Comparison-based File Server Verification

Yuen-Lin Tan*, Terrence Wong, John D. Strunk, Gregory R. Ganger
Carnegie Mellon University

Abstract

Comparison-based server verification involves testing a server by comparing its responses to those of a reference server. An intermediary, called a "server Tee," interposes between clients and the reference server, synchronizes the system-under-test (SUT) to match the reference server's state, duplicates each request for the SUT, and compares each pair of responses to identify any discrepancies. The result is a detailed view into any differences in how the SUT satisfies the client-server protocol specification, which can be invaluable in debugging servers, achieving bug compatibility, and isolating performance differences. This paper introduces, develops, and illustrates the use of comparison-based server verification. As a concrete example, it describes a NFSv3 Tee and reports on its use in identifying interesting differences in several production NFS servers and in debugging a prototype NFS server. These experiences confirm that comparison-based server verification can be a useful tool for server implementors.

1 Introduction

Debugging servers is tough. Although the client-server interface is usually documented in a specification, there are often vague or unspecified aspects. Isolating specification interpretation flaws in request processing and in responses can be a painful activity. Worse, a server that works with one type of client may not work with another, and testing with all possible clients is not easy.

The most common testing practices are RPC-level test suites and benchmarking with one or more clients. With enough effort, one can construct a suite of tests that exercises each RPC in a variety of cases and verifies that each response conforms to what the specification dictates. This is a very useful approach, though time-consuming to develop and difficult to perfect in the face of specification vagueness. Popular benchmark programs, such as SPEC SFS [15] for NFS servers, are often used to stress-test servers and verify that they work for the clients used in the benchmark runs.

This paper proposes an additional tool for server testing: *comparison-based server verification*. The idea is simple: each request is sent to both the system-under-test (SUT) and a *reference server*, and the two responses are compared. This can even be done in a live environment with real clients to produce scenarios that artificial test suites may miss. The reference server is chosen based on the belief that it is a valid implementation of the relevant interface specification. For example, it might be a server that has been used for some time by many user communities. The reference server thus becomes a "gold standard" against which the SUT's conformity can be evaluated. Given a good reference server, comparison-based server verification can assist with debugging infrequent problems, achieving "bug compatibility," and isolating performance differences.

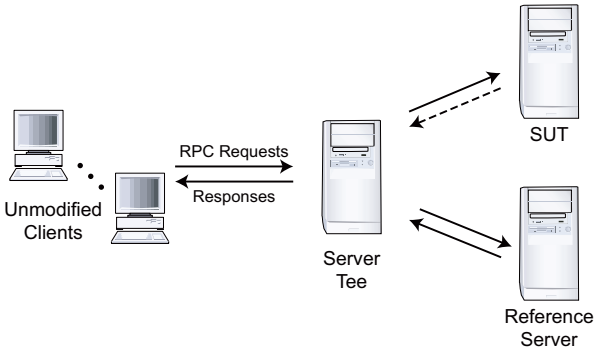

Figure 1: **Using a server Tee for comparison-based verification.** The server Tee is interposed between unmodified clients and the unmodified reference server, relaying requests and responses between them. The Tee also sends the same requests to the system-under-test and compares the responses to those from the reference server. With the exception of performance interference, this latter activity should be invisible to the clients.

This paper specifically develops the concept of comparison-based verification of file servers via use of a *file server Tee* (See Figure 1).[1] A file server Tee interposes on communication between clients and the reference server. The Tee automatically sets and maintains SUT state (i.e., directories, files, etc.) to match the reference server's state, forwards client requests to the reference server, duplicates client requests for the SUT, and compares the two responses for each request. Only the reference server's responses are sent to clients, which

*Currently works for VMware.

[1] The name, "server Tee," was inspired by the UNIX `tee` command, which reads data from standard input and writes it to both standard output and one or more output files.

makes it possible to perform comparison-based verification even in live environments.

The paper details the design and implementation of a NFSv3 Tee. To illustrate the use of a file server Tee, we present the results of using our NFSv3 Tee to compare several popular production NFS servers, including FreeBSD, a Network Appliance box, and two versions of Linux. A variety of differences are identified, including some discrepancies that would affect correctness for some clients. We also describe experiences using our NFSv3 Tee to debug a prototype NFS server.

The remainder of this paper is organized as follows. Section 2 puts comparison-based server verification in context and discusses what it can be used for. Section 3 discusses how a file server Tee works. Section 4 describes the design and implementation of our NFSv3 Tee. Section 5 evaluates our NFSv3 Tee and presents results of several case studies using it. Section 6 discusses additional issues and features of comparison-based file server verification. Section 7 discusses related work.

2 Background

Distributed computing based on the client-server model is commonplace. Generally speaking, this model consists of clients sending RPC requests to servers and receiving responses after the server finishes the requested action. For most file servers, for example, system calls map roughly to RPC requests, supporting actions like file creation and deletion, data reads and writes, and fetching of directory entry listings.

Developing functional servers can be fairly straightforward, given the variety of RPC packages available and the maturity of the field. Fully debugging them, however, can be tricky. While the server interface is usually codified in a specification, there are often aspects that are insufficiently formalized and thus open to interpretation. Different client or server implementors may interpret them differently, creating a variety of de facto standards to be supported (by servers or clients).

There are two common testing strategies for servers. The first, based on RPC-level test suites, exercises each individual RPC request and verifies proper responses in specific situations. For each test case, the test scaffolding sets server state as needed, sends the RPC request, and compares the response to the expected value. Verifying that the RPC request did the right thing may involve additional server state checking via follow-up RPC requests. After each test case, any residual server state is cleaned up. Constructing exhaustive RPC test suites is a painstaking task, but it is a necessary step if serious robustness is desired. One challenge with such test suites, as with almost all testing, is balancing coverage with development effort and test completion time. Another challenge, related to specification vagueness, is accuracy: the test suite implementor interprets the specification, but may not do so the same way as others.

The second testing strategy is to experiment with applications and benchmarks executing on one or more client implementation(s).[2] This complements RPC-level testing by exercising the server with specific clients, ensuring that those clients work well with the server when executing at least some important workloads; thus, it helps with the accuracy issue mentioned above. On the other hand, it usually offers much less coverage than RPC-level testing. It also does not ensure that the server will work with clients that were not tested.

