Failure-Atomic File Access in the Slice Interposed Network Storage System

Darrell Anderson (anderson@cs.duke.edu) and Jeff Chase (chase@cs.duke.edu)*
Department of Computer Science
Duke University

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.

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.

Keywords: failure atomic, virtualized, network storage

1. Introduction

Faster I/O interconnect standards and the arrival of 10 Gigabit Ethernet greatly expand the capacity of inexpensive commodity computers to handle large amounts of data for scalable computing, cluster-based Internet services, digital audio and video, 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 band-

* 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.

width 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 redirector at the client's interface to the network storage system; the role of the redirector is to "wrap" a standard IP-based client/server file system protocol, extending it to incorporate an incrementally expandable array of servers and network-attached block storage nodes (Anderson et al., 2000). The Slice prototype implements the architecture by virtualizing the Network File System version 3 protocol (NFS V3). The request redirector 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 service. The system is compatible with off-the-shelf NFS clients and servers, enabling it to leverage the large installed base of NFS clients and the high-quality NFS server appliances now on the market. A related paper (Anderson et al., 2000) presents the Slice interposed request routing architecture in detail, and gives experimental evidence of scalability for industry-standard NFS file service workloads.

The contribution of this paper is to present a simple solution to the coordination and recovery issues raised by this structure. Our approach to scalability complicates recovery because the Slice architecture separates functions that are combined in central file servers. Our approach introduces a *coordinator* responsible for groups of files, preserving atomicity of key NFS operations including file truncate/remove, extending writes, and write commitment. The coordinators use a simple intention logging protocol similar to two-phase commit (Gray, 1978), with specialized variants for each operation type to minimize the common-case costs. We also show how to extend this approach to support failure-atomic write commitment for mirrored files, as a basis for high availability and reliability. 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,

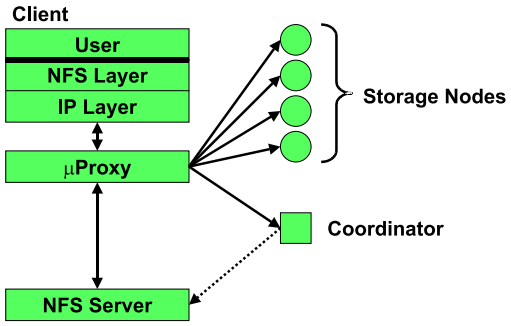

Figure 1. The Slice distributed storage architecture with a single NFS file manager.

and allows load-balanced reads (Arpaci-Dusseau et al., 1999; Lee and Thekkath, 1996).

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 configuration of the Slice architecture discussed in this paper, with NFS clients and servers. The architecture interposes a redirecting "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 (Anderson et al., 2000). 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 (e.g., a storage switch). 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, and their effects on performance and scalability of I/O operations. Note also that in the configuration depicted in Figure 1, operations on the volume name space and file attributes are handled by a single unmodified NFS server. This allows us to ignore complex issues related to distributing the name space, which is important for workloads involving large numbers of small files. Section 4 shows that this configuration 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 storage nodes. The companion paper (Anderson et al., 2000) shows how to extend the architecture to scale or replicate other file service functions including directory management, and it explores the complete μproxy architecture, implementation, and performance in more detail.

The Slice storage node architecture assumes a block storage model loosely based on a proposal in the National Storage Industry Consortium (NSIC) for object-based storage devices (OBSD) (Anderson, 1999). Key elements of the OBSD proposal were in turn inspired by research on Network Attached Secure Disks (NASD) (Gibson et al., 1997; Gibson et al., 1998). Storage nodes are "object-based" rather than sector-based, meaning that requesters address 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, and show good performance and scalability for large files (Gibson et al., 1997).

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.

The prototype Slice configuration discussed in this paper 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 (Gibson et al., 1997).

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.

