Afterlife: A Distributed and Recoverable File System Based on a Unified Approach to Logging

Edmond Lau, May Zhou, Emily Yan, and Xiao Yu

6.824: Distributed Computer Systems
Final Project Report

{edmond, amayz, kirason, xyu}@mit.edu
Massachusetts Institute of Technology

May 12, 2005

Abstract

We present a distributed and recoverable file system called Afterlife that only requires a single replicated log to recover from failures in both the file system layer and the storage layer. Based on this unified log, we devise recovery procedures for both file server and block server failures that can run as background processes while the file system continues serving incoming requests. Batch flushing of log records mitigates the performance impact of logging. Our design allows the batch flushing to be integrated with the file server block cache consistency scheme and lazy lock release scheme. Multiple identical block server replicas enable high performance file system read operations and provide availability in the face of failure. We evaluate their performance on our Afterlife prototype of three block servers and two file servers, and show that logging and data replication incurs only a factor of 10 performance versus a conventional NFS file server.

1 Introduction

We propose a design for Afterlife, a distributed file system that requires only a single log to recover from failures in both the storage layer and the file system layer. Multiple, identical block servers replicate this log for recovery purposes and also replicate metadata and user data to provide a highly available storage layer. A scalable number of Afterlife clients can interact with the file system layer via file servers that communicate with the storage layer. Afterlife tolerates up to n-1 failures of n block servers and m-1 failures of m file servers while maintaining metadata and user data consistency, assuming that no network partition occurs.

We design Afterlife to achieve two major recovery goals:

- After a file server crashes, any metadata and user data that it modified prior to the crash is restored to a consistent state on stable storage. We define consistency to mean that if a file server crashes while executing a sequence of client operations and the client sees the results of a particular operation on stable storage, then the client should also see the effects of all preceding operations.
- After a block server crashes and reboots, its copies of metadata, user data, and the log should be brought up-to-date and synchronized with other live block server replicas.

To accomplish these goals, we implement our file system update operations as atomic transactions using a single replicated log. Our major research contribution in Afterlife is the use of this unified redo log for crash recovery of both

the file system layer and the storage layer. The key to our solution is an innovative log-aware block server interface that enables block servers to distinguish between log data and file system data sent by file servers. This distinction allows block servers to follow a write-ahead logging protocol which, in conjunction with our data flushing protocol, guarantees that only committed data reaches stable storage. Other than this distinction, block servers treat log records as opaque data.

The performance impact of logging is mitigated by the batch flushing of log records from file servers to block servers and from each block server's in-memory cache to disk. Moreover, our recovery procedures read the log and run as background processes so that the file system can continue handling incoming client requests.

Data replication on multiple block servers eliminates the network bottleneck that would otherwise occur on file system read operations to centralized block servers. We instead distribute read operations to any available block server. Read operations on large data sets can be decomposed into smaller read operations that execute in parallel on multiple block servers to achieve higher performance. To maintain replica consistency, every block server must execute a given operation before the next operation can proceed.

A centralized lock server serializes concurrent file system operations via file-level shared read locks and exclusive write locks. To improve performance and minimize communication overhead, file servers use a lazy lock release scheme and a write-back data cache to reduce the load on the lock server and block servers, respectively.

The rest of this paper is organized as follows. We survey related work in Section 2. In Section 3, we describe the design and architecture of Afterlife, followed by a detailed discussion of our unified logging approach and recovery mechanisms in Section 4. We discuss the current status of the implementation and preliminary performance results of our Afterlife prototype in Section 5. Finally, we conclude with a summary of our contributions in Section 6.

2 Related Work

Existing distributed file systems address recovery for different layers by using multiple distinct logs. For instance, in the Frangipani file system [7], the Petal [8] storage layer maintains its own log for block server recovery while each Frangipani file server stores a separate recovery log as opaque data on Petal servers to handle file server failures. Petal replicates both types of logs for reliability. The use of multiple logs for recovery of different components is redundant and unnecessary, and stems partly from inadequate integration between the file system and storage layers. Afterlife, on the other hand, represents an integrated solution that requires only a single replicated log to recover from failures in both the file system layer and the storage layer.