2.1 Comparison-based verification

Comparison-based verification complements these testing approaches. It does not eliminate the coverage problem, but it can help with the accuracy issue by conforming to someone else's interpretation of the specification. It can help with the coverage issue, somewhat, by exposing problem "types" that recur across RPCs and should be addressed en masse.

Comparison-based verification consists of comparing the server being tested to a "gold standard," a reference server whose implementation is believed to work correctly. Specifically, the state of the SUT is set up to match that of the reference server, and then each RPC request is duplicated so that the two servers' responses to each request can be compared. If the server states were synchronized properly, and the reference server is correct, differences in responses indicate potential problems with the SUT.

Comparison-based verification can help server development in four ways: debugging client-perceived problems, achieving bug compatibility with existing server implementations, testing in live environments, and isolating performance differences.

1. Debugging: With benchmark-based testing, in particular, bugs exhibit themselves as situations where the benchmark fails to complete successfully. When this happens, significant effort is often needed to determine exactly what server response(s) caused the client to fail. For example, single-stepping through client actions might be used, but this is time-consuming and may alter client behavior enough that the problem no longer arises. Another approach is to sniff network packets and interpret the exchanges between client and server to identify the last interactions before problems arise. Then, one

[2] Research prototypes are almost always tested only in this way.

can begin detailed analysis of those RPC requests and responses.

Comparison-based verification offers a simpler solution, assuming that the benchmark runs properly when using the reference server. Comparing the SUT's responses to the problem-free responses produced by the reference server can quickly identify the specific RPC requests for which there are differences. Comparison provides the most benefit when problems involve nuances in responses that cause problems for clients (as contrasted with problems where the server crashes)—often, these will be places where the server implementors interpreted the specification differently. For such problems, the exact differences between the two servers' responses can be identified, providing detailed guidance to the developer who needs to find and fix the implementation problem.

2. Bug compatibility: In discussing vagueness in specifications, we have noted that some aspects are often open to interpretation. Sometimes, implementors misinterpret them even if they are not vague. Although it is tempting to declare both situations "the other implementor's problem," that is simply not a viable option for those seeking to achieve widespread use of their server. For example, companies attempting to introduce a new server product into an existing market **must** make that server work for the popular clients. Thus, deployed clients introduce de facto standards that a server must accommodate. Further, if clients (existing and new) conform to particular "features" of a popular server's implementation (or a previous version of the new server), then that again becomes a de facto standard. Some use the phrase, "bug compatibility," to describe what must be achieved given these issues.

As a concrete example of bug compatibility, consider the following real problem encountered with a previous NFSv2 server we developed: Linux clients (at the time) did not invalidate directory cookies when manipulating directories, which our interpretation of the specification (and the implementations of some other clients) indicated should be done. So, with that Linux client, an "rm -rf" of a large directory would read part of the directory, remove those files, and then do another READDIR with the cookie returned by the first READDIR. Our server compressed directories when entries were removed, and thus the old cookie (an index into the directory) would point beyond some live entries after some files were removed—the "rm -rf" would thus miss some files. We considered keeping a table of cookie-to-index mappings instead, but without a way to invalidate entries safely (there are no definable client sessions in NFSv2), the table would have to be kept persistently; we finally just disabled directory compression. (NFSv3 has a "cookie verifier," which would allows a server to solve this problem, even when other clients change the directory.)

Comparison-based verification is a great tool for achieving bug compatibility. Specifically, one can compare each response from the SUT with that produced by a reference server that implements the de facto standard. Such comparisons expose differences that might indicate differing interpretations of the specification or other forms of failure to achieve bug compatibility. Of course, one needs an input workload that has good coverage to fully uncover de facto standards.

3. In situ verification: Testing and benchmarking allow offline verification that a server works as desired, which is perfect for those developing a new server. These approaches are of less value to IT administrators seeking comfort before replacing an existing server with a new one. In high-end environments (e.g., bank data centers), expensive service agreements and penalty clauses can provide the desired comfort. But, in less resource-heavy environments (e.g., university departments or small businesses), administrators often have to take the plunge with less comfort.

Comparison-based verification offers an alternative, which is to run the new server as the SUT for a period of time while using the existing server as the reference server.[3] This requires inserting a server Tee into the live environment, which could introduce robustness and performance issues. But, because only the reference server's responses are sent to clients, this approach can support reasonably safe in situ verification.

4. Isolating performance differences: Performance comparisons are usually done with benchmarking. Some benchmarks provide a collection of results on different types of server operations, while others provide overall application performance for more realistic workloads.

Comparison-based verification could be adapted to performance debugging by comparing per-request response times as well as response contents. Doing so would allow detailed request-by-request profiles of performance differences between servers, perhaps in the context of application benchmark workloads where disappointing overall performance results are observed. Such an approach might be particularly useful, when combined with in situ verification, for determining what benefits might be expected from a new server being considered.

[3] Although not likely to be its most popular use, this was our original reason for exploring this idea. We are developing a large-scale storage service to be deployed and maintained on the Carnegie Mellon campus as a research expedition into self-managing systems [4]. We wanted a way to test new versions in the wild before deploying them. We also wanted a way to do live experiments safely in the deployed environment, which is a form of the fourth item.

3 Components of a file system Tee

Comparison-based server verification happens at an interposition point between clients and servers. Although there are many ways to do this, we believe it will often take the form of a distinct proxy that we call a "server Tee". This section details what a server Tee is by describing its four primary tasks. The subsequent section describes the design and implementation of a server Tee for NFSv3.

Relaying traffic to/from reference server: Because it interposes, a Tee must relay RPC requests and responses between clients and the reference server. The work involved in doing so depends on whether the Tee is a passive or an active intermediary. A passive intermediary observes the client-server exchanges but does not manipulate them at all—this minimizes the relaying effort, but increases the effort for the duplicating and comparing steps, which now must reconstruct RPC interactions from the observed packet-level communications. An active intermediary acts as the server for clients and as the only client for the server—it receives and parses the RPC requests/responses and generates like messages for the final destination. Depending on the RPC protocol, doing so may require modifying some fields (e.g., request IDs since all will come from one system, the Tee), which is extra work. The benefit is that other Tee tasks are simplified.