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

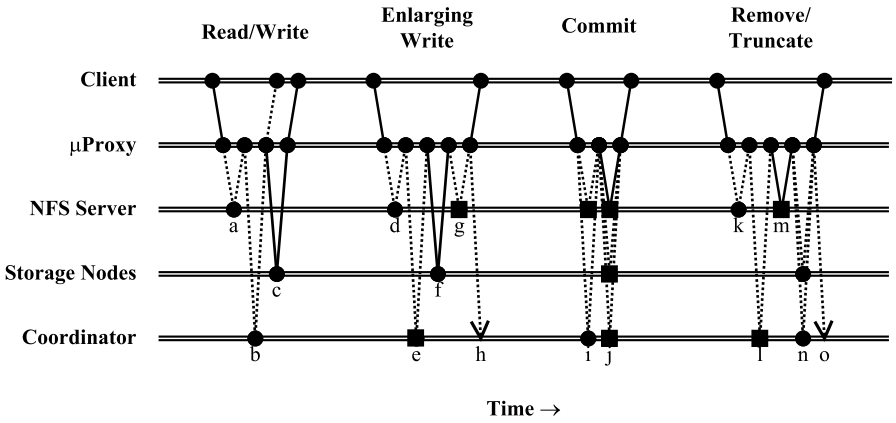

Figure 2. Message exchanges for multisite Slice/NFS operations. Dotted line message exchanges are avoided in common cases. Square endpoints represent synchronous storage writes.

is compatible with NFS file leasing extensions for consistent concurrent write sharing, as defined in NQ-NFS (Macklem, 1994) and the NFS V4 protocol (Pawlowski et al., 2000).

2.1. Related Work

The Cambridge Universal File Server (Birrell and Needham, 1980) proposed structuring a distributed file system as a separate name service and file block storage service. One system to take this approach was Swift (Cabrera and Long, 1991). 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 (Gibson et al., 1998; Gibson et al., 1997). The Swift and Cheops work did not directly address atomicity or recovery issues.

Amiri et. al. (Amiri et al., 2000) 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 (Hartman and Ousterhout, 1993; Anderson et al., 1995) and conventional (Thekkath et al., 1997; Preslan et al., 1999) file system structures. In these systems, multisite oper-

ations 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 File 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. 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 (Gray, 1978) 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 verifier 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 coordinator rolls the operation forward by contacting the storage nodes to release any orphaned storage.

3.4. Enlarging Writes

The truncate/remove protocol in Section 3.3 must avoid a race with an *enlarging write*, a special case of extending write that extends a "small" file beyond the threshold offset and into the striping zone, making it a "large" file. The danger is that another client will complete an enlarging write after the μproxy consults the file's logical size, recognizing it as a small file, and before the μproxy issues the *truncate/remove* operation to the NFS server. If this occurs, the μproxy could fail to notify the coordinator of the need to release network storage allocated to the newly enlarged file, leaving it orphaned by the *truncate/remove*.

One way to prevent the race is to conservatively notify the coordinator of all *truncate/remove* operations, even for small files. However, this imposes an extra message latency and perhaps a disk fault on truncates and removes of small files. We prefer instead to shift the costs to the enlarging write operation, increasing the creation cost of large files. The enlarging write cost is incurred once for each large file, and is amortized over all I/O operations on the file.

Our approach uses a variant of the basic intention protocol to detect the race when it occurs, and to release any orphaned storage. The trick is for the coordinator to detect that a μproxy has executed a *truncate/remove* operation based on attributes that were fetched before the completion of an enlarging write. After an enlarging write has completed, the file's coordinator contacts the NFS server to validate the file's existence and logical size. The coordinator delays this validation until a fixed *waiting period* has elapsed. The waiting period is chosen to exceed the time bound on the staleness of cached attributes in NFS (the three second rule) with ample slack time to account for clock skew and operation latencies. In the event of a race between the first enlarging write and a remove, this check will discover the file absent, and release the orphaned storage.

