet Server [25], a specialized file server that optimizes small files. As described in Section 4.4, the prototype small-file server draws on techniques from the Bullet Server, FFS fragments [19], and SquidMLA [18], a Web proxy server that maintains a user-level “filesystem” of small cached Web pages.

3 Request Routing Policies

This section explains the structure of the μ proxy and the request routing schemes used in the Slice prototype. The purpose is to illustrate concretely the request routing policies enabled by the architecture, and the implications of those policies for the way the servers interact to maintain and recover consistent file system states. We use the NFS V3 protocol as a reference point because it is widely understood and our prototype supports it.

The μ proxy intercepts NFS requests addressed to virtual NFS servers, and routes the request to a physical server by applying a function to the request type and arguments. It then rewrites the IP address and port to redirect the request to the selected server. When a response arrives, the μ proxy rewrites the source address and port before forwarding it to the client, so the response appears to originate from the virtual NFS server.

The request routing functions must permit reconfiguration to add or remove servers, while minimizing state requirements in the μ proxy. The μ proxy directs most requests by extracting relevant fields from the request, perhaps hashing to combine multiple fields, and interpreting the result as a logical server site ID for the request. It then looks up the corresponding physical server in a compact routing table. Multiple logical sites may map to the same physical server, leaving flexibility for reconfiguration (Section 3.3.1). The routing tables constitute soft state; the mapping is determined externally, so the μ proxy never modifies the tables.

The μ proxy examines up to four fields of each request, depending on the policies configured:

- *Request type.* Routing policies are keyed by the NFS request type, so the μ proxy may employ different policies for different functions. Table 1 lists the important NFS request groupings discussed in this paper.
- *File handle.* Each NFS request targets a specific file or directory, named by a unique identifier called a *file handle* (or *fhandle*). Although NFS fhandles are opaque to the client, their structure can be known to the μ proxy, which acts as an extension of the service. Directory servers encode a *fileID* in each fhandle, which the μ proxies extract as a routing key.
- *Read/write offset.* NFS I/O operations specify the range of offsets covered by each *read* and

write. The μ proxy uses these fields to select the server or storage node for the data.

- *Name component.* NFS name space requests include a symbolic name component in their arguments (see Table 1). A key challenge for scaling file management is to obtain a balanced distribution of these requests. This is particularly important for *name-intensive* workloads with small files and heavy *create/lookup/remove* activity, as often occurs in Internet services for mail, news, message boards, and Web access.

We now outline some μ proxy policies that use these fields to route specific request groups.

3.1 Block I/O

Request routing for *read/write* requests have two goals: separate small-file *read/write* traffic from bulk I/O, and decluster the blocks of large files across the storage nodes for the desired access properties (e.g., high bandwidth or a specified level of redundancy). We address each in turn.

When small-file servers are configured, the prototype's routing policy defines a fixed *threshold offset* (e.g., 64KB); the μ proxy directs I/O requests below the threshold to a small-file server selected from the request fhandle. The threshold offset is necessary because the size of each file may change at any time. Thus the small-file servers also receive a subset of the I/O requests on large files; they receive *all* I/O below the threshold, even if the target file is large. In practice, large files have little impact on the small-file servers because there tends to be a small number of these files, even if they make up a large share of the stored bytes. Similarly, large file I/O below the threshold is limited by the bandwidth of the small-file server, but this affects only the first *threshold* bytes, and becomes progressively less significant as the file grows.

The μ proxy redirects I/O traffic above the threshold directly to the network storage array, using some placement policy to select the storage site(s) for each block. A simple option is to employ static striping and placement functions that compute on the block offset and/or fileID. More flexible placement policies would allow the μ proxy to consider other factors, e.g., load conditions on the network or storage nodes, or file attributes encoded in the fhandle. To generalize to more flexible placement policies, Slice optionally records block locations in per-file block maps managed by the block service coordinators. The μ proxies interact with the coordinators

Name Space Operations	
<i>lookup</i> (<i>dir</i> , <i>name</i>) returns (fhandle, attr)	Look up a name in <i>dir</i> ; return handle and attributes.
<i>create</i> (<i>dir</i> , <i>name</i>) returns (fhandle, attr)	Create a file/directory and update the parent entry/link count and modify timestamp.
<i>mkdir</i> (<i>dir</i> , <i>name</i>) returns (fhandle, attr)	
<i>remove</i> (<i>dir</i> , <i>name</i>), <i>rmdir</i> (<i>dir</i> , <i>name</i>)	Remove a file/directory or hard link and update the parent entry/link count and modify timestamp.
<i>link</i> (<i>olddir</i> , <i>oldname</i> , <i>newdir</i> , <i>newname</i>) returns (fhandle, attr)	Create a new name for a file, update the file link count, and update modify timestamps on the file and <i>newdir</i> .
<i>rename</i> (<i>olddir</i> , <i>oldname</i> , <i>newdir</i> , <i>newname</i>) returns (fhandle, attr)	Rename an existing file or hard link; update the link count and modify timestamp on both the old and new parent.
Attribute Operations	
<i>getattr</i> (<i>object</i>) returns (attr)	Retrieve the attributes of a file or directory.
<i>setattr</i> (<i>object</i> , <i>attr</i>)	Modify the attributes of a file or directory, and update its modify timestamp.
I/O Operations	
<i>read</i> (<i>file</i> , <i>offset</i> , <i>len</i>) returns (data, attr)	Read data from a file, updating its access timestamp.
<i>write</i> (<i>file</i> , <i>offset</i> , <i>len</i>) returns (data, attr)	Write data to a file, updating its modify timestamp.
Directory Retrieval	
<i>readdir</i> (<i>dir</i> , <i>cookie</i>) returns (entries, cookie)	Read some or all of the entries in a directory.