Whether a Tee is an active intermediary or a passive one, it must see all accesses that affect server state in order to avoid flagging false positives. For example, an unseen file write to the reference server would cause a subsequent read to produce a mismatch during comparison that has nothing to do with the correctness of the SUT. One consequence of the need for complete interposing is that tapping the interconnect (e.g., via a network card in promiscuous mode or via a mirrored switch port) in front of the reference server will not work—such tapping is susceptible to dropped packets in heavy traffic situations, which would violate this fundamental Tee assumption.

Synchronizing state on the SUT: Before RPC requests can be productively sent to the SUT, its state must be initialized such that its responses could be expected to match the reference server's. For example, a file read's responses won't match unless the file's contents are the same on both servers. Synchronizing the SUT's state involves querying the reference server and updating the SUT accordingly.

For servers with large amounts of state, synchronizing can take a long time. Since only synchronized objects can be compared, few comparisons can be done soon after a SUT is inserted. Requests for objects that have yet to be synchronized produce no useful comparison data. To combat this, the Tee could simply deny client requests until synchronization is complete. Then, when all objects have been synchronized, the Tee could relay and duplicate client requests knowing that they will all be for synchronized state. However, because we hope for the Tee to scale to terabyte- and petabyte-scale storage systems, complete state synchronization can take so long that denying client access would create significant downtime. To maintain acceptable availability, if a Tee is to be used for in situ testing, requests must be handled during initial synchronization even if they fail to yield meaningful comparison results.

Duplicating requests for the SUT: For RPC requests that can be serviced by the SUT (because the relevant state has been synchronized), the Tee needs to duplicate them, send them, and process the responses. This is often not as simple as just sending the same RPC request packets to the SUT, because IDs for the same object on the two servers may differ. For example, our NFS Tee must deal with the fact that the two file handles (reference server's and SUT's) corresponding to a particular file will differ; they are assigned independently by each server. During synchronization, any such ID mappings must be recorded for use during request duplication.

Comparing responses from the two servers: Comparing the responses from the reference server and SUT involves more than simple bitwise comparison. Each field of a response falls into one of three categories: bitwise-comparable, non-comparable, or loosely-comparable.

Bitwise-comparable fields should be identical for any correct server implementation. Most bitwise-comparable fields consist of data provided directly by clients, such as file contents returned by a file read.

Most non-comparable fields are either server-chosen values (e.g., cookies) or server-specific information (e.g., free space remaining). Differences in these fields do not indicate a problem, unless detailed knowledge of the internal meanings and states suggest that they do. For example, the disk space utilized by a file could be compared if both server's are known to use a common internal block size and approach to space allocation.

Fields are loosely-comparable if comparing them requires more analysis than bitwise comparison—the reference and SUT values must be compared in the context of the field's semantic meaning. For example, timestamps can be compared (loosely) by allowing differences small enough that they could be explained by clock skew, communication delay variation, and processing time variation.

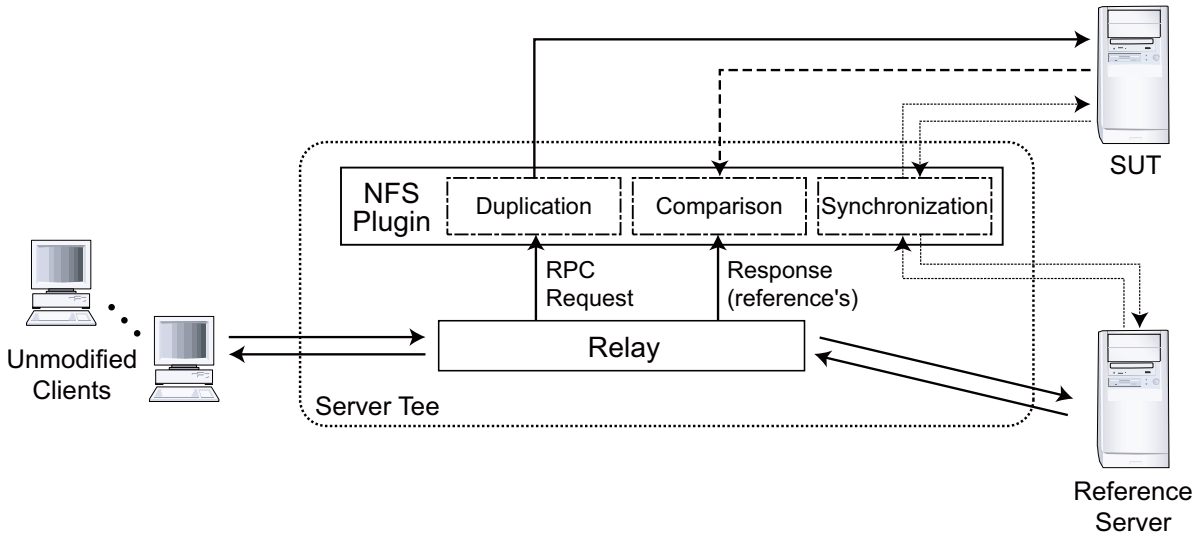

Figure 2: **Software architecture of an NFS Tee.** To minimize potential impact on clients, we separate the relaying functionality from the other three primary Tee functions (which contain the vast majority of the code). One or more NFS plug-ins can be dynamically initiated to compare a SUT to the reference server with which clients are interacting.

4 A NFSv3 Tee

This section describes the design and implementation of an NFSv3 Tee. It describes how components performing the four primary Tee tasks are organized and explains the architecture in terms of our design goals. It details nuanced aspects of state synchronization and response comparison, including some performance enhancements.

4.1 Goals and architecture

Our NFSv3 Tee's architecture is driven by five design goals. First, we want to be able to use the Tee in live environments, which makes the reliability of the relay task crucial. Second, we want to be able to dynamically add a SUT and initiate comparison-based verification in a live environment.[4] Third, we want the Tee to operate using reasonable amounts of machine resources, which pushes us to minimize runtime state and perform complex comparisons off-line in a post-processor. Fourth, we are more concerned with achieving a functioning, robust Tee than with performance, which guides us to have the Tee run as application-level software, acting as an active intermediary. Fifth, we want the comparison module to be flexible so that a user can customize of the rules to increase efficiency in the face of server idiosyncrasies that are understood.

