Comparison-based Filesystem Verification (The NFS Tee)

Yuen-Lin Tan
Advised by: Greg Ganger, John Strunk

Abstract

This work explores a particular approach to testing a developmental filesystem – comparing its behavior with that of a reference implementation in a live environment, that is, one consisting of real users, precious data and non-synthetic workloads. We believe such a configuration has the potential to offer unique insight into the system's behavior as well as aid the debugging effort. The major areas considered are live synchronization of NFS servers and comparison-based analysis of an NFSv3 request-response stream. The work will include the construction and subsequent experimental usage of a tool, dubbed the NFS Tee, which will serve as a key testing mechanism in the Parallel Data Lab's Self-* Storage project.

1. Background

The Parallel Data Lab's Self-* (pronounced "self star") Storage project [1] seeks to drastically reduce the amount of human intervention required to administer a large-scale storage system, employing a variety of techniques to increase the system's autonomy while granting the human administrator control over high-level service goals. Self-* is a complex system with a rich assortment of interacting components.

At the heart of Self-* lies a distributed object store. Users do not interact directly with this component, instead communicating with various "head-ends" which translate between existing network storage protocols (NFS, AFS, etc.) and Self-*'s internal protocol. One reason behind this design decision was that supporting well-established protocols would ease deployment of the prototype and thus allow the system to be tested under workloads produced by real users. These workloads are likely to be richer and more varied than those obtainable via traces and simulation alone.

Certain issues arise. Users are unlikely to entrust their daily work and valuable data to an unproven developmental system, even if it exports familiar interfaces. Given that a way is found to mitigate that shortfall of confidence, the newly obtained workloads must be harnessed to aid debugging and gain insight into the developmental system's behavior. Our system addresses these issues in an NFS context. From here on, we refer to it as the tee[1].

2. Introduction

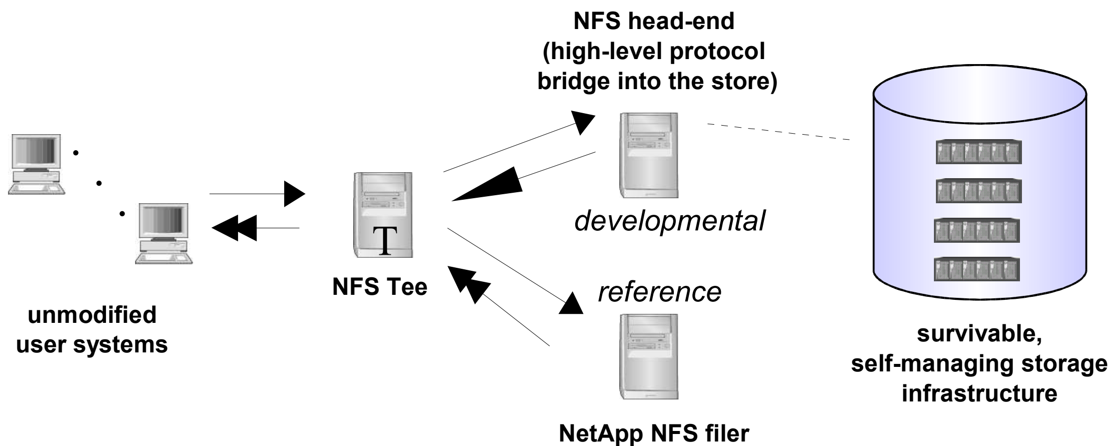
The tee provides users with industry-standard levels of reliability and availability by doing exactly that – storing their data on a commercial filer, and fulfilling their requests from that machine. It forwards a duplicate of the request stream to the Self-* NFS head-end and compares the results produced with those produced by the commercial filer.

As a pre-requisite to performing those comparisons, the tee replicates filesystem objects from the commercial filer to Self-* and maintains synchronization between the two. Doing this efficiently, in a live environment and without resorting to locking proved to be a significant challenge.The

---

1 In light of similarities between it and the UNIX `tee` utility.

tee's location in a deployed Self-* environment is depicted in the following diagram. A single RPC transaction is taking place. Each arrow style depicts a unique RPC packet. The NFS filer is the “reference” server and the Self-* NFS head-end is the “developmental” server.