Unlike the Petal [8] storage layer, which fragments its data across different Petal servers, Afterlife's storage layer uses identical block server replicas to address fault tolerance and simplify the recovery process. Afterlife's storage layer provides a block-level interface that replicates file data and metadata across multiple block servers to provide availability and delivers high performance to a large number of clients using a multi-reader/single-writer locking mechanism.

Ivy [9] is a peer-to-peer file system that serializes file system operations without locks and relies on a set of per-client logs to maintain the integrity of metadata. Data exists only in logs, and there is no explicit storage of file and directory objects. File System for Dragon (FSD) [6] uses a redo log and group commit to facilitate fast file recovery from memory and limited hard disk sector failures. Afterlife maintains a single write-ahead redo log to limit the need for conflict resolution among log updates during recovery.

Other file systems, such as Harp [1], use a primary copy replication scheme. Harp's replication scheme requires a two-phase protocol to coordinate block servers during modification operations. It tolerates network partition and replica failure through a view change algorithm based on Paxos.

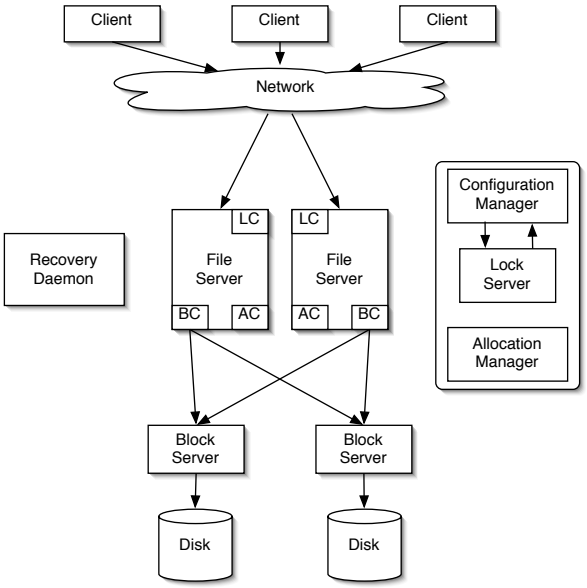

Figure 1: Architectural Design for Afterlife.

3 Architectural Design

The Afterlife file system architecture, as shown in Figure 1, consists of the block servers, file servers, a configuration manager, an allocation manager, and a lock server. Afterlife clients are standard NFS clients that connect to any of the available file servers to access the file system. Clients may connect to a file server over the network as depicted in the figure or to a loopback file server on their local machine. This section describes each of the system components in detail.

3.1 Block Servers

Block servers in Afterlife are replicas that store identical copies of file system metadata and user data at identical physical locations on their local disks. Together, they expose a highly available, replicated storage layer that can function correctly as long as at least one block server remains running. File servers interact with the storage layer via remote procedure calls (RPCs).

To support basic storage operations, each block server exposes a simple block interface consisting of the following RPCs: `put(key, value)`, `get(key)`, and `remove(key)`. The `key` parameter specifies a physical block number on disk, and the `value` parameter specifies a sequence of bytes that fit within a fixed block size. For our current prototype, we have selected a block size of 8 KB to match the block size used by the operating system.

To support write-ahead logging, we make the block server interface log-aware by introducing an `appendLog(data)` RPC, which appends the bytes specified in `data` to the end of a single, unified log that is replicated on every block server. Each block server stores its own copy of the log in a fixed location on disk so that the recovery mechanism always knows where to find the log after a crash. We describe the details of the log data in Section 3.2.

In order to achieve higher read/write performance, each block server maintains an in-memory cache of recently read or written blocks as well as recent log data. The previously mentioned RPCs actually interact with the cache rather than directly with the disk. A separately scheduled event periodically appends the in-memory log data to the tail of the on-disk log, in the order that the log data was received, and then flushes any dirty data from the block cache to disk. As this event occurs, the block server delays handling any `put` RPCs to in-memory blocks, in order to prevent blocks from being modified between the flushing of the log and the flushing of dirty blocks.