3.5. File Service Decomposition

For simplicity this paper assumes that a single standard NFS file server manages the entire volume name space, as well as the "small" file data for all files. The Slice architecture allows separation and independent scaling of each of these functions (Anderson et al., 2000). This section discusses failure atomicity in this larger, decomposed file service scope.

There are two important decompositions to address with respect to atomic operations. First, the name space may be distributed across multiple *directory* servers. Individual name entries are managed by a single directory server, and together the name entries held by all directory servers comprise the unified name space. Update operations

for existing files follow the the single namespace server model. *Create* and *remove* operations require some discussion.

A *create* operation must create the file entry (name and attributes) as well as update the contents and last-modify-time of its parent directory, which may exist on a different directory server from the entry. Attempts to access an entry while its *create* is in progress block until the *create* is resolved, and a *create* does not complete until both the name entry and parent directory contents have been created/updated, in that order. If a competing *create* races, the first to instantiate itself in the parent directory wins. The second backs off, returning a name-conflict error. If the system fails while a *create* is in progress, the operation is lost in its entirety. If the client has not failed, it retries its request and the retry proceeds normally. If the client fails mid-*create* but the Slice service does not, the service carries out the operation and sends its response (either success or failure) to the dead or restarted client, just as would a normal NFS server. *Remove* operations follow the same model, however also must deal with small-file data, addressed below.

By introducing specialized small-file servers, file names and attributes are separated from the file's initial data. Small-file *writes* must route to different servers, and *removes* have additional state to release. File attributes (e.g., size and last modify time) are maintained in the same way as for large files.

Remove with small-file data separation follows the policy illustrated in figure 2, removing small-file data in the same phase as any large file data. Small-file removals may proceed without intention logging, potentially leaking storage after an ill-timed failure. Unlike large files, the amount of storage consumed by orphaned small-files is bounded by the striping zone threshold, and could be lazily reclaimed by an online *fsck*-like scan. Alternatively, any removal (small or large) could invoke intention-logging protocol to guarantee safe removal, imposing significant "common-case" overheads. The prototype does not use intention logging for small-file removals.

3.6. Comparison to Two-Phase Commit

The basic intention logging protocol used in Slice is similar to conventional two-phase commit (Gray, 1978), but there are several key differences. These are brought about by the simple nature of the file system operations, which tends to make the protocol more efficient than a general two-phase commit in the common cases.

- For simplicity, the μproxy assumes most of the functions of the traditional commit coordinator: it transmits requests to participants

and gathers commit votes. However, it never actually performs a commit since it has no stable storage.

- Participants execute their portion of the operation in a fixed partial order, with one participant acting as the primary commit site. The purpose of the intention protocol is to detect and recover from failures that interrupt the sequence before the primary commit site executes its part of the operation. For example, the NFS server itself unwittingly acts as the primary commit site for removes, truncates, and extending writes (or extending write commits). For *truncate* and *remove*, a failure after the NFS server commits causes the recovery protocol to roll forward by releasing orphaned storage, similar to a conventional journaling file system or a file system scavenger (*fsck*).
- There is no need to notify participants other than the coordinator that the operation committed. The precommit is sufficient to stabilize the data, and the participants do not hold locks on the committed state. File operations are serialized (when necessary) at the NFS server (for name space operations) or at the coordinator (for reads and writes of shared files).

4. Prototype and Experimental Results

We have implemented the Slice prototype as a set of loadable kernel modules for the FreeBSD 4.0 operating system. The network storage nodes in our prototype are FreeBSD PCs serving blocks from local disks using UFS/FFS as a storage manager, with an external hash to map opaque NFS file handles to local files. The coordinator is implemented as an extension to the storage node module, consisting of a total of about 1400 lines of code. In our prototype, the μproxy is an IP filter between the IP stack and the network driver. The μproxy may rewrite or consume packets, and it may also generate new IP packets. The μproxy is a non-blocking state machine consisting of about 2500 lines of code. An overarching goal is to keep the μproxy simple, small, and fast.

