Failure-Atomic File Access in an Interposed Network Storage System

Darrell Anderson and Jeff Chase*
Department of Computer Science
Duke University
{anderson, chase}@cs.duke.edu

Abstract

This paper presents a recovery protocol for block I/O operations in Slice, a storage system architecture for high-speed LANs incorporating network-attached block storage. The goal of the Slice architecture is to provide a network file service with scalable bandwidth and capacity while preserving compatibility with off-the-shelf clients and file server appliances. The Slice prototype "virtualizes" the Network File System (NFS) protocol by interposing a request switching filter at the client's interface to the network storage system (e.g., in a network adapter or switch).

The distributed Slice architecture separates functions typically combined in central file servers, introducing new challenges for failure atomicity. This paper presents a protocol for atomic file operations and recovery in the Slice architecture, and related support for reliable file storage using mirrored striping. Experimental results from the Slice prototype show that the protocol has low cost in the common case, allowing the system to deliver client file access bandwidths approaching gigabit-per-second network speeds.

1 Introduction

Faster I/O interconnect standards and the arrival of Gigabit Ethernet greatly expand the capacity of inexpensive commodity computers to handle large amounts of data for scalable computing, network services, multimedia and visualization. These advances and the growing demand for storage increase the need for network storage systems that are incrementally scalable, reliable, and easy to administer, while serving the needs of diverse workloads running on a variety of client platforms.

Commercial systems increasingly provide scalable shared storage by interconnecting storage devices and servers with dedicated Storage Area Networks (SANs), e.g., FibreChannel. Yet recent order-of-magnitude improvements in LAN performance have narrowed the bandwidth gap between SANs and LANs. This creates an opportunity to deliver competitive storage solutions by aggregating low-cost storage nodes and servers, using a general-purpose LAN as the storage backplane. In such a system it is possible to incrementally scale either capacity or bandwidth of the shared storage resource by attaching additional storage to the network.

A variety of commercial products and research proposals pursue this vision by layering device protocols (e.g., SCSI) over IP networks, building cluster file systems that manage distributed block storage as a shared disk volume, or installing large server appliances to export SAN storage to a LAN using network file system protocols. Section 2.1 surveys some of these systems.

This paper deals with a network storage architecture — called Slice — that takes an alternative approach. Slice places a request switching filter at the client's interface to the network storage system; the role of the filter is to "wrap" a standard IP-based client/server file system protocol, extending it to incorporate an incrementally expandable array of network-attached block storage nodes. The Slice prototype implements the architecture by virtualizing the Network File System version 3 protocol (NFS V3). The request switching filter intercepts and rewrites a subset of the NFS V3 packet stream, directing I/O requests to the network storage array and associated servers that make up a Slice ensemble appearing to the client as a unified NFS volume. The system is compatible with off-the-shelf NFS clients and servers, in order to leverage the large installed base of NFS clients and the high-quality NFS server appliances now on the market.

The Slice architecture assumes a block storage model loosely based on a proposal in the National Storage Industry Consortium (NSIC) for object-based storage devices (OBSD) [2]. Key elements of the OBSD proposal were in turn inspired by research on Network Attached Secure Disks (NASD) [8, 9]. Storage nodes are "object-based" rather than sector-based, meaning that requesters address

*This work is supported by the National Science Foundation (CCR-96-24857, EIA-9870724, and EIA-9972879) and by equipment grants from Intel Corporation and Myricom.

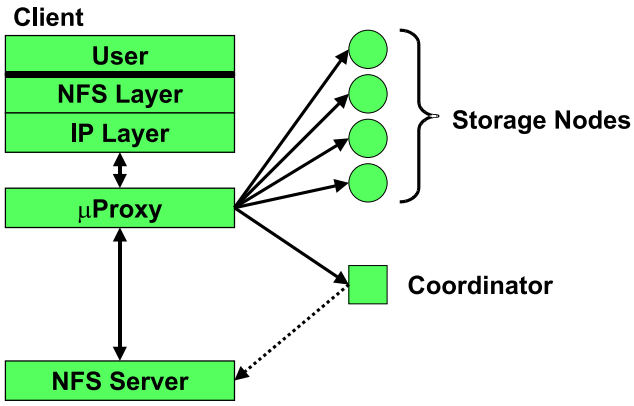

Figure 1. The Slice distributed storage architecture.

data on each storage node as logical offsets within *storage objects*. A storage object is an ordered sequence of bytes with a unique identifier. The NASD work and the OBSD proposal allow for cryptographic protection of object identifiers if the network is insecure [8].