The tee performs three functions:

1. Relays each user request to both servers, reference and developmental. Relays the *reference server*'s response back to user.
2. For each request, compares the developmental server's response with that of the reference server. The comparison logic used is discussed later.
3. Replicates the contents of the reference server onto the developmental server and maintains synchronization between the two copies of each object.

It is subject to the following primary design requirements:

1. It should be transparent. Its behavior as observed by a client should be indistinguishable from that of the reference server. The tee may perturb the request and response stream at most to the extent of the underlying network.
2. Functionality non-essential to client sessions (function 2 and 3, above) should be hot-pluggable, meaning able to start running without causing any downtime.
3. It should have minimal impact on client-perceived throughput and response time.
4. It should be robust. File service to users should continue uninterrupted despite faults in components not essential to providing that service.

10TB of data and $100*10^6$ filesystem objects [2] were used as ballpark figures to guide the initial design.

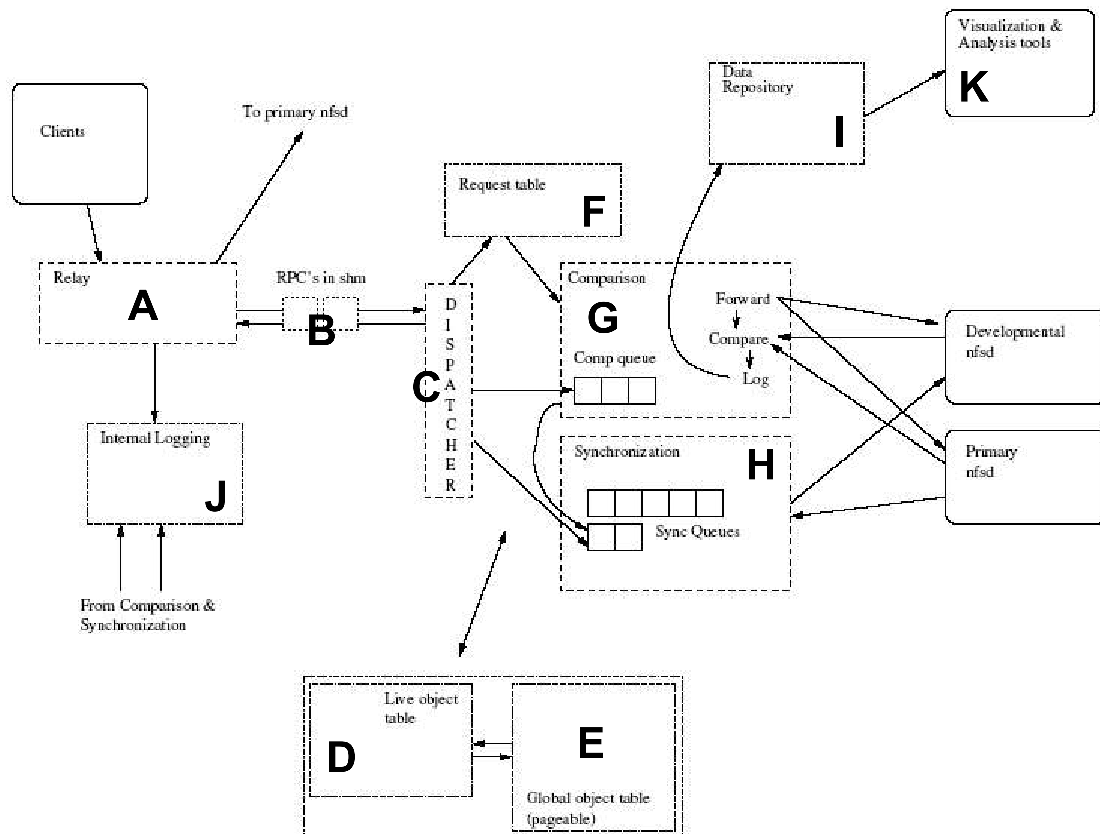
To perform its functions, the tee must be fully interposed between user systems and the reference server such that it sees every request that reaches the reference server and every response that gets sent back. In the absence of that assumption, the tee would not be able to conclusively determine the state of each filesystem object at all times. This would jeopardize its ability to perform function 3 and consequently, function 2. For the same reason, filesystem objects must not be modifiable via other channels.