Table 1: Some important Network File System (NFS) protocol operations.

to fetch and cache fragments of the block maps as they handle I/O operations on files.

As one example of an attribute-based policy, Slice supports a *mirrored striping* policy that replicates each block of a mirrored file on multiple storage nodes, to tolerate failures up to the replication degree. Mirroring consumes more storage and network bandwidth than striping with parity, but it is simple and reliable, avoids the overhead of computing and updating parity, and allows load-balanced reads [5, 16].

3.2 Name Space Operations

Effectively distributing name space requests presents different challenges from I/O request routing. Name operations involve more computation, and name entries may benefit more from caching because they tend to be relatively small and fragmented. Moreover, directories are frequently shared. Directory servers act as synchronization points to preserve integrity of the name space, e.g., to prevent clients from concurrently creating a file with the same name, or removing a directory while a name create is in progress.

A simple approach to scaling a file service is to partition the name space into a set of volumes, each managed by a single server. Unfortunately, this VOLUME PARTITIONING strategy compromises transparency and increases administrative overhead in two ways. First, volume boundaries are visible to clients as *mount points*, and naming operations such as *link* and *rename* cannot cross volume boundaries. Sec-

ond, the system develops imbalances if volume loads grow at different rates, requiring intervention to repartition the name space. This may be visible to users through name changes to existing directories.

An important goal of name management in Slice is to automatically distribute the load of a single file volume across multiple servers, without imposing user-visible volume boundaries. We propose two alternative name space routing policies to achieve this goal. MKDIR SWITCHING yields balanced distributions when the average number of active directories is large relative to the number of directory server sites, but it binds large directories to a single server. For workloads with very large directories, NAME HASHING yields probabilistically balanced request distributions independent of workload. The cost of this effectiveness is that more operations cross server boundaries, increasing the cost and complexity of coordination among the directory servers (Section 4.3).

MKDIR SWITCHING works as follows. In most cases, the μ proxy routes name space operations to the directory server that manages the parent directory; the μ proxy identifies this server by indexing its routing table with the fileID from the parent directory fhandle in the request (refer to Table 1). On a *mkdir* request, the μ proxy decides with probability p to redirect the request to a different directory server, placing the new directory — and its descendents — on a different site from the parent directory. The policy uniquely selects the new server by hashing on the parent fhandle and the symbolic name of the

new directory; this guarantees that races over name manipulation involve at most two sites. Reducing directory affinity by increasing p makes the policy more aggressive in distributing name entries across sites; this produces a more balanced load, but more operations involve multiple sites. Section 5 presents experimental data illustrating this tradeoff.

NAME HASHING extends this approach by routing *all* name space operations using a hash on the name component and its position in the directory tree, as given by the parent directory fhandle. This approach represents the entire volume name space as a unified global hash table distributed among the directory servers. It views directories as distributed collections of name entries, rather than as files accessed as a unit. Conflicting operations on any given name entry (e.g., *create/create*, *create/remove*, *remove/lookup*) always hash to the same server, where they serialize on the shared hash chain. Operations on different entries in the same directory (e.g., *create*, *remove*, *lookup*) may proceed in parallel at multiple sites. For good performance, NAME HASHING requires sufficient memory to keep the hash chains memory-resident, since the hashing function sacrifices locality in the hash chain accesses. Also, *read-dir* operations span multiple sites; this is the right behavior for large directories, but it increases *read-dir* costs for small directories.

3.3 Storage Service Structure

Request routing policies impact storage service structure. The primary challenges are coordination and recovery to maintain a consistent view of the file volume across all servers, and reconfiguration to add or remove servers within each class.

Most of the routing policies outlined above are independent of whether small files and name entries are bound to the server sites that create them. One option is for the servers to share backing objects from a *shared disk* using a block ownership protocol (see Section 2.5); in this case, the role of the μ proxy is to enhance locality in the request stream to each server. Alternatively, the system may use *fixed placement* in which items are controlled by their create sites unless reconfiguration or failover causes them to move; with this approach backing storage objects may be private to each site, even if they reside on shared network storage. Fixed placement stresses the role of the request routing policy in the placement of new name entries or data items. The next two subsections discuss reconfiguration and recovery issues for the Slice architecture with respect to these structural alternatives.