As long as we guarantee that Afterlife file servers only send updates to block servers for committed transactions, and as long as block servers always flush in-memory log records before writing modified data, block servers will never write any uncommitted data to disk. We discuss this guarantee in Section 3.2.3

3.2 File Servers

File servers in Afterlife act as NFS servers that provide an interface between Afterlife clients and the storage layer. To simplify the mapping from file handles to storage blocks, Afterlife stores physical block identifiers in the file handles sent out to clients.

As illustrated in Figure 1, each file server contains three modules to facilitate interaction with the rest of the system: an allocation client, a block client, and a lock client. The allocation client communicates with a centralized allocation manager in order to acquire new physical block identifiers for free blocks and to deallocate removed blocks. The block client exposes a more intelligent log-aware interface to support transactions and interacts with the block server through the RPCs specified in the previous section. The lock client interacts with a centralized lock server in order to serialize concurrent file server operations.

In this section, we describe the block structure that file servers use, the block client interface, how we achieve atomicity via logging, and conclude with a cache consistency and lazy lock release scheme that we support for improved performance.

3.2.1 Block Structure

As illustrated in Figure 2, Afterlife uses three types of block structures to store file system data. Inode blocks contain the metadata (represented as an NFS `fattr3` object), an integer denoting the number of valid bytes in the block, followed by a list of physical block identifiers that point to the data blocks of that file. File data blocks consist purely of user data written to the file.

Each directory contains a list of directory entries, which can either be files or additional directories. Afterlife stores directory entries in directory data blocks that use a slightly different block representation. Directory data blocks store an integer representing the number of valid bytes in the block. Each directory entry stores a variable-length filename, the filename length, and the corresponding file handle for that file.

3.2.2 Block Client Interface

The block client wraps the basic set of `put`, `get`, and `remove` RPCs as function calls with callbacks for the file servers. To improve file server read/write performance, the block client caches

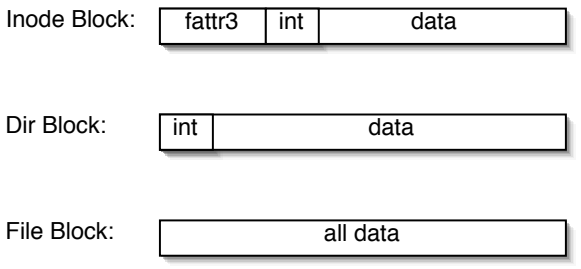

Figure 2: Different block structures found in Afterlife

recently used blocks rather than always communicating with the block servers. We discuss how we achieve cache consistency in Section 3.2.4.

The block client also propagates a more intelligent logging interface to the file server to support transactions, consisting of the following operations: `logBegin(xid)` , `logUpdate(xid, block-addr, data)` , and `logCommit(xid)` . The parameter `xid` is a transaction identifier that is uniquely generated for each transaction by concatenating the current value of a local transaction counter to the file server's local identifier. The parameter `block-addr` contains a physical block identifier and `data` contains data of the fixed block size.

The logging functions generate log records that the block client appends to a cached copy of its log tail. To increase log-writing performance, the block client accumulates several log entries in its cache before batch sending them to block servers via `appendLog` RPCs.

In order to synchronize the state of all the block servers, the block client sends each `put`, `remove`, and `appendLog` RPC to all the live and recovering block servers in the current block server configuration and waits for responses from all the live block servers before invoking the corresponding callback to return control to the file server. We discuss block server configurations in more detail in Section 3.4. Log records for a given transaction arrive in order at all the block servers, but log records across transactions may arrive in different orders at different block servers.

To handle reads, on the other hand, a block client sends a `get` RPC to a random live block

server; the randomness ensures load balancing in the face of multiple file servers. Afterlife uses timeouts on the RPCs in order to detect block server failures.

3.2.3 Logging Protocol

We call an NFS operation that is not read-only an *update* operation. These operations include `setattr`, `create`, `write`, `rename`, `mkdir`, and `remove`. To provide recoverability of update operations with respect to failure, we treat each update operation as a transaction and write all block modifications to the redo log to ensure atomicity.