Figure 2 illustrates the software architecture of our NFSv3 Tee, which includes modules for the four primary tasks. The four modules are partitioned into two processes. One process relays communication between clients and the reference server. The other process (a "plug-in") performs the three tasks that involve interaction with the SUT. The relay process exports RPC requests and responses to the plug-in process via a queue stored in shared memory. This two-process organization was driven by the first two design goals: (1) running the relay as a separate process isolates it from faults in the plug-in components, which make up the vast majority of the Tee code; (2) plug-ins can be started and stopped without stopping client interactions with the reference server.

When a plug-in is started, it attaches to the shared memory and begins its three modules. The synchronization module begins reading files and directories from the reference server and writing them to the SUT. As it does so, it stores reference server-to-SUT file handle mappings.

The duplication module examines each RPC request exported by the relay and determines whether the relevant SUT objects are synchronized. If so, an appropriate request for the SUT is constructed. For most requests, this simply involves mapping the file handles. The SUT's response is passed to the comparison module, which compares it against the reference server's response.

Full comparison consists of two steps: a configurable on-line step and an off-line step. For each mismatch found in the on-line step, the request and both responses are logged for off-line analysis. The on-line comparison rules are specified in a configuration file that describes how each response field should be compared. Off-line post-processing prunes the log of non-matching

[4] On a SUT running developmental software, developers may wish to make code changes, recompile, and restart the server repeatedly.

responses that do not represent true discrepancies (e.g., directory entries returned in different orders), and then assists the user with visualizing the "problem" RPCs. Off-line post-processing is useful for reducing on-line overheads as well as allowing the user to refine comparison rules without losing data from the real environment (since the log is a filtered trace).

4.2 State synchronization

The synchronization module updates the SUT to enable useful comparisons. Doing so requires making the SUT's internal state match the reference server's to the point that the two servers' responses to a given RPC could be expected to match. Fortunately, NFSv3 RPCs generally manipulate only one or two file objects (regular files, directories, or links), so some useful comparisons can be made long before the entire file system is copied to the reference server.

Synchronizing an object requires establishing a point within the stream of requests where comparison could begin. Then, as long as RPCs affecting that object are handled in the same order by both servers, it will remain synchronized. The lifetime of an object can be viewed as a sequence of states, each representing the object as it exists between two modifications. Synchronizing an object, then, amounts to replicating one such state from the reference server to the SUT.

Performing synchronization offline (i.e., when the reference server is not being used by any clients) would be straightforward. But, one of our goals is the ability to insert a SUT into a live environment at runtime. This requires dealing with object changes that are concurrent with the synchronization process. The desire not to disrupt client activity precludes blocking requests to an object that is being synchronized. The simplest solution would be to restart synchronization of an object if a modification RPC is sent to the reference server before it completes. But, this could lead to unacceptably slow and inefficient synchronization of large, frequently-modified objects. Instead, our synchronization mechanism tracks changes to objects that are being synchronized. RPCs are sent to the reference server as usual, but are also saved in a *changeset* for later replay against the SUT.

Figure 3 illustrates synchronization in the presence of write concurrency. The state S1 is first copied from the reference server to the SUT. While this copy is taking place, a write (Wr1) arrives and is sent to the reference server. Wr1 is not duplicated to the SUT until the copy of S1 completes. Instead, it is recorded at the Tee. When the copy of S1 completes, a new write, Wr1', is constructed based on Wr1 and sent to the SUT. Since no further concurrent changes need to be replayed, the object is marked

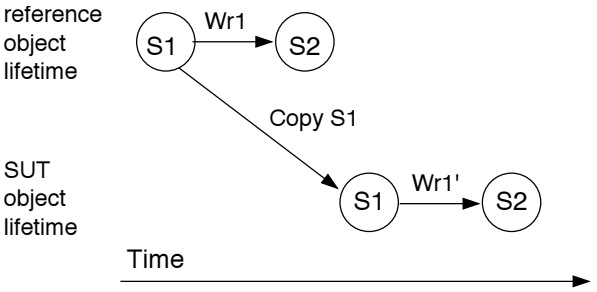

Figure 3: **Synchronization with a concurrent write.** The top series of states depicts a part of the lifetime of an object on the reference server. The bottom series of states depicts the corresponding object on the SUT. Horizontal arrows are requests executed on a server (reference or SUT), and diagonal arrows are full object copies. Synchronization begins with copying state S1 onto the SUT. During the copy of S1, write Wr1 changes the object on the reference server. At the completion of the copy of S1, the objects are again out of synchronization. Wr1' is the write constructed from the buffered version of Wr1 and replayed on the SUT.

synchronized and all subsequent requests referencing it are eligible for duplication and comparison.

Even after initial synchronization, concurrent and overlapping updates (e.g., Wr1 and Wr2 in Figure 4) can cause a file object to become unsynchronized. Two requests are deemed overlapping if they both affect the same state. Two requests are deemed concurrent if the second one arrives at the relay before the first one's response. This definition of concurrency accounts for both network reordering and server reordering. Since the Tee has no reliable way to determine the order in which concurrent requests are executed on the reference server, any state affected by both Wr1 and Wr2 is indeterminate. Resynchronizing the object requires re-copying the affected state from the reference server to the SUT. Since overlapping concurrency is rare, our Tee simply marks the object unsynchronized and repeats the process entirely.

The remainder of this section provides details regarding synchronization of files and directories, and describes some synchronization ordering enhancements that allow comparisons to start more quickly.

Regular file synchronization: A regular file's state is its data and its attributes. Synchronizing a regular file takes place in three steps. First, a small unit of data and the file's attributes are read from the reference server and written to the SUT. If a client RPC affects the object during this initial step, the step is repeated. This establishes a point in time for beginning the changeset. Second, the remaining data is copied. Third, any changeset entries are replayed.

A file's changeset is a list of attribute changes and written-to extents. A bounded amount of the written data

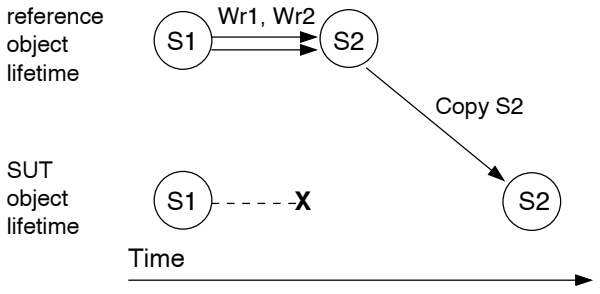

Figure 4: **Re-synchronizing after write concurrency.** The example begins with a synchronized object, which has state S1 on both servers. When concurrent writes are observed (Wr1 and Wr2 in this example), the Tee has no way of knowing their execution order at the reference server. As a consequence, it cannot know the resulting reference server state. So, it must mark the object as unsynchronized and repeat synchronization.

is cached. If more data was written, it must be read from the reference server to replay changes. As the changeset is updated, by RPCs to reference server, overlapping extents are coalesced to reduce the work of replaying them; so, for example, two writes to the same block will result in a single write to the SUT during the third step of file synchronization.