The rest of this paper is laid out as follows. Section 3 describes the tee's architecture at a high level and touches on some of the support facilities which were implemented. Section 4 discusses the approach to tackling the problem of synchronization, presents the resulting algorithms and describes their implementation. Section 5 does the same for comparison. Section 6 describes the

testing and evaluation methodology and [presents the results obtained thus far]. Section 7 mentions some possible future work, and Section 8 concludes.

3. Architecture

The tee's high-level architecture is depicted in the following diagram. The implementation was done in C++. The State Threads library was used as it is particularly well-suited to network I/O-bound applications, and provides a non-preemption property that greatly simplifies implementation. Various components of note are described below.



Relay (A). The tee's user-visible component is an RPC relay[2] which multiplexes user requests onto a single connection with the reference server and relays its responses back. Each RPC packet flowing from the clients to the reference server, and back, is made available on a read-only basis to the plugin via shared memory buffers (B).

NFS plugin (C – J). The NFS plugin implements the tee's comparison and synchronization functionality. In the interest of fault isolation, it runs as a separate process from the relay. Communication with the relay is performed over a local socket, and consists of small messages containing (shm key, offset) pairs.

Live object table (D). This hash table, keyed on reference filehandle, stores the combined working set of the comparison and synchronization modules. Each entry contains the corresponding developmental filehandle and various other object-specific fields. As data in this table is expected to be heavily accessed, it is kept in core at all times. It can be thought of as level 1 of a 2-level, internally managed cache of object-related data.

2 Based on code by John Strunk.

Global object table (E). This table, intended to be implemented as a b-tree, is a superset of the live object table, representing all objects on all exported filesystem trees in a space-efficient manner. The table's core utilization will be managed by manually “paging” portions of it between core and disk. Essentially, the size of the live object table and the “paged in” portion of the global object table combined is kept less than the amount of physical memory in the system.

Dispatcher (C). The dispatcher receives RPC buffer location information from the relay for each RPC that flows from client to reference server and back. It updates internal data structures (D, E, F) and enqueues objects that describe work onto the comparison and synchronization modules' work queues.

Synchronization & comparison modules (H & G). Described in sections 4 and 5, respectively.

The following support facilities were implemented but not depicted in the architectural diagram.

NFSv3 & RPC client library, Server classes. Functions representing NFSv3 calls and a simple RPC client library were implemented. These are transport-independent, performing network I/O via a transport-specific server object. Each server object exports a raw I/O interface in addition to an RPC call interface.

Thread groups. A thread group is an abstraction encapsulating the following components: a work queue, shared data, and a set of threads each with their own server object. Work is assigned to the thread group, and the threads spend their lifetimes processing work on the queue. Threads can be started and stopped at any time by altering the number of active threads in the thread group.

4. Synchronization

What does it mean for an NFS object to be synchronized? Let O_r and O_d be two objects. Let A be the set of arguments to an NFS request, and let R be the set of return values. Let $f: O \times A \rightarrow R$ be an NFS request, i.e. a function that given an object and an argument as input, returns a value. Then, O_r and O_d are synchronized if, for any f, $f(O_r, A_i) = f(O_d, A_i)$. In other words, two objects are synchronized if an NFS request performed on each of them with the same arguments can be expected to return the same value[3]. This expectation lays the foundation for comparison-based verification of the two programs which claim to implement the protocol.

The definition above becomes useful once we know, besides the arguments in a request, what object state affects the value computed by it, as it is this state that we must replicate. This is less implementation-specific and more constrained than one might think, as NFS requests and the objects they operate on are very similar to those found in conventional filesystems. Thus, using that knowledge and the NFS protocol definition, it is possible to determine with some certainty what constitutes an object's state.

The following table describes what we consider state for each of the NFS object types.