3.3.1 Reconfiguration

Consider the problem of reconfiguration to add or remove file managers, i.e., directory servers, small-file servers, or map coordinators. For requests routed by keying on the fileID, the system updates μ proxy routing tables to change the binding from fileIDs to physical servers if servers join or depart the ensemble. To keep the tables compact, Slice maps the fileID to a smaller logical server ID before indexing the table. The number of logical servers defines the size of the routing tables and the minimal granularity for rebalancing. The μ proxy’s copy of the routing table is a “hint” that may become stale during reconfiguration; the μ proxy may load new tables lazily from an external source, assuming that servers can identify misdirected requests.

This approach generalizes to policies in which the logical server ID is derived from a hash that includes other request arguments, as in the NAME HASHING approach. For NAME HASHING systems and other systems with fixed placement, the reconfiguration procedure must move logical servers from one physical server to another. One approach is for each physical server to use multiple backing objects, one for each hosted logical server, and reconfigure by reassigning the binding of physical servers to backing objects in the shared network storage array. Otherwise, reconfiguration must copy data from one backing object to another. In general, an ensemble with N servers must move $1/N$ th of its data to rebalance after adding or losing a physical server [15].

3.3.2 Atomicity and Recovery

File systems have strong integrity requirements and frequent updates; the system must preserve their integrity through failures and concurrent operations. The focus on request routing naturally implies that the multiple servers must manage distributed state.

File managers prepare for recovery by generating a write-ahead log in shared storage. For systems that use the shared-disk model without fixed placement, all operations execute at a single manager site, and it is necessary and sufficient for the system to provide locking and recovery procedures for the shared disk blocks [24]. For systems with fixed placement, servers do not share blocks directly, but some operations must update state at multiple sites through a peer-peer protocol. Thus there is no need for distributed locking or recovery of individual blocks, but the system must coordinate logging and recovery across sites, e.g., using two-phase commit.

For MKDIR SWITCHING, the operations that update multiple sites are those involving the “orphaned” directories that were placed on different sites from their parents. These operations include the redirected *mkdirs* themselves, associated *rmdirs*, and any *rename* operations involving the orphaned entries. Since these operations are relatively infrequent, as determined by the redirection probability parameter p , it is acceptable to perform a full two-phase commit as needed to guarantee their atomicity on systems with fixed placement. However, NAME HASHING requires fixed placement — unless the directory servers support fine-grained distributed caching — and any name space update involves multiple sites with probability $(N - 1)/N$ or higher. While it is possible to reduce commit costs by logging asynchronously and coordinating rollback, this approach weakens failure properties because recently completed operations may be lost in a failure.

Shared network storage arrays present their own atomicity and recovery challenges. In Slice, the block service coordinators preserve atomicity of operations involving multiple storage nodes, including mirrored striping, *truncate/remove*, and NFS V3 write commitment (*commit*). Amiri et al. [1] addresses atomicity and concurrency control issues for shared storage arrays; the Slice coordinator protocol complements [1] with an intention logging protocol for atomic filesystem operations [2]. The basic protocol is as follows. At the start of the operation, the μ proxy sends to the coordinator an *intention* to perform the operation. The coordinator logs the intention to stable storage. When the operation completes, the μ proxy notifies the coordinator with a *completion* message, asynchronously clearing the intention. If the coordinator does not receive the completion within some time bound, it probes the participants to determine if the operation completed, and initiates recovery if necessary. A failed coordinator recovers by scanning its intentions log, completing or aborting operations in progress at the time of the failure. In practice, the protocol eliminates some message exchanges and log writes from the critical path of most common-case operations by piggybacking messages, leveraging the NFS V3 *commit* semantics, and amortizing intention logging costs across multiple operations.

4 Implementation

The Slice prototype is a set of loadable kernel modules for the FreeBSD operating system. The prototype includes a μ proxy implemented as a packet

filter below the Internet Protocol (IP) stack, and kernel modules for the basic server classes: block storage service and block storage coordinator, directory server, and small-file server. A given server node may be configured for any subset of the Slice server functions, and each function may be present at an arbitrary number of nodes. The following subsections discuss each element of the Slice prototype in more detail.

4.1 The μ proxy

The Slice μ proxy is a loadable packet filter module that intercepts packets exchanged with registered NFS virtual server endpoints. The module is configurable to run as an intermediary at any point in the network between a client and the server ensemble, preserving compatibility with NFS clients. Our premise is that the functions of the μ proxy are simple enough to integrate more tightly with the network switching elements, enabling wire-speed request routing. The μ proxy may also be configured below the IP stack on each client node, to avoid the store-and-forward delays imposed by host-based intermediaries in our prototype.