This section presents experimental results from the Slice prototype. The intent is to show the costs of the interposed μproxy architecture, and the effect of these costs on delivered file access bandwidths. The prototype μproxy, coordinator, and storage service implement mechanisms needed for recovery during normal operation, including the coordinator intentions log. Thus they reflect the costs of recovery as

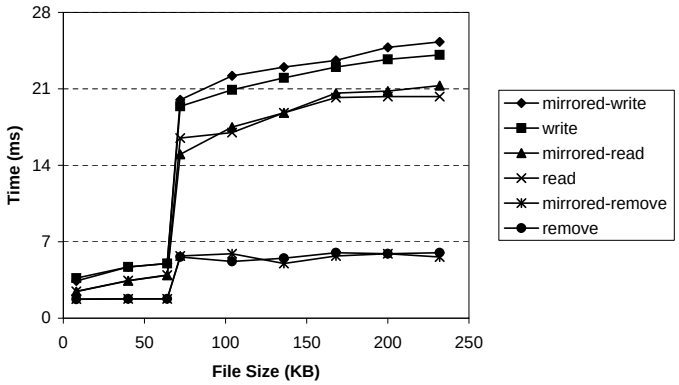

Figure 3. File read, write, and remove timings using a 64 KB threshold offset.

described in Section 3. However, reconciliation of active regions for mirrored replicas is not implemented.

In these experiments, clients are 450 MHz Pentium-III PCs using the Asus P2B motherboard with a 32-bit, 33 MHz PCI bus and Intel 440BX chipset. The NFS server and Slice storage nodes are Dell 4400 systems each with one 733 MHz Pentium-III Xeon using a ServerWorks chipset. The server network adapter and disk controllers are on independent peer 64-bit, 66 MHz PCI buses. Each has eight 18 GB Seagate Ultra-2 Cheetah disks. All machines are equipped with Myricom adapters based on the LANai-4 or and LANai-7 chipsets, with kernels built from the same FreeBSD 4.0 source pool. The LANai 7 adapter supports the same link speed as the LANai 4 (1.28 Gb/s), but it has higher internal speeds. We use the LANai-7 to reduce network bottlenecks in saturation experiments.

All network communication in these experiments uses Trapeze, a locally developed Myrinet messaging system optimized for network I/O traffic (Anderson et al., 1998; Chase et al., 1999). In this configuration, Trapeze/Myrinet provides 130 MB/s of point-to-point bandwidth with a 32 KB transfer size. In these experiments, the file system clients and servers access the Trapeze device through the standard UDP/IP network stack and Trapeze device driver. In this context, Trapeze/Myrinet yields better network performance than our Gigabit Ethernet network and support for large packets (32KB MTU), but it does not otherwise limit the generality of the system or results in any way.

Figure 3 shows the total time to *read*, *write*, and *remove* a file, varying file size from 8 KB to 232 KB, with the striping zone threshold set to 64 KB. All tests start with cold client and storage node caches. The *write* timings include the cost of *commit*. Each data point is the average of 50 trials. Slice uses four storage nodes for this experiment.

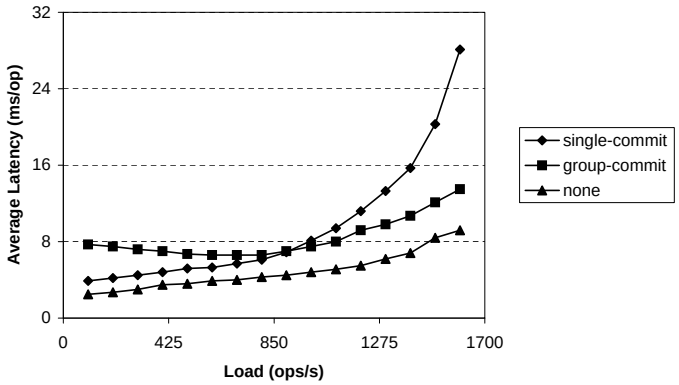

Figure 4. Coordinator logging overhead.