Directory	Regular file	Symlink	Special file
1. Attributes 2. Directory entries (name-type pairs)	1. Attributes 2. File data (bytes)	1. Attributes 2. Stored pointer	1. Attributes

Table 1 Definition of "state" for NFS object types

Besides returning a result, an NFS request may transform an object to a new state. If it does, we

3 The concept of equality referred to here is not strict equality – Section 5 elaborates.

call it a “write”. An object's lifetime consists of a sequence of states with transitions between adjacent states brought about by writes. This is depicted in Figure 1, which shows a period in the lifetime of two objects.

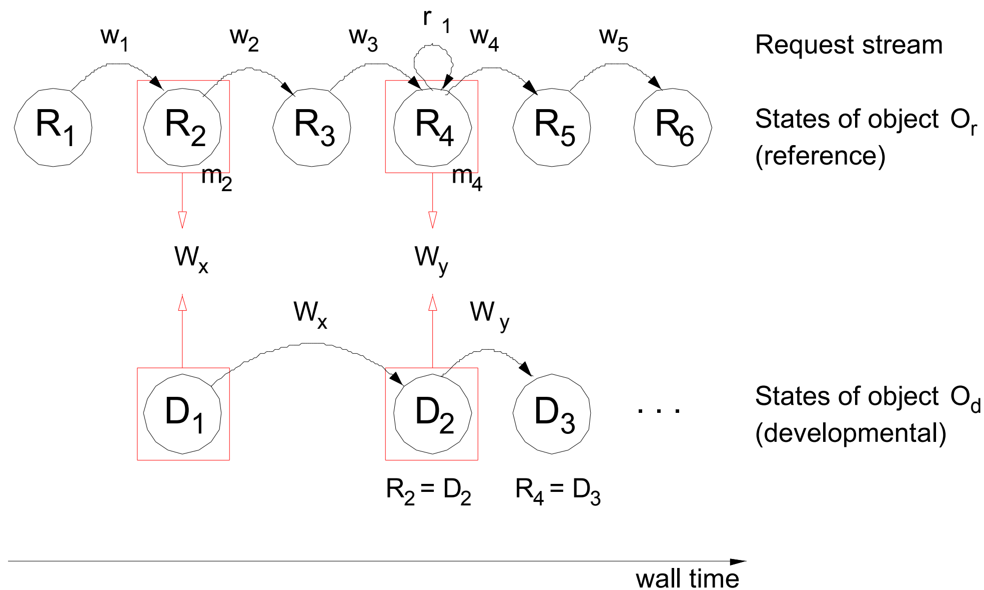


Figure 1 Basic synchronization of 2 objects

In Figure 1, w_i, ..., r_i, w_j is the request stream passing through the relay from users to the reference server. R_1, ..., R_n represent consistent states (states between 2 writes). For any object, it is crucial that we replicate a consistent state – otherwise, the results of future comparisons are indeterminate (recall that to compare $f(O_r)$ and $f(O_d)$, must have $O_r = O_d$ and that on the reference server, requests always take consistent states as input).

As will be seen, *replicating* a consistent state does not imply we must *read* a consistent state. With regular files, we can get away without doing the latter (exactly how is explained later). With directories, however, we don't have this luxury. The reason is that the NFSv3 calls for reading a directory behave unpredictably across directory operations [3]. Thus, upon detecting a change we are forced to rescan the directory from the beginning.

After a consistent state has been replicated, the object's state on the developmental server (D_2 in the figure) corresponds to the object's state on the reference server some time in the past (R_3 in the figure). Notice that in the time it took to replicate R_3, 2 writes and a read have reached the reference server. With respect to the request sequence w_3, ..., w_5, O_r and O_d are synchronized. However, with respect to the request sequences r_1, ..., w_5 and w_4, ..., w_5, O_r and O_d are not synchronized ($w_3(O_r)$ and $w_3(O_d)$ are). The question then is how to deal with writes that occur during the synchronization procedure. There are 3 possible ways:

1. *Resynchronize from scratch*. Basic synchronization takes this approach, as shown in Figure 1. If O_r changed during replication (detected by noticing m_4 and m_2 – the object's mtimes – differ), synchronization is restarted.
2. *Store the writes in their original format.* In this approach, the original write RPCs are buffered and played back after initial replication completes. This approach was not chosen due to space concerns, and because the 3rd approach had the potential to be more efficient.
3. *Store the writes in an internal format.* In this approach, the writes are parsed and

converted into an internal representation (called a "changeset") that is then committed to the developmental server after initial replication completes. This approach was used both for directory and regular file synchronization, and is elaborated on in their respective sections.