The μ proxy is a nonblocking state machine with soft state consisting of pending request records and routing tables for I/O redirection, MKDIR SWITCHING, and NAME HASHING, as described in Section 3. The prototype statically configures the policies and table sizes for name space operations and small-file I/O; it does not yet detect and refresh stale routing tables for reconfiguration. These policies use the MD5 [22] hash function; we determined empirically that MD5 yields a combination of balanced distribution and low cost that is superior to competing hash functions available to us. For *reads* and *writes* beyond the threshold offset the μ proxy may use either a static block placement policy or a local cache of per-file block maps supplied by a block service coordinator (see Section 4.2).

The μ proxy also maintains a cache over file attribute blocks returned in NFS responses from the servers. Directory servers maintain the authoritative attributes for files; the system must keep these attributes current to reflect I/O traffic to the block storage nodes, which affects the modify time, access time, and/or size attributes of the target file. The μ proxy updates these attributes in its cache as each operation completes, and returns a complete set of attributes to the client in each response (some clients depend on this behavior, although the NFS specification does not require it). The μ proxy generates an NFS *setattr* operation to push modified at-

tributes back to the directory server when it evicts attributes from its cache, or when it intercepts an NFS V3 write *commit* request from the client. Most clients issue *commit* requests for modified files from a periodic system update daemon, and when a user process calls *fsync* or *close* on a modified file.

The prototype may yield weaker attribute consistency than some NFS implementations. First, attribute timestamps are no longer assigned at a central site; we rely on the Network Time Protocol (NTP) to keep clocks synchronized across the system. Most NFS installations already use NTP to allow consistent assignment and interpretation of timestamps across multiple servers and clients. Second, a *read* or an uncommitted *write* is not guaranteed to update the attribute timestamps if the μ proxy fails and loses its state. In the worst case an uncommitted *write* might complete at a storage node but not affect the modify time at all (if the client also fails before reissuing the write). The NFS V3 specification permits this behavior: uncommitted writes may affect any subset of the modified data or attributes. Third, although the attribute timestamps cached and returned by each μ proxy are always current with respect to operations from clients bound to that μ proxy, they may drift beyond the “three second window” that is the de facto standard in NFS implementations for concurrently shared files. We consider this to be acceptable since NFS V3 offers no firm consistency guarantees for concurrently shared files anyway. Note, however, that NFS V4 proposes to support consistent file sharing through a leasing mechanism similar to NQ-NFS [17]; it will then be sufficient for the μ proxy to propagate file attributes when a client renews or relinquishes a lease for the file. The current μ proxy bounds the drift by writing back modified attributes at regular intervals.

Since the μ proxy modifies the contents of request and response packets, it must update the UDP or TCP checksums to match the new packet data. The prototype μ proxy recomputes checksums incrementally, generalizing a technique used in other packet rewriting systems. The μ proxy’s differential checksum code is derived from the FreeBSD implementation of Network Address Translation (NAT). The cost of incremental checksum adjustment is proportional to the number of modified bytes and is independent of the total size of the message. It is efficient because the μ proxy rewrites at most the source or destination address and port number, and in some cases certain fields of the file attributes.

4.2 Block Storage Service

The Slice block storage servers use a kernel module that exports disks to the network. The storage nodes serve a flat space of storage objects named by unique identifiers; storage is addressed by (*object, logicalblock*), with physical allocation controlled by the storage node software as described in Section 2.2. The key operations are a subset of NFS, including *read*, *write*, *commit*, and *remove*. The storage nodes accept NFS file handles as object identifiers, using an external hash to map them to storage objects. Our current prototype uses the Fast File System (FFS) as a storage manager within each storage node. The storage nodes prefetch sequential files up to 256 KB beyond the current access, and also leverage FFS write clustering.

The block storage service includes a coordinator implemented as an extension to the storage node module. Each coordinator manages a set of files, selected by fileID. The coordinator maintains optional per-file block maps giving the storage site for each logical block of the file; these maps are used for dynamic I/O routing policies (Section 3.1). The coordinator also implements the intention logging protocol to preserve failure atomicity for file accesses involving multiple storage sites (Section 3.3.2), including remove/truncate, consistent write commitment, and mirrored writes, as described in [2]. The coordinator backs its intentions log and block maps within the block storage service using a static placement function. A more failure-resilient implementation would employ redundancy across storage nodes.

4.3 Directory Servers

Our directory server implementations use fixed placement and support both the NAME HASHING and MKDIR SWITCHING policies. The directory servers store directory information as webs of linked fixed-size cells representing name entries and file attributes, allocated from memory zones backed by the block storage service. These cells are indexed by hash chains keyed by an MD5 hash fingerprint on the parent file handle and name. The directory servers place keys in each newly minted file handle, allowing them to locate any resident cell if presented with an fhandle or an (*fhandle, name*) pair. Attribute cells may include a remote key to reference an entry on another server, enabling cross-site links in the directory structure. Thus the name entries and attribute cells for a directory may be distributed arbitrarily across the servers, making it possible to support both NAME HASHING and MKDIR SWITCHING policies easily within the same code base.