Latencies remain almost constant below the striping zone threshold; all interactions are exclusively with the NFS server. The files are small enough to write in their entirety before issuing the *commit*, and server block clustering loads the whole file with the first read. These files use no indirect blocks on the NFS server, bounding deletion time.

When file size exceeds the striping zone threshold, latencies jump as operations begin to involve multiple sites and incur costs of the intention logging protocol. For example, read and write costs increase as the μproxy faults block maps from the coordinator before issuing I/O beyond the threshold. Writes and *removes* register an intent with the coordinator before performing the first extending write into the striping zone or before issuing the remove to the NFS server, respectively. The resulting discontinuities are clearly shown in the graph; however, the cost becomes progressively less significant as file sizes grow.

Both read and write times increase linearly with file size, and *remove* time remains constant. The prototype serializes some sub-operations of *commit* for simplicity, compromising *write* latency slightly. At these sizes, mirroring has a negligible effect on both read and write times.

For this microbenchmark at most one intention logging operation is in flight at any given time. As a result, the coordinator is nearly idle, writing intentions sequentially to its log. Under these circumstances the intention logging protocols incurs a near-constant 4 ms latency, affecting the *write* and *remove* tests. Intention logging costs are studied in more detail in the following experiment.

Figure 4 compares the costs of three coordinator logging policies. This test uses a Slice ensemble with eight storage nodes, with the striping zone threshold set to 0 KB. With this null threshold, all nonempty files are "large" files, emphasizing the costs of our intention logging protocol.

For each data point, we first clean and seed the Slice ensemble with 4000 1KB files in 20 directories, and then generate a mixed stream of 20% *create*, 60% *write*, and 20% *remove* operations. We measure only *remove* latency because nearly all *remove* operations involve "large" file targets, triggering the intention logging protocol. Our load generator is a user-level program using raw sockets to send and receive NFS/UDP datagrams, similar to the SPECsfs97 benchmark (Corporation, 1997). Using raw sockets avoids the limiting overhead of the client NFS implementation, but still communicates through the Slice μproxy. Our generator selects random targets for each operation.

Figure 4 shows the effect of load (in operations per second on the x-axis) on *remove* latency for three coordinator logging policies. The first policy, NONE, omits intention logging altogether, and may suffer inconsistencies after failure(s). The SINGLE-COMMIT policy uses intention logging with a coordinator node, immediately writing log entries synchronously to stable storage. Finally, GROUP-COMMIT (Hagmann, 1987) uses intention logging where the coordinator amortizes synchronous I/O costs, waiting up to ten milliseconds or until at least five log entries arrive before initiating the synchronous log write, delaying its response until after the write completes. Both SINGLE-COMMIT and GROUP-COMMIT policies are failure-atomic, guaranteeing correctness under failures and consistent state after recovery. All other experiments in this paper use the SINGLE-COMMIT policy.

Naturally NONE is the cheapest of these policies, demonstrating the base cost of a *remove* operation. For this configuration, base *remove* cost scales linearly with request rate until the load reaches about 1300 operations per second. The SINGLE-COMMIT policy performs well at low load, saturating at high loads when intention log entries queue waiting for earlier synchronous writes to complete. Finally, the GROUP-COMMIT policy performs poorly at low loads when intention log requests wait for the timeout, but benefits from amortized log writes at higher loads, servicing more log requests with each disk I/O.

The architecture allows very high bandwidth for large files. Figure 5 shows I/O bandwidth delivered to a single client and a group of clients, varying the number of storage nodes. Bandwidths are measured using *dd* to read or write a 1.25 GB file in 32 KB chunks, with a Slice striping grain of 32 KB. Each graph gives both non-redundant and mirrored storage results.

The left-hand graph shows the measured I/O bandwidth delivered to a single client with a Lanai-7 adapter. We modified the FreeBSD NFS client for zero-copy I/O. Single client bandwidth scales with the number of storage nodes until the client CPU saturates reading at 105 MB/s or writing at 80 MB/s.