Specifically, file servers follow the following protocol for any update operations, where each function call refers to the one exposed by the block client interface:

1. Initiate the transaction with a call to `logBegin`.
2. Acquire any locks needed to complete the operation.
3. For each new block that must be put, log the modified block via a call to `logUpdate` and then `put` the block.
4. End the transaction with a call to `logCommit`.
5. Release any locks that have been acquired.

On top of this protocol, we further guarantee that a block client only sends a `put` RPC for a modified block only after 1) any transaction dealing with that block has committed and after 2) it has sent `appendlog` RPCs for its entire cached log tail. This guarantee ensures that a block server's periodic flushing of log records and dirty data to disk will never write to disk any data not covered by a committed transaction on the block server's on-disk log. We explain how we achieve this guarantee in the next section.

3.2.4 Cache Consistency and Lazy Lock Release

Afterlife tightly couples its cache consistency scheme with a lazy lock release mechanism, in a manner similar to that of Frangipani [7]. As previously mentioned, the block client caches all recently read and written blocks as well as the recent log entries; similarly, the lock client also caches all lock releases. If a lock server revokes a lock (because another file server wishes to acquire it), the revoke triggers the following three ordered events in the *lock release protocol*:

1. The block client sequentially flushes all log entries from its cache to the block servers, in the order that the log entries were created.
2. If the file server has modified any blocks guarded by the lock, the block client flushes those modified blocks to the block servers.
3. The lock client releases the lock.

Because the logging protocol from the previous section requires that a file server release a lock into the lock client cache only after transactions involving the locked file have completed, we are ensured that any `commit` records for block modifications in that file have already been appended to the log in the block client cache. Therefore, the lock release protocol guarantees that the block clients only flush committed data to the block servers and that the `commit` record will be flushed before the actual modified data in a write-ahead logging fashion.

3.3 Lock Server

A centralized lock server serializes concurrent file system operations via file-level locks. We name a lock for a file according to the file handle of the file's inode in order to simplify the design; the file handle, and hence the lock name, contains the physical block identifier for the file's inode.

The lock server manages two types of locks: *read locks* and *write locks*. Read locks are *shared* and write locks are *exclusive*. Multiple clients

can simultaneously hold read locks to the same file and have read-only access. If a client holds an exclusive write lock on a file, it has both read and write access, and no other client can hold a lock to the same file.

Before a file server reads any metadata or data block, it must obtain a shared read lock for the block's associated file, which is granted after any outstanding write lock on that file has been downgraded. Before the file server may write to a file block, it must ask the lock server for an exclusive write lock to the file, which is granted after all read and write locks for the file have been revoked.

The lock server uses four different RPCs: `acquire`, `release`, `grant`, and `revoke`. A lock client sends `acquire` and `release` RPCs to the lock server, and the lock server sends `revoke` and `grant` RPCs to to the clients. To maintain reliability in the face of failure, a primary/secondary replica scheme can be designed as a future extension for the lock server.

3.4 Configuration Manager

The centralized configuration manager facilitates communication between file servers and the available block servers. It maintains the current view of all block servers by tracking each block server's status as either *live*, *recovering*, or *crashed*. File servers notify the configuration manager of any crashed block servers when a block client-related RPC times out, and the configuration manager propagates information regarding the current block server configuration to all other file servers.

A file server uses knowledge of the block server configuration to determine which block servers to send block client RPCs to. A file server can direct a `get` RPC can be directed to any live block server. A `put` or `remove` RPC must be directed to block servers with either live or recovering statuses, but file servers do not need to wait for responses from recovering block servers prior to proceeding.

The configuration manager achieves two purposes through the maintenance of this information. First, by notifying file servers that a particular block server has crashed and is no longer available, it offers a performance hint to the file servers so that they do not needlessly wait for a time-out to detect that the same block server has crashed. Second, the *recovering* status flag serves a recovery purpose and enables a recovering block server to queue up write requests, which is necessary for block server recovery (described in Section 4.3).