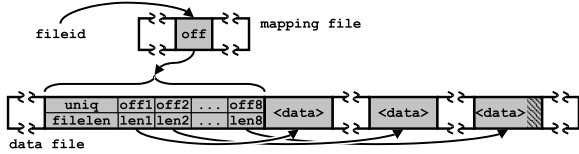


Figure 2: Small-file server data structures.

Given the distribution of entries across directory servers, some NFS operations involve multiple sites. The μ proxy interacts with a single site for each request. Directory servers use a simple peer-peer protocol to update link counts for *create/link/remove* and *mkdir/rmdir* operations that cross sites, and to follow cross-site links for *lookup*, *getattr/setattr*, and *readdir*. For NAME HASHING we implemented *rename* as a *link* followed by a *remove*.

Support for recovery and reconfiguration is incomplete in our prototype. Directory servers log their updates, but the recovery procedure itself is not implemented, nor is the support for shifting ownership of blocks and cells across servers.

4.4 Small-file Servers

The small-file server is implemented by a module that manages each file as a sequence of 8KB logical blocks. Figure 2 illustrates the key data structures and their use for a *read* or *write* request. The locations for each block are given by a per-file map record. The server accesses this record by indexing an on-disk map descriptor array using the fileID from the fhandle. Like the directory server, storage for small-file data is allocated from zones backed by objects in the block storage service.

Each map record gives a fixed number of (*offset,length*) pairs mapping 8KB file extents to regions within a backing object. Each logical block may have less than the full 8KB of physical space allocated for it; physical storage for a block rounds the space required up to the next power of two to simplify space management. New files or writes to empty segments are allocated space according to best fit, or if no good fragment is free, a new region is allocated at the end of the backing storage object. The best-fit variable fragment approach is similar to SquidMLA [18].

This structure allows efficient space allocation and supports file growth. For example, a 8300 byte file would consume only 8320 bytes of physical storage space, 8192 bytes for the first block, and 128 for the remaining 108 bytes. Under a create-heavy workload, the small-file allocation policy lays out data on

backing objects sequentially, batching newly created files into a single stream for efficient disk writes. The small-file servers comply with the NFS V3 *commit* specification for writes below the threshold offset.

Map records and data from the small-file server backing objects are cached in the kernel file buffer cache. This structure performs well if file accesses and the assignment of fileIDs show good locality. In particular, if the directory servers assign fileIDs with good spatial locality, and if files created together are accessed together, then the cost of reading the map records is amortized across multiple files whose records fit in a single block.

5 Performance

This section presents experimental results from the Slice prototype to show the overheads and scaling properties of the interposed request routing architecture. We use synthetic benchmarks to stress different aspects of the system, then evaluate whole-system performance using the industry-standard SPECsfs97 workload.

The storage nodes for the test ensemble are Dell PowerEdge 4400s with a 733 MHz Pentium-III Xeon CPU, 256MB RAM, and a ServerWorks LE chipset. Each storage node has eight 18GB Seagate Cheetah drives (ST318404LC) connected to a dual-channel Ultra-160 SCSI controller. Servers and clients are 450 MHz Pentium-III PCs with 512MB RAM and Asus P2B motherboards using a 440BX chipset. The machines are linked by a Gigabit Ethernet network with Alteon ACEnic 710025 adapters and a 32-port Extreme Summit-7i switch. The switch and adapters use 9KB (“Jumbo”) frames; the adapters run locally modified firmware that supports header splitting for NFS traffic. The adapters occupy a 64-bit/66 MHz PCI slot on the Dell 4400s, and a 32-bit/33 MHz PCI slot on the PCs. All kernels are built from the same FreeBSD 4.0 source pool.

	single client	saturation
read	62.5 MB/s	437 MB/s
write	38.9 MB/s	479 MB/s
read-mirrored	52.9 MB/s	222 MB/s
write-mirrored	32.2 MB/s	251 MB/s

Table 2: Bulk I/O bandwidth in the test ensemble.

Read/write performance. Table 2 shows raw read and write bandwidth for large files. Each test (*dd*) issues *read* or *write* system calls on a 1.25 GB file in a Slice volume mounted with a 32KB NFS

block size and a read-ahead depth of four blocks. The μ proxies use a static I/O routing function to stripe large-file data across the storage array. We measure sequential access bandwidth for unmirrored files and mirrored files with two replicas.

The left column of Table 2 shows the I/O bandwidth driven by a single PC client. Writes saturate the client CPU below 40 MB/s, the maximum bandwidth achievable through the FreeBSD NFS/UDP client stack in this configuration. We modified the FreeBSD client for zero-copy reading, allowing higher bandwidth with lower CPU utilization; in this case, performance is limited by a prefetch depth bound in FreeBSD. Mirroring degrades read bandwidth because the client μ proxies alternate between the two mirrors to balance the load, leaving some prefetched data unused on the storage nodes. Mirroring degrades write bandwidth because the client host writes to both mirrors.