The Slice architecture separates functions that are combined in central file servers. The contribution of this paper is to present a simple solution to the coordination and recovery issues raised by this structure. Our approach introduces a *coordinator* responsible for preserving atomicity of key NFS operations, including file truncate/remove, extending writes, and write commitment. The coordinators use a simple intention logging protocol, with variants for each operation type that minimize the common-case costs. We also show how the protocol supports failure-atomic write commitment for mirrored files in the Slice prototype. Mirroring consumes more storage and network bandwidth than striping with RAID redundancy, but it is simple and reliable, avoids the overhead of computing and updating parity, and allows load-balanced reads [4, 12].

This paper is structured as follows. Section 2 summarizes the Slice architecture. Section 3 describes mechanisms for operation atomicity and failure handling. Section 4 presents experimental results from the Slice prototype on a Myrinet network, showing that the Slice architecture and recovery protocols achieve file access performance approaching gigabit-per-second network speeds, limited primarily by the client NFS implementation. Section 5 concludes.

2 Overview

Figure 1 depicts the Slice architecture with NFS clients and servers. The architecture interposes a "microproxy" (*μproxy*) between the client IP stack and the Slice server ensemble. The μproxy examines NFS requests and responses, redirecting requests and transforming responses as necessary to represent the distributed storage service as a unified NFS service to its client. For some operations, the μproxy must generate new requests and pair responses with requests. The μproxy may reside within the client itself, or in a network element along the communication path between the client and the servers. In our current prototype the μproxy is implemented as a packet filter installed on the client below the NFS/UDP/IP stack.

The μproxy is a simple state machine with minimal buffering requirements. It uses only soft state; the μproxy may fail without compromising correctness. The μproxy may reside outside of the trust boundary, although it may damage the contents of specific files by misusing the authority of users whose requests are routed through it. In this paper we limit our focus to aspects of the μproxy internals and policies that are directly related to operation atomicity and the recovery protocol.

The *coordinator* plays an important role in managing global recovery of operations involving multiple sites. A Slice configuration may contain any number of coordinators, with each coordinator managing operations for some subset of files. The functions of the coordinator may be combined with the file server, but we consider them separately to emphasize that the architecture is compatible with standard file servers.

Our implementation combines the coordinator with a *map service* responsible for tracking file block location. The coordinator servers maintain a *global block map* for each file giving the storage site for each block. The μproxies read, cache, modify, and write back fragments of the global maps as they execute *read* and *write* operations on files. The global maps allow flexible per-file policies for block placement and striping in the network storage array; although the system may use deterministic block placement functions as an alternative to the global maps, this paper includes a discussion of the maps to show how the recovery protocol incorporates them.

The μproxy intercepts read and write operations targeted at file regions beyond a configurable *threshold offset*. Logical file offsets beyond the threshold are referred to as the *striping zone*; the μproxy redirects all reads and writes covering offsets in the striping zone to an array of *block storage nodes* according to system striping policies and the block maps maintained by the coordinators. The policies and protocols include support for *mirrored striping* ("RAID-10") for redundancy to protect against storage node failures, as described in Section 3.2. The Slice storage nodes export object-based block storage to the network; our prototype storage nodes accept NFS *read* and *write* operations on a flat space of storage objects uniquely identified by NFS file handles. Although NFS file handles provide only a weak

form of protection in our prototype, the architecture is compatible with proposals for cryptographic protection of storage object identifiers for insecure networks [8].

The μproxy identifies *read* and *write* operations in the striping zone by examining the request offset and length. Small files are not striped; these are files whose logical size is below the threshold offset, i.e., that have never received a write in the striping zone. Note that even large files are not striped in their entirety; data written below the threshold offset of a large file is stored along with the small files. File regions outside the striping zone do not benefit from striping, but the performance cost becomes progressively less significant as file sizes grow.