Directory synchronization: A directory's state is its attributes and the name and type of each of its children.[5] This definition of state allows a directory to be synchronized regardless of whether its children are synchronized. This simplifies the tracking of a directory's synchronization status and allows the comparison of responses to directory-related requests well before the children are synchronized.

Synchronizing a directory is done by creating missing directory entries and removing extraneous ones. Hard links are created as necessary (i.e., when previously discovered file handles are found). As each unsynchronized child is encountered, it is enqueued for synchronization. When updates occur during synchronization, a directory's changeset will include new attribute values and two lists: entries to be created and entries to be removed. Each list entry stores the name, file handle, and type for a particular directory entry.

Synchronization ordering: By default, the synchronization process begins with the root directory. Each unknown entry of a directory is added to the list of files to be synchronized. In this way, the synchronization process works its way through the entire reference file system.

One design goal is to begin making comparisons as quickly as possible. To accomplish this, our Tee synchronizes the most popular objects first. The Tee maintains a weighted moving average of access frequency for each object it knows about, identifying accesses by inspecting the responses to lookup and create operations. These quantities are used to prioritize the synchronization list. Because an object cannot be created until its parent directory exists on the SUT, access frequency updates are propagated from an object back to the file system root.

[5] File type is not normally considered to be part of a directory's contents. We make this departure to facilitate the synchronization process. During comparison, file type is a property of the file, not of the parent directory.

4.3 Comparison

The comparison module compares responses to RPC requests on synchronized objects. The overall comparison functionality proceeds in two phases: on-line and post-processed. The on-line comparisons are performed at runtime, by the Tee's comparison module, and any non-matching responses (both responses in their entirety) are logged together with the associated RPC request. The logged information allows post-processing to eliminate false non-matches (usually with more detailed examination) and to help the user to explore valid non-matches in detail.

Most bitwise-comparable fields are compared on-line. Such fields include file data, file names, soft link contents, access control fields (e.g., modes and owner IDs), and object types. Loosely-comparable fields include time values and directory contents. The former are compared on-line, while the latter (in our implementation) are compared on-line and then post-processed.

Directory contents require special treatment, when comparison fails, because of the looseness of the NFS protocol. Servers are not required to return entries in any particular order, and they are not required to return any particular number of entries in a single response to a READDIR or READDIRPLUS RPC request. Thus, entries may be differently-ordered and differently-spread across multiple responses. In fact, only when the Tee observes complete listings from both servers can some non-matches be definitively declared. Rather than deal with all of the resulting corner cases on-line, we log the observed information and leave it for the post-processor. The post-processor can link multiple RPC requests iterating through the same directory by the observed file handles and cookie values. It filters log entries that cannot be definitively compared and that do not represent mismatches once reordering and differing response boundaries are accounted for.

4.4 Implementation

We implemented our Tee in C++ on Linux. We used the State Threads user-level thread library. The relay runs

as a single process that communicates with clients and the reference server via UDP and with any plug-ins via a UNIX domain socket over which shared memory addresses are passed.

Our Tee is an active intermediary. To access a file system exported by the reference server, a client sends its requests to the Tee. The Tee multiplexes all client requests into one stream of requests, with itself as the client so that it receives all responses directly. Since the Tee becomes the source of all RPC requests seen by the reference server, it is necessary for the relay to map client-assigned RPC transaction IDs (XIDs) onto a separate XID space. This makes each XID seen by the reference server unique, even if different clients send requests with the same XID, and it allows the Tee to determine which client should receive which reply. This XID mapping is the only way in which the relay modifies the RPC requests.

The NFS plug-in contains the bulk of our Tee's functionality and is divided into four modules: synchronization, duplication, comparison, and the dispatcher. The first three modules each comprise a group of worker threads and a queue of lightweight request objects. The dispatcher (not pictured in Figure 2) is a single thread that interfaces with the relay, receiving shared memory buffers.

For each file system object, the plug-in maintains some state in a hash table keyed on the object's reference server file handle. Each entry includes the object's file handle on each server, its synchronization status, pointers to outstanding requests that reference it, and miscellanous book-keeping information. Keeping track of each object consumes 236 bytes. Each outstanding request is stored in a hash table keyed on the request's reference server XID. Each entry requires 124 bytes to hold the request, both responses, their arrival times, and various miscellanous fields. The memory consumption is untuned and could be reduced.

Each RPC received by the relay is stored directly into a shared memory buffer from the RPC header onward. The dispatcher is passed the addresses of these buffers in the order that the RPCs were received by the relay. It updates internal state (e.g., for synchronization ordering), then decides whether or not the request will yield a comparable response. If so, the request is passed to the duplication module, which constructs a new RPC based on the original by replacing file handles with their SUT equivalents. It then sends the request to the SUT.

Once responses have been received from both the reference server and the SUT, they are passed to the comparison module. If the comparison module finds any discrepancies, it logs the RPC and responses and optionally alerts the user. For performance and space reasons, the Tee discards information related to matching responses, though this can be disabled if full tracing is desired.

5 Evaluation

This section evaluates the Tee along three dimensions. First, it validates the Tee's usefulness with several case studies. Second, it measures the performance impact of using the Tee. Third, it demonstrates the value of the synchronization ordering optimizations.

5.1 Systems used

All experiments are run with the Tee on an Intel P4 2.4GHz machine with 512MB of RAM running Linux 2.6.5. The client is either a machine identical to the Tee or a dual P3 Xeon 600MHz with 512MB of RAM running FreeBSD 4.7. The servers include Linux and FreeBSD machines with the same specifications as the clients, an Intel P4 2.2GHz with 512MB of RAM running Linux 2.4.18, and a Network Appliance FAS900 series filer. For the performance and convergence benchmarks, the client and server machines are all identical to the Tee mentioned above and are connected via a Gigabit Ethernet switch.