An important consideration in the design of the configuration manager is the minimization of communication overhead while maintaining a timely view of block servers. To solve this problem, we establish a tight coupling between the lock server and the configuration manager. We augment each `grant` RPC with a current list of live and recovering block servers. We augment each `release` RPC with a current list of crashed block servers that the file servers discover through timeout mechanisms in their block client modules. Our lazy lock release policy guarantees eventual updates to the configuration manager's block server list. This eventual consistency is acceptable because the configuration manager only hints for performance.

3.5 Allocation Manager

File server operations such as `create` and `remove` require allocation or reclaiming of storage blocks for metadata and user data. The centralized allocation manager maintains a global *free list* for all block server replicas. A file server sends `allocate` and `free` RPCs to the allocation manager.

If the allocation manager crashes, the allocation manager can reconstruct the block server's free list upon recovery by traversing the file system blocks starting at the root and designating all untraversed blocks as free. We do not handle failure of the allocation manager in the current design, but it can be augmented with a primary/secondary replica scheme in the future.

4 Recovery

The major innovation of Afterlife is the use of a single, unified log for recovery of both the file

system layer and the storage layer. For our preliminary design prototype, we assume the absence of network partitions, and we also assume that all failures of system components are fail-stop in order to focus on other issues related to a coherent approach to logging. Systems such as Harp [1] and Fab [3] already offer solutions based on configuration management and view changes instead of assuming fail-stop behavior. A potential design extension would be to apply similar techniques to Afterlife.

The design challenge in Afterlife is to recover both file servers and block servers from the same log. If a file server crashes in the middle of some operations, the metadata and data that it was in the process of updating must be restored to a consistent state. The locks it held must also be released before other file servers can read or write the data. For this problem, we employ a Frangipani-like approach to scan the log, redo any committed operations not reflected on stable storage, and finally release the locks.

If a block server crashes and reboots, its data must be synchronized with other replicas and brought up-to-date before it can serve further requests.

In this section, we describe the use of checkpoints, the recovery procedure for crashed file servers, and the recovery procedure for crashed block servers.

4.1 Checkpoints

After flushing dirty blocks to disk, a block server writes a `checkpoint` log record to disk to indicate that all data from committed transactions have been flushed. Each `checkpoint` record contains a list of all pending transactions at the time of the checkpoint. Any log record preceding the earliest `begin` record for pending transactions stored in the `checkpoint` may be garbage-collected. Block servers can thus periodically recover unneeded log space.

4.2 File Server Recovery

If a file server crashes while it is holding locks, it may not flushed all committed changes to the block server. For example, the log on the block servers may contain log records for a committed transaction of the NFS operation `create(dirfh, file.txt, 0666)`, but the file server may only have written the file inode to the block server but not yet sent the corresponding update to the directory entries in the directory's inode. A recovery algorithm would need to restore the system to a consistent state by finishing the partially completed operation.

Afterlife detects a crashed file server when a lock server revokes a lock from the file server but receives no response after a time-out period. Because we assume fail-stop behavior, we assume that the file server is actually down and not temporarily out of service due to an overloaded network. If no locks were held by the crashed file server, no recovery is necessary because the file server was not modifying any data at the point of the crash.

After detection of a crashed file server, the lock server boots up a recovery daemon. From the protocols specified in Sections 3.2.3 and 3.2.4, we know that the block server does not see changes from uncommitted transactions, so any of the file server's uncommitted transactions have been automatically aborted. The recovery daemon must only redo any committed transactions not reflected on disk.