In addition to the interactions required for I/O requests, the μproxies cooperate with the network storage nodes and the file's coordinator to allocate global maps for extending *write* operations, and to release storage on *remove* and *truncate* operations. These multisite operations introduce recovery issues described in the next section. All other file operations pass through the μproxy to the NFS server as they did before, and incur no additional overhead for managing distributed storage.

This architecture scales to higher bandwidth and capacity by adding storage nodes, since the NFS server is outside the critical path of reads and writes handled by the block storage nodes. It is also possible to scale or replicate other file service functions within the context of the Slice request switching architecture. For simplicity this paper assumes that a single standard NFS file server manages the entire volume name space.

The goal of the mechanisms described in this paper is to deliver consistency and failure properties that are no weaker than commercial NFS implementations. While the basic approach is quite similar to write-ahead logging that might be taken on a journaling central file server with distributed disks, we extend it to support multisite operations without the awareness of the client, NFS file server, or the storage nodes. Our approach to committing writes assumes use of the NFS V3 asynchronous writes and write commitment protocol, as described below. This paper does not address the issue of concurrent write sharing of files, and Slice as defined may provide weaker concurrent write sharing guarantees than some NFS implementations. However, the architecture is compatible with NFS file leasing extensions for consistent concurrent write sharing, as defined in NQ-NFS [13] and early IETF draft proposals for the NFS V4 protocol.

2.1 Related Work

The Cambridge Universal File Server [5] proposed structuring a distributed file system as a separate name service and file block storage service. One system to take this approach was Swift [6]. Slice is similar to Swift in that each client reads or writes data directly to block storage sites on the network, choreographed by a client distribution agent using maps provided by a third-party storage mediator. Another system derived from the Swift architecture is Cheops, a striping file system for CMU NASD storage systems [9, 8]. The Swift and Cheops work did not directly address atomicity or recovery issues.

Amiri et. al. [1] show how to preserve read and write atomicity in a shared storage array using RAID striping with parity. This work focuses primarily on safe concurrent accesses to a fixed space of blocks. It does not address file system consistency in the presence of host failures.

A number of scalable file systems separate some striping functions from other file system code by building the file system above a striped network storage volume using a shared disk model. This approach has been used with both log-structured [11, 3] and conventional [15, 14] file system structures. In these systems, multisite operations including *truncate* and *remove* are made failure-atomic using write-ahead metadata logging on the file server. The log-structured approach also relies in part on a separate *cleaner* process to reclaim space.

Relative to these systems, this paper shows how to factor out recovery functions so that multisite recovery may be interposed in the context of a standard client/server file system protocol, without modifying the client or server.

3 Atomic Operations on Network Storage

A multisite operation begins when the μproxy intercepts an NFS V3 *write*, *remove*, *truncate* (*setattr*) or *commit* request from a client. To handle the request, the μproxy may redirect the request or generate additional request messages to nodes in the Slice ensemble, including storage nodes, the coordinator for the target file, and the NFS server. Figure 2 illustrates the message exchanges for the multisite operations discussed in this section.

When the operation is complete at all sites, the μproxy passes through an NFS V3 response to the client. If any participant fails during this sequence — the μproxy, a storage node, the coordinator, or the file server — a recovery protocol is initiated. The recovery protocol is specific to the particular operation in progress, and it may either complete the operation (roll forward) or abort it (roll back). If the system aborts the operation or delays the response, a standard NFS client may reinitiate the operation by retransmitting the request after a timeout, unless the client itself has failed.

The basic protocol is as follows. At the start of the operation, the μproxy sends to the coordinator an *intention* to perform the operation (e.g., Figure 2, messages e and l). The coordinator logs the intention to stable disk storage and responds, authorizing the μproxy to carry out the operation.