Ways 2 and 3 prevent having to resynchronize from scratch when an object changes during initial replication – useful for busy objects.

Regardless, once initial replication completes and intervening writes have been handled, the object is synchronized. Comparison is "turned on" for the object and all subsequent requests that reference it are forwarded to the developmental server for comparison. This continues until the object loses synchronization, which occurs when it undergoes write sharing. Write sharing is elaborated on in section 4.3.

4.1. Directory synchronization

Figure 2 Synchronization of a directory

The algorithm for directory synchronization is as follows. `O` is the tee data structure that represents the object pair `Or`, `Od`.

```
1 sync(Directory Or, Directory Od, bool initial):
2   if (initial):  // initial replication
3     d_dirlist = scandir(Od)
4
5     i_mtime = get_mtime(Or)
6     O.clear_changeset()
7     r_dirlist = scandir(Or)
8     f_mtime = get_mtime(Or)
9
10    if (i_mtime != f_mtime):
11      return RETRY_LATER
12
13    enqueue(intersection(d_dirlist, r_dirlist)) // for traversal
14    Od.remove(d_dirlist - r_dirlist)
15    Od.create_n_enqueue(r_dirlist - d_dirlist)  // after creating,
16                                                 // enqueue for sync
(continued)
```

```
17  // commit changeset (intervening writes)
18  while (!O.changeset_empty()):
19    Od.remove(O.rem_set)
20    Od.create_n_enqueue(O.add_set)
21
22  return SYNCED
```

In lines 13 & 14 the difference between the 2 directories is computed and initial replication performed ($D_2 = W_x(D_1)$ in Figure 2). In lines 17–19, the changeset entries (intervening writes) are committed to the developmental object. In Figure 2, the changeset is represented by W_{cs} and is formed from w_3, w_4 and w_5. A directory object's changeset consists of two sets of (name, type, filehandle) tuples, one containing entries to be created and the other, entries to be removed.

Special handlers for the various directory operations (CREATE, REMOVE, etc.) are executed by the dispatcher upon receipt of a successful response to those requests. The handlers update the changeset for the referenced directory by adding or removing an entry. If an entry that is being added or removed is currently in the opposing set, it is simply removed from the set and network I/O is conserved.

Directory synchronization replicates (name, type) entries within each directory and enqueues all the entries it encounters for synchronization (in effect, traversing the filesystem tree). The task of synchronizing an object's data is left to its object-specific synchronization handler. Thus, a directory can be deemed synchronized despite the fact that some or all of the objects pointed to from it are unsynchronized.

To handle hard links, before an object is created, it is looked up in the object table using its reference filehandle. If the object exists and has a reference-developmental filehandle mapping, a hard link is created instead of a new object.

4.2. Regular file synchronization

The regular file synchronization algorithm is as follows.

```
1 sync(RegFile Or, RegFile Od, bool initial):
2   if (initial):  // initial replication
3     i_mtime = get_mtime(Or)
4     O.clear_changeset()
5     (buf, f_mtime) = read(Or, 0)  // read 1st unit
6
7     if (i_mtime != f_mtime):
8       return RETRY_LATER
9
10    truncate(Od)
11    write(Od, 0, buf)
12
13    while (Or.eof()):  // replicate rest of file
14      (buf, f_mtime) = read(Or, ++curr)
15      write(Od, curr, buf)
16
17  while (!O.changeset_empty()):  // commit changeset
18    (start, end, len) = O.changeset_pop()
19    if (Or.cached(start, end, len)):
20      write(Od, start, len, cached_buf)
21    else:
22      (buf, _) = read(Or, start, len)
23      write(Od, start, len, buf)
24
25  if (f_mtime == i_mtime): return SYNCED
26  else if (O.changeset_last_mtime >= f_mtime): return SYNCED
27  else: return RETRY_LATER
```

A few points worth mentioning:

- The changeset for a regular file is a list of extents; each item within the list stores the tuple (start_offset, end_offset, cached_data). The list records the regions of the file that were modified by intervening writes. When a successful write is observed, and its post-operation mtime corresponds to a state after the initial read, the extent that was written to is inserted into the list, coalescing if necessary (adjacent extents are combined & after an insert, no extents overlap). As an extra optimization, a certain amount of data written is cached to avoid additional network I/O and server load for small writes. The amount of data cached per regular file is kept below a certain value.