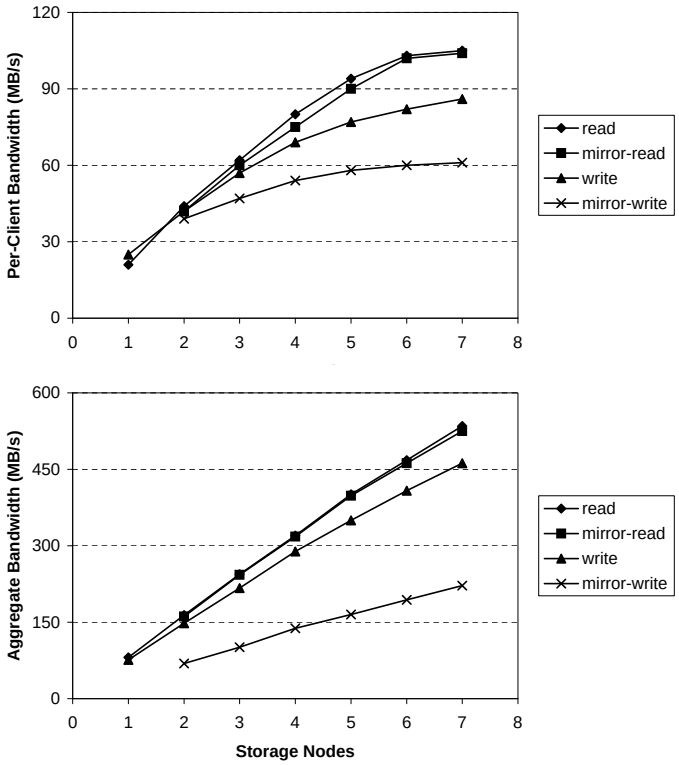

Figure 5. Single-client and saturation bandwidth for sequential read and write.

Mirrored read bandwidth to a single client is nearly identical to non-mirrored. In the absence of contention, the client may read one mirror replica in the same way it would read the only copy in the non-mirrored case. Mirroring writes pushes twice as many bytes out a client's network interface and consumes additional space overheads in the storage nodes, yielding slightly better than half non-mirrored write bandwidth.

Mirrored writes consume twice the main memory bus, I/O bus, and network link bandwidth as non-mirrored writes. Payload caching (Yocum and Chase, 2001), a technique which uses network interface memory to cache data payloads, can reduce the load on the main memory and I/O buses. The second (mirror) write does not require an I/O bus transfer; it is sent from the cached copy on the network interface. In these experiments the main memory and I/O buses are not bottlenecks, and payload caching delivers little improvement in observed bandwidth.

The right-hand graph in figure 5 shows aggregate saturation I/O bandwidth. Eight clients are used to saturate the storage nodes. Seven storage nodes deliver a sum total of about 550 MB/s to readers, and

450 MB/s to writers. Mirrored reads favor one replica, almost matching non-mirrored read performance. Mirrored writes consume additional transfer time, disk arm utilization (unlike single client with no contention), and space overheads within the storage nodes, delivering slightly less than half of non-mirrored write bandwidth.

5. Conclusion

This paper presents protocols for reliable mirrored files and failure-atomic file operations in Slice, a scalable network storage system for high-speed LANs with network-attached block storage. Slice *interposes* a request switching filter at the client's interface to the network storage system, to provide scalable bandwidth and capacity by distributing file data across the network storage nodes.

The Slice architecture is designed to leverage emerging models of object-based network storage, while allowing incremental deployment of scalable network I/O as an add-on to existing network file system installations. The request switching architecture presents challenges for failure-atomicity because it separates functions typically combined in central file servers.