The right column of Table 2 shows the aggregate bandwidth delivered to eight clients, saturating the storage node I/O systems. Each storage node sources reads to the network at 55 MB/s and sinks writes at 60 MB/s. While the Cheetah drives each yield 33 MB/s of raw bandwidth, achievable disk bandwidth is below 75 MB/s per node because the 4400 backplane has a single SCSI channel for all of its internal drive bays, and the FreeBSD 4.0 driver runs the channel in Ultra-2 mode because it does not yet support Ultra-160.

Operation	CPU
Packet interception	0.7%
Packet decode	4.1%
Redirection/rewriting	0.5%
Soft state logic	0.8%

Table 3: μ proxy CPU cost for 6250 packets/second.

Overhead of the μ proxy. The interposed request routing architecture is sensitive to the costs to intercept and redirect file service protocol packets. Table 3 summarizes the CPU overheads for a client-based μ proxy under a synthetic benchmark that stresses name space operations, which place the highest per-packet loads on the μ proxy. The benchmark repeatedly unpacks (*untar*) a set of zero-length files in a directory tree that mimics the FreeBSD source distribution. Each file create generates seven NFS operations: *lookup*, *access*, *create*, *getattr*, *lookup*, *setattr*, *setattr*. We used *iprobe* (Instruction Probe), an on-line profiling tool for Alpha-based systems, to measure the μ proxy CPU cost on

a 500 MHz Compaq 21264 client (4MB L2). This *untar* workload generates mixed NFS traffic at a rate of 3125 request/response pairs per second.

The client spends 6.1% of its CPU cycles in the μ proxy. Redirection replaces the packet destination and/or ports and restores the checksum as described in Section 4.1, consuming a modest 0.5% of CPU time. The cost of managing soft state for attribute updates and response pairing accounts for 0.8%. The most significant cost is the 4.1% of CPU time spent decoding the packets to prepare for rewriting. Nearly half of the cost is to locate the offsets of the NFS request type and arguments; NFS V3 and ONC RPC headers each include variable-length fields (e.g., access groups and the NFS V3 file handle) that increase the decoding overhead. Minor protocol changes could reduce this complexity. While this complexity affects the cost to implement the μ proxy in network elements, it does not limit the scalability of the Slice architecture.

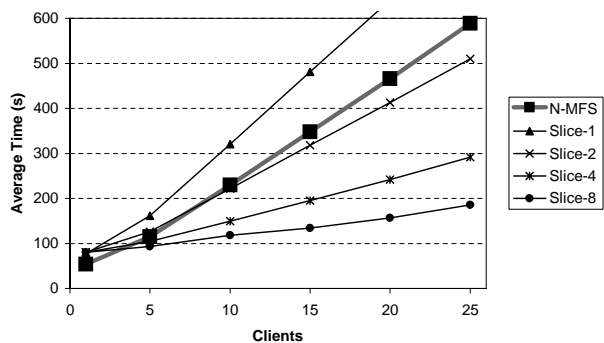


Figure 3: Directory service scaling.

Directory service scaling. We used the name-intensive *untar* benchmark to evaluate scalability of the prototype directory service using the NAME HASHING and MKDIR SWITCHING policies. In this test, a variable number of client processes execute the *untar* benchmark on five client PCs. Each process creates 36,000 files and directories generating a total of 250,000 NFS operations. For MKDIR SWITCHING we chose the most aggressive $p = 1$, i.e., the μ proxy hashes to select a target for every *mkdir* request to distribute the directories across the N server sites. For this experiment, in which the name space spans many directories, aggressive MKDIR SWITCHING and NAME HASHING perform identically.

Figure 3 shows the average total latency perceived by each client process as a function of the number of processes. We use multiple client nodes to avoid client saturation, and vary the number of di-

rectory servers; each line labeled “Slice- N ” has N PCs acting as directory servers. For comparison, the N-MFS line measures an NFS server exporting a memory-based file system (FreeBSD MFS). MFS initially performs better due to Slice logging and update traffic, but the MFS server’s CPU quickly saturates with more clients. In contrast, the Slice request routing schemes spread the load among multiple directory servers, and both schemes show good scaling behavior with more servers. Each server saturates at 6000 ops/s generating about 0.5 MB/s of log traffic.