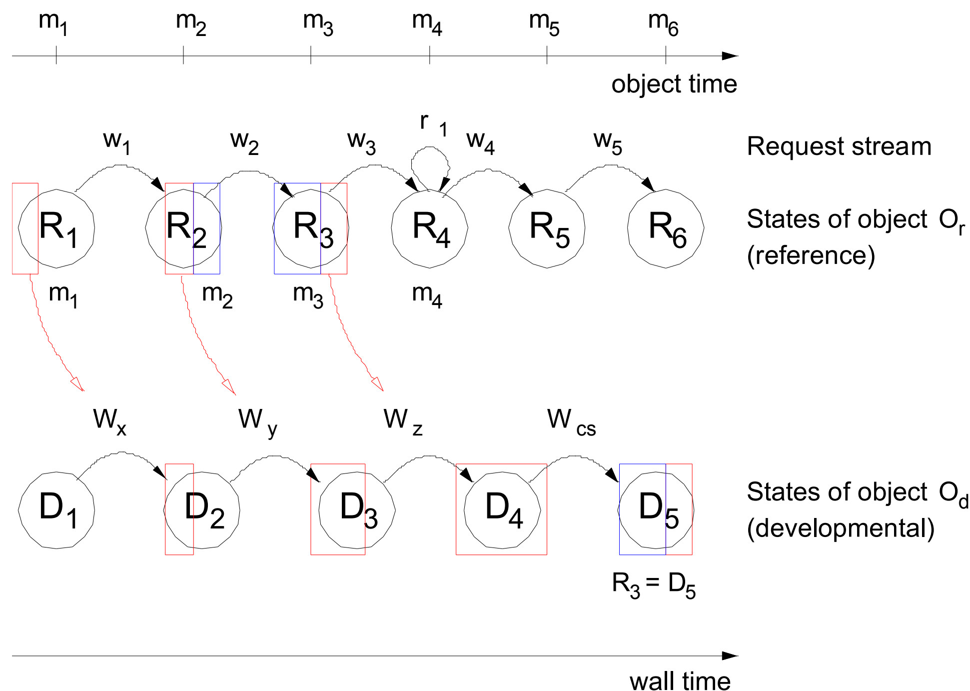


Figure 3 Synchronization of a regular file

- For regular files, it is not necessary to read the entire file in a consistent state. Instead, it is sufficient to read the first arbitrarily sized unit consistently (lines 3–8 in the pseudocode). The reason we must do this has to do with updating the object's changeset upon observing a write.

  In Figure 3, our first read took place at R_1. Clearly, all writes that occurred subsequent to R_1 should be recorded in the changeset. In the figure, it is easy to see that this corresponds to all writes with a post-operation mtime[4] greater than m1. Consider, however, what would happen if O_r changed while we were performing this initial read (the post-operation mtime observed on line 5 differed from the mtime observed on line 3). This means that between those points, any number of writes took place. When do we begin recording changes in the changeset, i.e. what is the minimum post-operation mtime for which we update the extent list with the written-to extent? If we begin at `i_mtime`, a write occurred at `i_mtime+1` and our initial read read the state at `i_mtime+2`, we may erroneously overwrite upon committing the changeset. If we begin at `f_mtime` as observed on line 5, our initial read read the state at `f_mtime-2` and a write occurred at `f_mtime-1`, we may miss the contents of the write.