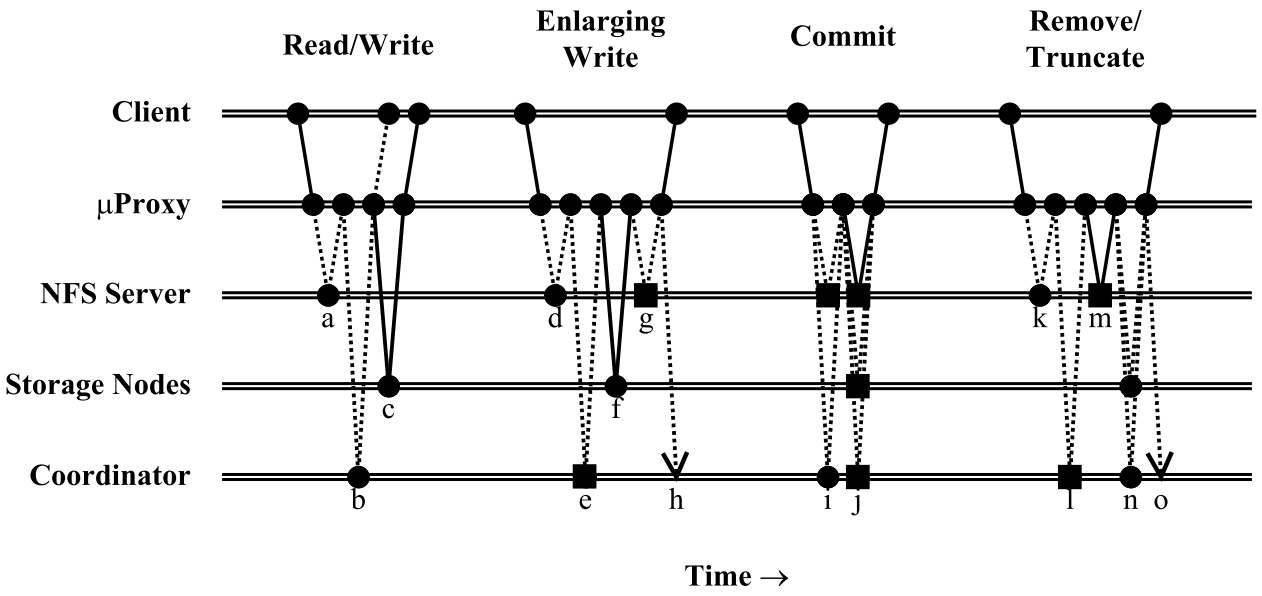

Figure 2. Message exchanges for multisite Slice/NFS operations. Dotted line message exchanges are avoided in common cases. Square endpoints represent synchronous storage writes.

When the operation is complete, the μproxy notifies the coordinator with a *completion* message, asynchronously clearing the intention (e.g., messages h and o). If the coordinator does not receive the completion within a specified period, it probes one or more 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.

This is a variant of the standard two-phase commit protocol [10] adapted to a file system context with idempotent operations. The details for each operation vary significantly. In particular, each operation allows optimizations to avoid most messaging and logging delays in common cases, as described below. Slice further improves performance by avoiding multisite operations for small files stored entirely on the file server, i.e., files that have never received writes beyond the configurable threshold offset. In this way, the system amortizes the costs of the protocol across a larger number of bytes and operations, since it incurs these costs only to create and truncate/remove large files, and to commit groups of writes to large files.

The following subsections describe the protocol as it applies to each type of multisite operation. We then set the protocol in context with conventional two-phase commit.

3.1 Write Commitment

An NFS V3 *commit* operation stabilizes pending or unstable writes on a given file. The NFS V3 protocol allows a server failure to legally discard any subset of the uncommitted writes and associated metadata, provided that the client can detect any loss by comparing verifier values returned by the file service in its responses to *write* and *commit* operations. NFS V3 clients buffer uncommitted writes locally so that they may re-execute these writes after a server failure. Clients may safely discard their buffered writes after a successful *commit*. Note that the verifier value returned by *write* and *commit* is not itself significant; the service guarantees only that the verifier changes after a failure.

To handle a *commit* on a file that has unstable writes in the striping zone, the μproxy executes a message exchange with each storage node that owns uncommitted writes on the file (Figure 2, message j). The μproxy also completes the writes, which may involve an exchange with the coordinator (map service) and/or the NFS server. The μproxy pushes any updates to the file's map back to the coordinator (message i). If the write enlarged the file, it pushes the new file size to the NFS server via a *setattr* (message i). When all operations have completed successfully, the μproxy responds to the client with a valid verifier.

The μproxy detects any failures by comparing response verifiers against a stored copy of the previous verifier returned by each participant. If any participant fails, the μproxy reports the failure by changing the response veri-

fier to the client. If the μproxy itself loses its state, it may report failure for a *commit* that has successfully completed at all sites. This forces the client to reinitiate writes unnecessarily, but is otherwise harmless.