5.2 Case studies

An interesting use of the Tee is to compare popular deployed NFS server implementations. To do so, we ran a simple test program on a FreeBSD client to compare the responses of the different server configurations. The short test consists of directory, file, link, and symbolic link creation and deletion as well as reads and writes of data and attributes. No other filesystem objects were involved except the root directory in which the operations were done. Commands were issued at 2 second intervals.

Comparing Linux to FreeBSD: We exercised a setup with a FreeBSD SUT and a Linux reference server to see how they differ. After post-processing READDIR and READDIRPLUS entries, and grouping like discrepancies, we are left with the nineteen unique discrepancies summarized in Table 1. In addition to those nineteen, we observed many discrepancies caused by the Linux NFS server's use of some undefined bits in the MODE field (i.e., the field with the access control bits for owner, group, and world) of every file object's attributes. The Linux server encodes the object's type (e.g., directory, symlink, or regular file) in these bits, which causes the MODE field to not match FreeBSD's values in every response. To eliminate this recurring discrepancy, we modified the comparison rules to replace bitwise-comparison

Field	Count	Reason
EOF flag	1	FreeBSD server failed to return EOF at the end of a read reply
Attributes follow flag	10	Linux sometimes chooses not to return pre-op or post-op attributes
Time	6	Parent directory pre-op **`ctime`** and **`mtime`** are set to the current time on FreeBSD
Time	2	FreeBSD does not update a symbolic link's **`atime`** on READLINK

Table 1: **Discrepancies when comparing Linux and FreeBSD servers.** The fields that differ are shown along with the number of distinct RPCs for which they occur and the reason for the discrepancy.

of the entire MODE field with a loose-compare function that examines only the specification-defined bits.

Perhaps the most interesting discrepancy is the EOF flag, which is the flag that signifies that a read operation has reached the end of the file. Our Tee tells us that when a FreeBSD client is reading data from a FreeBSD server, the server returns FALSE at the end of the file while the Linux server correctly returns TRUE. The same discrepancy is observed, of course, when the FreeBSD and Linux servers switch roles as reference server and SUT. The FreeBSD client does not malfunction, which means that the FreeBSD client is not using the EOF value that the server returns. Interestingly, when running the same experiment with a Linux client, the discrepancy is not seen because the Linux client uses different request sequences. If a developer were trying to implement a FreeBSD NFS server clone, the NFS Tee would be an useful tool in identifying and properly mimicking this quirk.

The "attributes follow" flag, which indicates whether or not the attribute structure in the given response contains data,[6] also produced discrepancies. These discrepancies mostly come from pre-operation directory attributes in which Linux, unlike FreeBSD, chooses not to return any data. Of course, the presence of these attributes represents additional discrepancies between the two servers' responses, but the root cause is the same decision about whether to include the optional information.

The last set of interesting discrepancies comes from timestamps. First, we observe that FreeBSD returns incorrect pre-operation directory modification times (`mtime` and `ctime`) for the parent directory for RPCs that create a file, a hard link, or a symbolic link. Rather than the proper values being returned, FreeBSD returns the current time. Second, FreeBSD and Linux use different policies for updating the last access timestamp (`atime`). Linux updates the `atime` on the symlink file when the symlink is followed, whereas FreeBSD only updates the `atime` when the symlink file is accessed directly (e.g., by writing it's value). This difference exhibits discrepancies in RPCs that read the symlink's attributes.

We also ran the test with the servers swapped (FreeBSD as reference and Linux as SUT). Since the client interacts with the reference server's implementation, we were interested to see if the FreeBSD client's interaction with a FreeBSD NFS server would produce different results when compared to the Linux server, perhaps due to optimizations between the like client and server. But, the same set of discrepancies were found.

Comparing Linux 2.6 to Linux 2.4: Comparing Linux 2.4 to Linux 2.6 resulted in very few discrepancies. The Tee shows that the 2.6 Kernel returns file metadata timestamps with nanosecond resolution as a result of its updated VFS layer, while the 2.4 kernel always returns timestamps with full second resolution. The only other difference we found was that the parent directory's pre-operation attributes for SETATTR are not returned in the 2.4 kernel but are in the 2.6 kernel.

Comparing Network Appliance FAS900 to Linux and FreeBSD: Comparing the Network Appliance FAS900 to the Linux and FreeBSD servers yields a few interesting differences. The primary observation we are able to make is that the FAS900 replies are more similar to FreeBSD's that Linux's. The FAS900 handles its file MODE bits like FreeBSD without Linux's extra file type bits. The FAS900, like the FreeBSD server, also returns all of the pre-operation directory attributes that Linux does not. It is also interesting to observe that the FAS900 clearly handles directories differently from both Linux and FreeBSD. The cookie that the Linux or FreeBSD server returns in response to a `READDIR` or `READDIRPLUS` call is a byte offset into the directory file whereas the Network Appliance filer simply returns an entry number in the directory.

Aside: It is interesting to note that, as an unintended consequence of our initial relay implementation, we discovered an implementation difference between the FAS900 and the Linux or FreeBSD servers. The relay modifies the NFS call's XIDs so that if two clients happen to use the same XID, they don't get mixed up when the Tee relays them both. The relay is using a sequence of values

[6] Many NFSv3 RPCs allow the affected object's attributes to be included in the response, at the server's discretion, for the client's convenience.

for XIDs that is identical each time the relay is run. We found that, after restarting the Tee, requests would often get lost on the FAS900 but not on the Linux or FreeBSD servers. It turns out that the FAS900 caches XIDs for much longer than the other servers, resulting in dropped RPCs (as seeming duplicates) when the XID numbering starts over too soon.

Debugging the Ursa Major NFS server: Although the NFS Tee is new, we have started to use it for debugging an NFS server being developed in our group. This server is being built as a front-end to Ursa Major, a storage system that will be deployed at Carnegie Mellon as part of the Self-* Storage project [4]. Using Linux as a reference, we have found some non-problematic discrepancies (e.g., different choices made about which optional values to return) and one significant bug. The bug occurred in responses to the READ command, which never set the EOF flag even when the last byte of the file was returned. For the Linux clients used in testing, this is not a problem. For others, however, it is. Using the Tee exposed and isolated this latent problem, allowing it to be fixed proactively.