4 The NFSv3 WRITE call returns the mtime of the file after the operation completed.

Thus, by beginning changeset updates between the initial mtime sample and the initial read, and by ensuring that no changes took place between the initial mtime and the post-operation mtime of the read, we are guaranteed to observe exactly the writes we want. Viewed in another light, we have successfully "named" the consistent state at which we began synchronization. Each consistent state can be labelled by the post-op mtime of the write that caused the transition to it.

- After our initial replication (by line 17) we arrived at state D_4 which is consistent with R_1 but not with subsequent states. See Figure 3. In the O_r sequence, the red rectangle represents the region of the file that we are replicating at that point in time. In the O_d sequence, the red rectangle represents the growing region that has been replicated. The blue rectangles represent the region of the file that was overwritten by the most recent write. Note that state D_4 is inconsistent with R_2 and R_3, but by committing the changeset (replicating observed changes), state D_5 becomes consistent with R_3.

  This changeset update policy is likely to save us significant amounts of time when synchronizing large files that undergo small changes.

- We do not bother reading the initial contents of O_d, comparing that with what we read from O_r, and committing the difference. Firstly, upon synchronizing O for the first time we must do a full scan of O_r in any case. Doing a full scan of O_d and then computing and committing the difference between O_r and O_d would impose additional network and computational load over simply overwriting O_d, at no apparent benefit.

4.3. Write sharing

Write sharing is the condition where there are 2 or more outstanding requests on an object at a server such that the order in which the requests were actually executed cannot be determined by an external observer. Because we cannot determine the state of the object after an occurrence of write sharing by observing the request stream, we consider that the object has become unsynchronized and enqueue it for resynchronization.

The write sharing detection algorithm is as follows. Assume w_2 arrived after w_1. In the diagram, w.req and w.ack are the times at which the request's call and reply are observed by the tee, respectively.

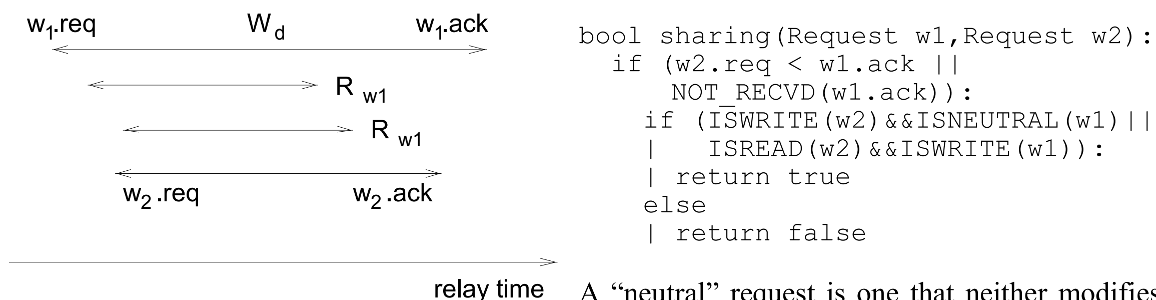


Figure 4 Example: write sharing detection

A "neutral" request is one that neither modifies nor accesses an object's state. The algorithm checks if the execution windows of the requests overlap, as observed by the tee. A request's observed execution window is guaranteed to fully overlap its actual execution window – the period it spends at the server, and thus the period in which it possibly contends with another request – for the following 2 reasons:

1. the request cannot have begun executing on the server prior to w.req (server would not have received the call)

2. the request cannot have finished executing on the server after w.ack (the reply cannot have been sent after execution completed)

Thus, the algorithm may produce false positives, but never false negatives.

4.4. Synchronization module

The synchronization module is responsible for replicating the various filesystem trees exported by the reference server onto the developmental server, and for keeping them synchronized. It takes the form of a thread group with a priority queue containing pointers to the object table entries of objects queued for synchronization. These include objects newly encountered since the last invocation of the tee, and objects for which an occurrence of write-sharing was detected.

An object's synchronization priority is equal to the sum of the weighted moving average of its access frequency and the synchronization priorities of objects descended from it in the filesystem tree. Backpointers that depict child-parent relationships are opportunistically established within object table entries. When an object is accessed, the weighted moving average of its access frequency is updated, and the path from it to the root is traversed as far as possible, updating the synchronization priority of each object along the way. Doing this optimizes the following case that is likely to arise frequently in the early stages of synchronization – if an object's name does not exist, it cannot be created until all directories from its parent to the filesystem root have been created.