This paper shows how to address the reliability and failure-atomicity for file system operations in the Slice architecture and related distributed network storage systems. We show how to provide reliable file storage with support for mirrored striping. Experimental results from the Slice prototype quantify the costs of the recovery protocol and show that they are acceptable in the common case. Our results show that the Slice approach has low cost and delivers client file access bandwidths approaching gigabit-per-second Myrinet network speeds. We also show the need for streamlining NFS client stacks to respond to faster networks and high-performance I/O services for I/O-intensive workloads including scalable computing, data-intensive network services, multimedia, and visualization.

References

Amiri, K., G. Gibson, and R. Golding: 2000, 'Highly concurrent shared storage'. In: *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*. pp. 298–307.

Anderson, D.: 1999, 'Object Based Storage Devices: A Command Set Proposal'. Technical report, National Storage Industry Consortium.

Anderson, D., J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley: 1998, 'Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet'. In: *1998 Usenix Technical Conference*.

Anderson, D. C., J. S. Chase, and A. M. Vahdat: 2000, 'Interposed Request Routing for Scalable Network Storage'. In: *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*. Award paper selected for a special issue of ACM Transactions on Computer Systems (TOCS).

Anderson, T., M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang: 1995, 'Serverless Network File Systems'. In: *Proceedings of the ACM Symposium on Operating Systems Principles*. pp. 109–126.

Arpaci-Dusseau, R. H., E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick: 1999, 'Cluster I/O with River: Making the Fast Case Common'. In: *I/O in Parallel and Distributed Systems (IOPADS)*.

Birrell, A. D. and R. M. Needham: 1980, 'A Universal File Server'. *IEEE Transactions on Software Engineering* **SE-6**(5), 450–453.

Cabrera, L.-F. and D. D. E. Long: 1991, 'Swift: Using Distributed Disk Striping to Provide High I/O Data Rates'. *Computing Systems* **4**(4), 405–436.

Chase, J. S., D. C. Anderson, A. J. Gallatin, A. R. Lebeck, and K. G. Yocum: 1999, 'Network I/O with Trapeze'. In: *1999 Hot Interconnects Symposium*. Invited paper.

Corporation, S. P. E.: 1997, 'SPEC SFS Release 2.0 Run and Report Rules'.

Gibson, G. A., D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka: 1997, 'File Server Scaling with Network-Attached Secure Disks'. In: *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Vol. 25,1 of *Performance Evaluation Review*. New York, pp. 272–284, ACM Press.

Gibson, G. A., D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka: 1998, 'A Cost-Effective, High-Bandwidth Storage Architecture'. In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*.

Gray, J. N.: 1978, 'Notes on Database Operating Systems'. In: *Operating Sys.: An Advanced Course*, Vol. 60 of *Lecture Notes in CS*. New York: Springer, p. 393.

Hagmann, R.: 1987, 'Reimplementing the Cedar File System Using Logging and Group Commit'. In: *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*. pp. 155–162.

Hartman, J. H. and J. K. Ousterhout: 1993, 'The Zebra Striped Network File System'. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. pp. 29–43.

Lee, E. K. and C. A. Thekkath: 1996, 'Petal: Distributed Virtual Disks'. In: *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, pp. 84–92.

Macklem, R.: 1994, 'Not Quite NFS, Soft Cache Consistency for NFS'. In: *USENIX Association Conference Proceedings*. pp. 261–278.

Pawlowski, B., S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow: 2000, 'The NFS Version 4 Protocol'. In: *Second International Systems and Networking (SANE) Conference*.

Preslan, K., A. Barry, J. Brassow, G. Erickson, E. Nygaard, C. Sabol, S. Soltis, D. Teigland, and M. O'Keefe: 1999, 'A 64-bit, Shared Disk File System for Linux'. In: *Sixteenth IEEE Mass Storage Systems Symposium*.

Thekkath, C. A., T. Mann, and E. K. Lee: 1997, 'Frangipani: A Scalable Distributed File System'. In: *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*.

Yocum, K. G. and J. S. Chase: 2001, 'Payload Caching: High-Speed Data Forwarding for Network Intermediaries'. In: *2001 USENIX Technical Conference*.