Intention logging is unnecessary for *write* and *commit* on unmirrored files. This is because the file service remains in a legal state throughout the write sequence and *commit*. The exact ordering of operations is not strictly important; the *commit* is complete only when the client discarded its buffered writes after receiving a valid response. If a failure occurs, the client itself is responsible for restarting the write sequence after receiving a negative response or no response to its *commit* request.

3.2 Mirrored Writes

Writes to a mirrored file are replicated using a read-any-write-all model. Without loss of generality we assume that the replication degree is two. A replication degree of two guarantees that a file is available unless two or more storage nodes fail concurrently, or the file's coordinator fails together with one storage node and a client who was actively writing the file.

Block maps for a mirrored file have dual entries for each logical block, with one entry for each block replica. The μproxy writes each block to a pair of storage nodes selected according to some placement policy, which is not important for the purposes of this paper. A mirrored write is considered complete only after it has committed; i.e., both storage nodes confirm that the block is stable, and (if applicable) the file's coordinator (map service) confirms that the covering map fragment is stable.

Mirrored writes use the intention protocol to reconcile replicas in the event of a failure. If a participant fails while there are incomplete mirrored writes, then it is possible that the write executed at one replica but not the other. In practice, this does not occur unless a client fails concurrently with one or more server failures, since an NFS V3 client retransmits all uncommitted writes after a server failure, as described in Section 3.1.

The mirrored write protocol piggybacks intention messages for mirrored writes on the μproxy's request for the map fragment covering the write. Before returning the requested map fragment, the coordinator logs the intention record and updates a conservative in-memory *active region list* of offset ranges or map fragments that might be held by each μproxy, and that may have incomplete writes. These intentions are cleared implicitly by a *commit* request covering the region; *commit* causes the μproxy to discard all covered map fragments for a mirrored file.

If a client (or its μproxy) fails, any uncommitted mirrored writes are guaranteed to be covered by the coordinator's active region list. The coordinator can reconcile the replicas for these regions by traversing the region list; any conflict within the active regions may be resolved by selecting one replica to dominate. In principle, the system can serve one copy of the file concurrently with reconciliation, even if a storage node fails. If the coordinator fails, it recovers a conservative approximation of its active region list from its intentions log.

In practice, most intention logging activity for mirrored writes may be optimized away. Slice logs these intentions only when a mirrored file first comes into active write use, e.g., when a μproxy first requests map fragments with intent to write. If a file falls out of write use (no map fragment requests received since the last *commit* completion), the coordinator marks the file as inactive by logging a *write-complete* entry. This protocol adds a synchronous log write to the write-open path for mirrored files, but this cost is amortized over all writes on the file. It allows a recovering coordinator to identify a superset of the mirrored files that may need reconciliation after a multiple failure.

One drawback of the protocol is that a buggy or malicious client might cause the active region list to grow without bound by issuing large numbers of writes and never committing them. This is not a problem with clients that correctly buffer their uncommitted writes, since the number of writes is limited by available memory; in any case, standard clients commit writes at regular intervals under the control of a system update daemon. For malicious clients, the system may avoid this problem by weakening replica consistency guarantees for mirrored files with writes left uncommitted for unreasonably long periods.

3.3 Truncate and Remove

The protocol for *truncate* and *remove* relies on the NFS server to maintain an authoritative record of the file length and link count. The μproxy first consults a set of attributes for the target file (Figure 2, message k); the attributes must be current up to the "three second window" defined by NFS implementations (see Section 3.4. If the target file's logical size shows that it has data in the striping zone, the μproxy issues an intention to the coordinator (message l) before issuing the NFS operation to the file server (message m). Once the operation has committed at the NFS server, the protocol contacts the storage nodes and coordinator (map service) to release storage (message n), then registers a completion with the coordinator (message o). In our current prototype the μproxy executes the entire protocol, but it could be done directly by the coordinator, simplifying the μproxy and saving one message exchange (the intention response and the completion).

If the intention expires, the coordinator probes the NFS server (using a *getattr*) to determine the status of the operation. If the operation completed on the NFS server, the

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.