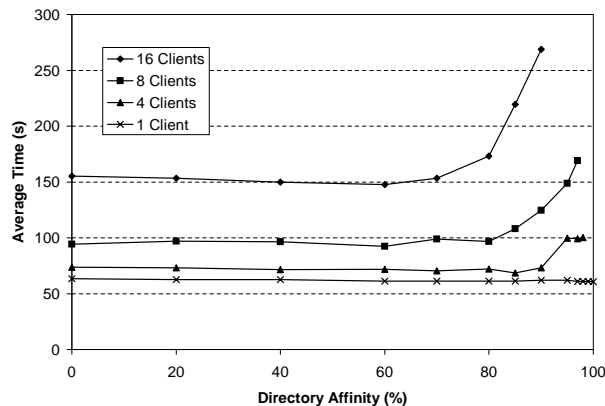


Figure 4: Impact of affinity for MKDIR SWITCHING.

Figure 4 shows the effect of varying directory affinity ($1 - p$) for MKDIR SWITCHING under the name-intensive *untar* workload. The X-axis gives the probability $1 - p$ that a new directory is placed on the same server as its parent; the Y-axis shows the average *untar* latency observed by the clients. This test uses four client nodes hosting one, four, eight, or sixteen client processes against four directory servers. For light workloads, latency is unaffected by affinity, since a single server can handle the load. For heavier workloads, increasing directory affinity along the X-axis initially yields a slight improvement as the number of cross-server operations declines. Increasing affinity toward 100% ultimately degrades performance due to load imbalances. This simple experiment indicates that MKDIR SWITCHING can produce even distributions while redirecting fewer than 20% of directory create requests. A more complete study is needed to determine the best parameters under a wider range of workloads.

Overall performance and scalability. We now report results from SPECsfs97, the industry-standard benchmark for network-attached storage. SPECsfs97 runs as a group of workload generator processes that produce a realistic mix of NFS V3

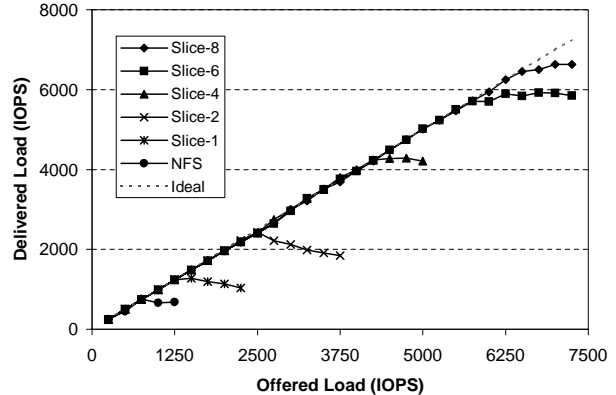


Figure 5: SPECsfs97 throughput at saturation.

requests, check the responses against the NFS standard, and measure latency and delivered throughput in I/O operations per second (IOPS). SPECsfs is designed to benchmark servers but not clients; it sends and receives NFS packets from user space without exercising the client kernel NFS stack. SPECsfs is a demanding, industrial-strength, self-scaling benchmark. We show results as evidence that the prototype is fully functional, complies with the NFS V3 standard, and is independent of any client NFS implementation, and to give a basis for judging prototype performance and scalability against commercial-grade servers.

The SPECsfs file set is skewed heavily toward small files: 94% of files are 64 KB or less. Although small files account for only 24% of the total bytes accessed, most SPECsfs I/O requests target small files; the large files serve to “pollute” the disks. Thus saturation throughput is determined largely by the number of disk arms. The Slice configurations for the SPECsfs experiments use a single directory server, two small-file servers, and a varying number of storage nodes. Figures 5 and 6 report results; lines labeled “Slice- N ” use N storage nodes.

Figure 5 gives delivered throughput for SPECsfs97 in IOPS as a function of offered load. As a baseline, the graph shows the 850 IOPS saturation point of a single FreeBSD 4.0 NFS server on a Dell 4400 exporting its disk array as a single volume (using the CCD disk concatenator). Slice-1 yields higher throughput than the NFS configuration due to faster directory operations, but throughput under load is constrained by the disk arms. The results show that Slice throughput scales with larger numbers of storage nodes, up to 6600 IOPS for eight storage nodes with a total of 64 disks.

Figure 6 gives average request latency as a function

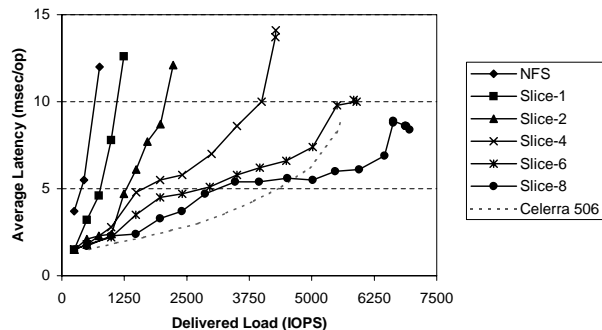


Figure 6: SPECSfs97 latency.