Our recovery algorithm consists of two phases and is detailed in Figure 3. In the first phase, the recovery daemon asks the lock server for the set of write locks held by the crashed file server. Because we name the locks based on physical block identifiers of file inodes, the recovery daemon can use the log and the block servers to compute the set of *recovery blocks*, including both the inode blocks and corresponding data blocks, that were locked by the crashed file server. The log is necessary because file servers may have logged modifications inode blocks but not yet flushed the blocks to the block server; thus, the log may contain more recent information. We use the block server retrieve inodes prior to the checkpoint from the block server to improve recovery performance. These recovery blocks form a superset of all the blocks that the file server may have modified as part of a committed transaction but only yet flushed to the

```
locks := all write locks held by crashed file server
recovery-blocks := {}
xids-to-process := {}

// Phase 1:  Compute set of recovery blocks.
for each record r from end of log until checkpoint {
  if r is a COMMIT record
    xids-to-process := xids-to-process ∪ r.xid
  if r is a BEGIN record
    xids-to-process := xids-to-process - r.xid
  if r is an UPDATE record and
    r.xid ∈ xids-to-process and
    r.block-addr ∈ locks {
      recovery-blocks =
        recovery-blocks ∪ filehandles in r.data's inode
      locks := locks - r.block-addr
  }
}
for each lock l in locks {
  recovery-blocks := recovery-blocks ∪
    filehandles in inode at physical block id lock.name
}

// Phase 2:  Redo last update of recovery blocks
// in committed transactions.
checkpoint-found = false

for each record r from end of log until beginning {
  if recovery-blocks = {} or
      (checkpoint-found and xids = {})
    exit loop
  if r is a COMMIT record
    xids := xids ∪ r.xid
  if r is a BEGIN record
    xids := xids - r.xid
  if r is an UPDATE record and r.xid ∈ xids
    if r.block-addr ∈ blocks
      set value of block at r.block-addr to r.value
      blocks := blocks - r.block-addr
  if r is a CHECKPOINT record
    checkpoint-found := true
}
```

Figure 3: File Server Recovery Algorithm.

block servers.

In the second phase, the recovery daemon scans the log backward from the end to the most recent checkpoint and reapplies the last update for each recovery block that participated in any committed transactions. Because all transactions committed before the the most recent checkpoint have already been flushed to disk, no updates of committed transactions prior to the most recent update need to be reapplied. However, recovery blocks whose most recent updates belong to transactions that began prior to the checkpoint and committed after the checkpoint still need to be reapplied.

The recovery daemon runs as a background process and allows the rest of the Afterlife file system to continue servicing requests dealing with files not locked by the crashed file server.

4.3 Block Server Recovery

Upon rebooting, a crashed block server resubscribes itself to the configuration manager, which updates the block server's state from *crashed* to *recovering* and allows it to receive `put` and `remove` RPCs. These pending updates are stored in two data structures, a table that maps block identifiers to block updates and a queue of log entries. Behavior during the next stage of recovery depends on the block server's downtime.

The recovering block server sends a request to a live replica for a copy of the log. A server that has experienced a brief outage only needs to append the tail of this log to its own and roll forward all committed transactions from the last checkpoint. It determines the tail of the log by comparing transaction identifiers and finding the point in log that contains updates that it has missed. For longer outages in which no overlap exists between the recovering server's log and the live replica's log, the recovering server needs a copy of the entire log from the replica. It then copies over all the data from the live replica. Although this will still result in an inconsistent snapshot of data, all changes that need to be applied can be found within the restored log and queued updates. Upon completing the data transfer, the recovering block server rolls forward on the original log entries.

In either scenario, once the block server has finished rolling forward, it determines where the updated log and queued log entries overlap and appends the remaining queued entries onto its log. This log is then flushed to disk. As a final step, it flushes the updated blocks found in the table data structure to disk. The block server has now finished recovery and can notify the configuration manager that it is live.

	Create	Lookup	R/W
NFS (lab machine)	0.29s	0.01s	0.94s
Lab FS	3.25s	0.13s	12.55s
Afterlife (1 bsrv)	1.59s	0.13s	13.03s
Afterlife (2 bsrv)	2.54s	0.13s	20.49s
Afterlife (3 bsrv)	3.50s	0.13s	28.86s

Table 1: Connectathon benchmark performance results for the NFS file system running on 6.824 lab machines, the 6.824 lab file system (Lab FS), and our Afterlife prototype running on 1, 2, and 3 block servers.

	Throughput
NFS (lab machine)	11.51 MB/s
Lab FS	994.34 KB/s
Afterlife (1 bsrv)	971.34 KB/s
Afterlife (2 bsrv)	577.32 KB/s
Afterlife (3 bsrv)	413.69 KB/s

Table 2: Throughput performance for repeated writes of a 1 MB file using the Afterlife prototype, the NFS file system on 6.824 lab machines, and the 6.824 lab file system (Lab FS).