5.3 Performance impact of prototype

We use PostMark to measure the impact the Tee would have on a client in a live environment. We compare two setups: one with the client talking directly to a Linux server and one with the client talking to a Tee that uses the same Linux server as the reference. We expect a significant increase in latency for each RPC, but less significant impact on throughput.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [6]. It creates a large number of small randomly-sized files (between 512 B and 9.77 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append.

The experiments were done with a single client and up to sixteen concurrent clients. Except for the case of a single client, two instances of PostMark were run on each physical client machine. Each instance of PostMark ran with 10,000 transactions on 500 files and the biases for transaction types were equal. Except for the increase in the number of transactions, these are default PostMark values.

Figure 5 shows that using the Tee reduces client throughput when compared to a direct NFS mount. The reduction is caused mainly by increased latency due to the added network hop and overheads introduced by the fact that the Tee is a user-level process.

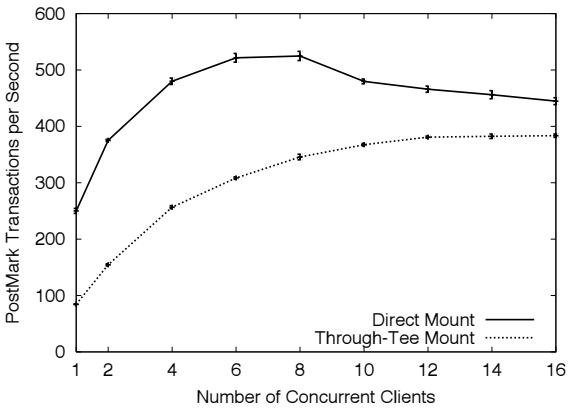

Figure 5: **Performance with and without the Tee.** The performance penalty caused by the Tee decreases as concurrency increases, because higher latency is the primary cost of inserting a Tee between client and reference serer. Concurrency allows request propagation and processing to be overlapped, which continues to benefit the Through-Tee case after the Direct case saturates.. The graph shows average and standard deviation of PostMark throughput, as a function of the number of concurent instances.

The single-threaded nature of PostMark allows us to evaluate both the latency and the throughput costs of our Tee. With one client, PostMark induces one RPC request at a time, and the Tee decreases throughput by 61%. As multiple concurrent PostMark clients are added, the percentage difference between direct NFS and through-Tee NFS performance shrinks. This indicates that the latency increase is a more significant factor than the throughput limitation—with high concurrency and before the server is saturated, the decrease in throughput drops to 41%. When the server is heavily loaded in the case of a direct NFS mount, the Tee continues to scale and with 16 clients the reduction in throughput is only 12%.

Although client performance is reduced through the use of the Tee, the reduction does not prevent us from using it to test synchronization convergence rates, do offline case studies, or test in live environments where lower performance is acceptible.

5.4 Speed of synchronization convergence

One of our Tee design goals was to support dynamic addition of a SUT in a live environment. To make such addition most effective, the Tee should start performing comparisons as quickly as possible. Recall that operations on a file object may be compared only if the object is synchronized. This section evaluates the effectiveness of the synchronization ordering enhancements described in Section 4.2. We expect them to significantly increase the speed with which useful comparisons can begin.

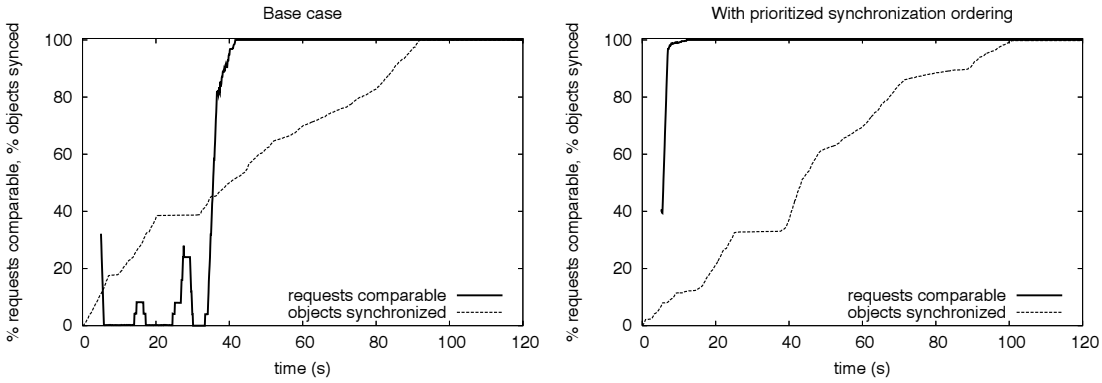

Figure 6: **Effect of prioritized synchronization ordering on speed of convergence.** The graph on the left illustrates the base case, with no synchronization ordering enhancements. The graph on the right illustrates the benefit of prioritized synchronization ordering. Although the overall speed with which the entire file system is synchronized does not increase (in fact, it goes down a bit due to contention on the SUT), the percentage of comparable responses quickly grows to a large value.

To evaluate synchronization, we ran an OpenSSH compile (the compile phase of the *ssh-build* benchmark used by Seltzer, et al. [12]) on a client that had mounted the reference server through the Tee. The compilation process was started immediately after starting the plugin. Both reference server and SUT had the same hardware configuration and ran the same version of Linux. No other workloads were active during the experiment. The OpenSSH source code shared a mount point with approximately 25,000 other files spread across many directories. The sum of the file sizes was 568MB.

To facilitate our synchronization evaluation, we instrumented the Tee to periodically write internal counters to a file. This mechanism provides us with two point-in-time values: the number of objects that are in a synchronized state and the total number of objects we have discovered thus far. It also provides us with two periodic values (counts within a particular interval): the number of requests enqueued for duplication to the SUT and the number of requests received by the plugin from the relay. These values allow us to compute two useful quantities. The first is the ratio of requests enqueued for duplication to requests received, expressed as a moving average; this ratio serves as a measure of the proportion of operations that were comparable in each time period. The second is the ratio of synchronized objects to the total number of objects in the file system; this value measures how far the synchronization process has progressed through the file system as a whole.

Figure 6 shows how both ratios grow over time for two Tee instances: one (on the left) without the synchronization ordering enhancements and one with them. Although synchronization of the entire file system requires over 90 seconds, prioritized synchronization ordering quickly enables a high rate of comparable responses. Ten seconds into the experiment, almost all requests produced comparable responses with the enhancements. Without the enhancements, we observe that a high rate of comparable responses is reached at about 40 seconds after the plugin was started. The rapid increase observed in the unoptimized case at that time can be attributed to the synchronization module reaching the OpenSSH source code directory during its traversal of the directory tree.