of delivered throughput. Latency jumps are evident in the Slice results as the ensemble overflows its 1 GB cache on the small-file servers, but the prototype delivers acceptable latency at all workload levels up to saturation. For comparison, we include vendor-reported results from *spec.org* for a recent (4Q99) commercial server, the EMC Celerra File Server Cluster Model 506. The Celerra 506 uses 32 Cheetah drives for data and has 4 GB of cache. EMC Celerra is an industry-leading product: it delivers better latency and better throughput than the Slice prototype in the nearest equivalent configuration (Slice-4 with 32 drives), as well as better reliability through its use of RAID with parity. What is important is that the interposed request routing technique allows Slice to scale to higher IOPS levels by adding storage nodes and/or file manager nodes to the LAN. Celerra and other commercial storage servers are also expandable, but the highest IOPS ratings are earned by systems using a VOLUME PARTITIONING strategy to distribute load within the server. For example, this Celerra 506 configuration exported eight separate file volumes. The techniques introduced in this paper allow high throughputs without imposing volume boundaries; all of the Slice configurations serve a single unified volume.

6 Conclusion

This paper explores interposed request routing in Slice, a new architecture for scalable network-attached storage. Slice interposes a simple redirecting μ proxy along the network path between the client and an ensemble of storage nodes and file managers. The μ proxy virtualizes a client/server file access protocol (e.g., NFS) by applying configurable request routing policies to distribute data and requests across the ensemble. The ensemble nodes cooperate to provide a unified, scalable file service.

The Slice μ proxy distributes requests by request type and by target object, combining functional decomposition and data decomposition of the request traffic. We describe two policies for distributing name space requests, MKDIR SWITCHING and NAME HASHING, and demonstrate their potential to automatically distribute name space load across servers. These techniques complement simple grouping and striping policies to distribute file access load.

The Slice prototype delivers high bandwidth and high request throughput on an industry-standard NFS benchmark, demonstrating scalability of the architecture and prototype. Experiments with a simple μ proxy packet filter show the feasibility of incorporating the request routing features into network elements. The prototype demonstrates that the interposed request routing architecture enables incremental construction of powerful distributed storage services while preserving compatibility with standard file system clients.

Availability. For more information please visit the Web site at <http://www.cs.duke.edu/ari/slice>.

Acknowledgements. This work benefited from discussions with many people, including Khalil Amiri, Mike Burrows, Carla Ellis, Garth Gibson, Dan Muntz, Tom Rodeheffer, Chandu Thekkath, John Wilkes, Ken Yocum, and Zheng Zhang. Andrew Gallatin assisted with hardware and software in numerous ways. We thank the anonymous reviewers and our shepherd, Timothy Roscoe, for useful critiques and suggestions.

References

- [1] Khalil Amiri, Garth Gibson, and Richard Golding. Highly concurrent shared storage. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, April 2000.
- [2] Darrell C. Anderson and Jeffrey S. Chase. Failure-atomic file access in an interposed network storage system. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2000.
- [3] David Anderson. Object Based Storage Devices: A command set proposal. Technical report, National Storage Industry Consortium, October 1999.
- [4] Tom Anderson, Michael Dahlin, J. Neefe, David Patterson, Drew Roselli, and Randy Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.
- [5] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein,

- David A. Patterson, and Katherine Yelick. Cluster I/O with River: Making the fast case common. In *I/O in Parallel and Distributed Systems (IOPADS)*, May 1999.
- [6] Andrew D. Birrell and Roger M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, SE-6(5):450–453, September 1980.
- [7] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.
- [8] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25,1 of *Performance Evaluation Review*, pages 272–284, New York, June 15–18 1997. ACM Press.
- [9] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [10] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 155–162, 1987.
- [11] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 69–79, December 1993.
- [12] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.
- [13] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [14] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 80–93, December 1993.
- [15] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth ACM Symposium on Theory of Computing*, pages 654–663, El Paso, May 1997.
- [16] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.
- [17] R. Macklem. Not quite NFS, soft cache consistency for NFS. In *USENIX Association Conference Proceedings*, pages 261–278, January 1994.
- [18] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Reducing the disk I/O of web proxy server caches. In *USENIX Annual Technical Conference*, June 1999.
- [19] Marshall K. McKusick, William Joy, Samuel Lefler, and Robert Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [20] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenopoele, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [21] Kenneth Preslan, Andrew Barry, Jonathan Brasow, Grant Erickson, Erling Nygaard, Christopher Sabol, Steven Soltis, David Teigland, and Matthew O’Keefe. A 64-bit, shared disk file system for Linux. In *Sixteenth IEEE Mass Storage Systems Symposium*, March 1999.
- [22] Ron L. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. <ftp://ftp.internic.net/rfc/rfc1321.txt>.
- [23] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.
- [24] Chandu Thekkath, Tim Mann, and Ed Lee. Frangipani: A scalable distributed file system. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–237, October 1997.
- [25] Robbert van Renesse, Andrew Tanenbaum, and Annita Wilschut. The design of a high-performance file server. In *The 9th International Conference on Distributed Computing Systems*, pages 22–27, Newport Beach, CA, June 1989. IEEE.
- [26] Geoff M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’98)*, June 1998.