5 Prototype Implementation and Performance

We have evaluated a preliminary Afterlife file system prototype consisting of three nodes. At present, two of the nodes are dual-CPU machines with 1.2 GHz AMD and 696 MHz Intel processors and with 1 GB and 256 MB of RAM, respectively. The third node is a 300 MHz Intel processor with 256 MB of RAM. All three run FreeBSD 4.9.

We implemented our Afterlife prototype using 10,800 lines of C++ code. We implement the design described in Section 3 with the proposed disk layout with fixed 8 KB block sizes for metadata and data blocks. The prototype supports a scalable number of block servers and file servers. Transactions are fully supported, and we have implemented the file server recovery algorithm but not the block server recovery algorithm. We do not currently support read/write locks. We have implemented a simple allocation manager that does not handle block deallocation.

To evaluate system performance, we ran a modified subset of the Connectathon test suite [2] to benchmark operations supported by our Afterlife prototype. We ran three tests: a create test, a lookup test, and a read/write test. The create test created 155 files in 62 directories with a directory structure that was 5 levels deep. The lookup test performed lookups across the file system mount point with 500 `getcwd` and `stat` calls. The read/write test wrote ten 1 MB files and read the ten files.

Table 1 summarizes the performance benchmark results. We compare the performance of our Afterlife prototype using 1-3 block servers with that of the NFS file system and the file system created in the 6.824 lab assignments. Our Afterlife prototype achieves comparable performance relative to the lab file servers but is an order of magnitude worse in performance than the NFS file system. We also see a linear degradation for write operations as the number of block servers increases; we attribute this result to our implementation, which waits for a response to a `put` RPC from one block server before sending one to the next.

Table 2 summarizes the throughput performance for repeated writes of a 1 MB file using Afterlife, NFS, and the 6.824 lab file system. Our analysis indicates that the Afterlife prototype suffers a substantial amount of disk I/O overhead due to logging and flushing.

6 Contributions

We believe that data replication and logging can be used to design a distributed, highly available, and recoverable file system, and we have taken steps to validate this belief through our design for the Afterlife file system. Our experience illustrates that the maintenance of multiple logs for different system layers may in fact be an artifact of over-engineering and that a single, unified approach to logging is indeed possible.

Our contributions include:

1. Designing a distributed and recoverable file system that uses a single, unified log for

crash recovery in both the file system layer and block storage layer.

2. Specifying logging protocols and lock release protocols that together ensure that only data covered by committed transactions in a block server's log ever reaches stable storage.

3. Implementing an Afterlife prototype to demonstrate the feasibility of our file server recovery scheme.

4. Evaluating the performance of our Afterlife prototype to show that the overhead of logging and replication only decreases file system performance by roughly a factor of 10.

References

[1] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. "Replication in the Harp File System." *Proc. of the 13th ACM Symposium on Operating Systems Principles*, p226-238, October 1991.

[2] Connectathon NFS Testsuite. `http://www.connectathon.org/nfstests.html`.

[3] Y. Saito, S. Frlund, A. Veitch, A. Merchant, and S. Spence. "FAB: Building Distributed Enterprise Disk Arrays from Commodity Components." *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[4] Z. Dubitzsky, I. Gold, E. Henis, J. Satran, and D. Scheinwald. "DSF: Data Sharing Facility." Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2000. See also: `http://www.haifa.il.ibm.com/projects/systems/papers/DSF.pdf`

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. "Serverless network file systems." *Proc. of the 15th ACM Symposium on Operating System Principles*, p109-126, 1995.

[6] R. Hagmann. "Reimplementing the Cedar file system using logging and group commit." *Proc. of the 11th ACM Symp. on Operating Systems Principles*, p155-162, November 1987.

[7] C. Thekkath, T. Mann, and E. Lee. "Frangipani: A scalable distributed file system." *Proc. of the 16th ACM Symposium on Operating System Principles*, p224-237, 1997.

[8] E. Lee and C. Thekkath. "Petal: Distributed virtual disks." *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, p84-92, October 1996.

[9] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. "Ivy: A Read/Write Peer-to-Peer File System." *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, 2002.