The other noteworthy difference between the unordered case and the ordered case is the time required to synchronize the entire file system. Without prioritized synchronization ordering, it took approximately 90 seconds. With it, this figure was more than 100 seconds. This difference occurs because the prioritized ordering allows more requests to be compared sooner (and thus duplicated to the SUT), creating contention for SUT resources between synchronization-related requests and client requests. The variation in the rate with which objects are synchronized is caused by a combination of variation in object size and variation in client workload (which contends with synchronization for the reference server).

6 Discussion

This section discusses several additional topics related to when comparison-based server verification is useful.

Debugging FS client code: Although its primary *raison d'etre* is file server testing, comparison-based FS verification can also be used for diagnosing problems with client implementations. Based on prior experiences, we believe the best example of this is when a client is observed to work with some server implementations and not others (e.g., a new version of a file server). Detailed insight can be obtained by comparing server responses to

request sequences with which there is trouble, allowing one to zero in on what unexpected server behavior the client needs to cope with.

Holes created by non-comparable responses: Comparison-based testing is not enough. Although it exposes and clarifies some differences, it is not able to effectively compare responses in certain situations, as described in Section 4. Most notably, concurrent writes to the same data block are one such situation—the Tee cannot be sure which write was last and, therefore, cannot easily compare responses to subsequent reads of that block. Note, however, that most concurrency situations can be tested.

More stateful protocols: Our file server Tee works for NFS version 3, which is a stateless protocol. The fact that no server state about clients is involved simplifies Tee construction and allows quick ramp up of the percentage of comparable operations. Although we have not built one, we believe that few aspects would change significantly in a file server Tee for more stateful protocols, such as CIFS, NFS version 4, and AFS [5]. The most notable change will be that the Tee must create duplicate state on the SUT and include callbacks in the set of "responses" compared—callbacks are, after all, external actions taken by servers usually in response to client requests. A consequence of the need to track and duplicate state is that comparisons cannot begin until both synchronization completes **and** the plug-in portion of the Tee observes the beginnings of client sessions with the server. This will reduce the speed at which the percentage of comparable operations grows.

7 Related work

On-line comparison has a long history in computer fault-tolerance [14]. Usually, it is used as a voting mechanism for determining the right result in the face of problems with a subset of instances. For example, the triple modular redundancy concept consists of running multiple instances of a component in parallel and comparing their results; this approach has been used, mainly, in very critical domains where the dominant fault type is hardware problems. Fault-tolerant consistency protocols (e.g., Paxos [11]) for distributed systems use similar voting approaches.

With software, deterministic programs will produce the same answers given the same inputs, so one accrues little benefit from voting among multiple instances of the same implementation. With multiple implementations of the same service, on the other hand, benefits can accrue. This is generally referred to as N-version programming [2]. Although some argue that N-version programming does not assist fault-tolerance much [8, 9], we view comparison-based verification as a useful application of the basic concept of comparing one implementation's results to those produced by an independent implementation.

One similar use of inter-implementation comparison is found in the Ballista-based study of POSIX OS robustness [10]. Ballista [3] is a tool that exercises POSIX interfaces with various erroneous arguments and evaluates how an OS implementation copes. In many cases, DeVale, et al. found that inconsistent return codes were used by different implementations, which clearly creates portability challenges for robustness-sensitive applications.

Use of a server Tee applies the proxy concept [13] to allow transparent comparison of a developmental server to a reference server. Many others have applied the proxy concept for other means. In the file system domain, specifically, some examples include Slice [1], Zforce [17], Cuckoo [7], and Anypoint [16]. These all interpose on client-server NFS activity to provide clustering benefits to unmodified clients, such as replication and load balancing. Most of them demonstrate that such interposing can be done with minimal performance impact, supporting our belief that the slowdown of our Tee's relaying could be eliminated with engineering effort.

8 Summary

Comparison-based server verification can be a useful addition to the server testing toolbox. By comparing a SUT to a reference server, one can isolate RPC interactions that the SUT services differently. If the reference server is considered correct, these discrepancies are potential bugs needing exploration. Our prototype NFSv3 Tee demonstrates the feasibility of comparison-based server verification, and our use of it to debug a prototype server and to discover interesting discrepancies among production NFS servers illustrates its usefulness.

Acknowledgements

We thank Raja Sambasivan and Mike Abd-El-Malek for help with experiments. We thank the reviewers, including Vivek Pai (our shepherd), for constructive feedback that improved the presentation. We thank the members and companies of the PDL Consortium (including EMC, Engenio, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foun-

dation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

References

[1] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 22–25 October 2000), 2000.

[2] L. Chen and A. Avizienis. N-version programming: a fault tolerance approach to reliability of software operation. *International Symposium on Fault-Tolerant Compter Systems*, pages 3–9, 1978.

[3] J. P. DeVale, P. J. Koopman, and D. J. Guttendorf. The Ballista software robustness testing service. *Testing Computer Software Conference* (Bethesda, MD, 14–18 June 1999). Unknown publisher, 1999.

[4] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: Brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[5] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[6] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[7] A. J. Klosterman and G. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[8] J. C. Knight and N. G. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, **15**(1):24–35. ACM, January 1990.

[9] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumptions of independence in multiversion programming. *Trnsactions on Software Engineering*, **12**(1):96–109, March 1986.

[10] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. *International Symposium on Fault-Tolerant Compter Systems* (Madison, WI, 15–18 June 1999), 1999.

[11] L. Lamport. Paxos made simple. *ACM SIGACT News*, **32**(4):18–25. ACM, December 2001.

[12] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 71–84, 2000.

[13] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. *International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[14] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems: design and evaluation*. Digital Press, Second edition, 1992.

[15] SPEC SFS97_R1 V3.0 benchmark, Standard Performance Evaluation Corporation, August, 2004. http://www.specbench.org/sfs97r1/.

[16] K. G. Yocum, D. C. Anderson, J. S. Chase, and A. M. Vahdat. Anypoint: extensible transport switching on the edge. *USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003), 2003.

[17] Z-force, Inc., 2004. www.zforce.com.