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 Networks (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).

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

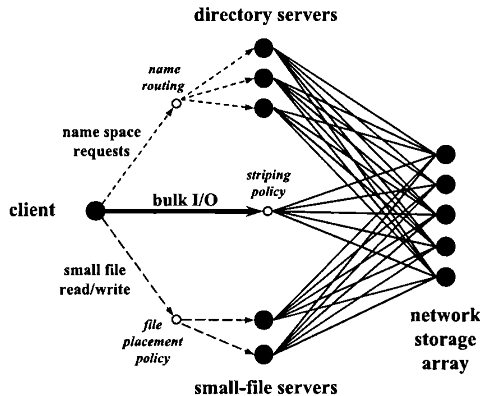

Figure 1: Combining functional decomposition and data decomposition in the Slice architecture.

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 μproxy, 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 μproxy 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 μproxy 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 μproxy 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 μproxy

An overarching goal is to keep the μproxy simple, small, and fast. The μproxy 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 μproxy 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 μproxy 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 μproxy.

The μproxy 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 μproxy. Although the μproxy resides "within the network", it acts as an extension of the service. For example, since the μproxy 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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

2.2 Network Storage Nodes

A shared array of network storage nodes provides all disk storage used in a Slice ensemble. The μproxy 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 μproxy to reside outside of the server ensemble's trust boundary. In this case, the damage from a compromised μproxy 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 μproxy 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 μproxy 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

DOCKET
ALARM Find authenticated court documents without watermarks at docketalarm.com.

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 μproxy 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 μproxy 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 system 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 μproxy 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 similiar 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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

3 Request Routing Policies

This section explains the structure of the μproxy 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 μproxy 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 μproxy 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 μproxy. The μproxy 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 μproxy never modifies the tables.

The μproxy 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 μproxy 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 μproxy, which acts as an extension of the service. Directory servers encode a *fileID* in each fhandle, which the μproxies extract as a routing key.

- *Read/write offset.* NFS I/O operations specify the range of offsets covered by each *read* and *write*. The μproxy 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 μproxy 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 μproxy 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 μproxy 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 μproxy 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 μproxies interact with the coordinators

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.