The hope is that by synchronizing entries in order of their access frequency, the tee can quickly reach a state where the majority of responses can be compared, since access frequencies within each exported tree are likely to vary significantly.

The overall synchronization procedure is bootstrapped by the dispatcher by enqueueing, for each exported tree, an entry corresponding to its root directory.

5. Comparison

As touched on in Section 4, given that two objects O_r and O_d are synchronized, we can compare the results of executing $f(O_r, A)$ and $f(O_d, A)$ on the reference and developmental servers, respectively. f is an NFS request and A is an argument list. Although server implementors are given significant freedom in building their implementations, the NFS protocol is, in most cases reasonably specific about the semantics of each request. Thus, if 2 servers claim to implement the protocol and one is accepted to be correctly implemented and robust, a discrepancy in the behavior of the other is at least worth a close look. The primary advantage of the tee with respect to such verification is its ability to perform comparison under a live, continuously varying workload – one which may give rise to behavior that would be unproduceable via a synthetic test.

An NFS request takes arguments and returns results in a format similar to a C `struct`. Fundamentally, for each field comprising a response, we are concerned with the question of whether it is comparable, and if so, by what method. A comparison rule expresses the derivation of an answer from the information available to the comparison module for each request: the request itself, the pair of responses, data on write sharing (if any). For instance, a rule may state that if the request is a "lookup", the resulting filehandles are uncomparable.

A set of comparison rules have been formulated by studying the protocol. The set will be refined based on the experienced gained from experimenting with the tee. In the common case, comparison amounts to testing the byte equality of a field. Some fields, however, require special handling (eg: time-related fields must be compared within a certain delta) – in such cases, a field-specific handler is invoked during comparison. A few example rules are given below in pseudocode (assume the default is that every field is comparable by simple equality).

```
1 // concept of "dirty set" described in Section 5.1
2 if (write_sharing) {dirty_set}.comp = !BITWISE | SPEC_HANDLER
3 ... other general rules ...
4
5 fattr3.{nlink,used,fsid,fileid}.comp = NO_COMP
6 fattr3.{atime,mtime,ctime}.comp = !BITWISE | SPEC_HANDLER
7 ... other field-specific rules ...
8
9 if (reqtype == SETATTR && obj_ctime) obj_wcc.comp = NO_COMP
10 if (reqtype == LOOKUP) obj_fh.comp = NO_COMP
11 if (reqtype == LOOKUP && !synced(resp.obj)) obj_attr.comp = NO_COMP
12 ... other request-specific rules ...
```

The rule on line 5 states that a number of subfields in an `fattr3` field are uncomparable. There can be 2 reasons for this. First, certain information may simply be different by virtue of the fact that the servers are separate entities – for instance, the physical space used by a file (`used`) depends on the block size of the underlying filesystem. Other information cannot be compared because in extremely rare instances, a property of an object may deliberately be allowed to diverge between the reference and developmental servers, thus causing the field that depends on it to be uncomparable. This is done due to practical concerns imposed by the design of the synchronization procedure. An example is the `nlink` field – the link count of a file. If a file's synchronization encompassed this field, all names for the file would have to be created before requests on it could be compared – a seemingly unnecessary sacrifice of comparison data.

Ideally, rules would be expressed in a simple high-level language of a form not unlike those commonly used in packet filters. Rules written in this language would then be interpreted at runtime. This would allow the ruleset to be modified at runtime without having to restart the tee. Because the NFSv3 protocol is of a tractable size and does not change often, rules are currently hard-coded. Each NFS request type is represented by a single "request class", which contains request-specific rules and inherits general and type-specific rules.

When a discrepancy is encountered, a log entry is made. This entry contains the request and response packets[5] and some object-specific statistics – number of successful/failed requests of each type, number of outstanding requests when the discrepancy occurred, etc. Periodically, server-wide statistics are logged – number of requests of each type (and the number of times each result code was seen), level of concurrency, etc.

Write sharing and the algorithm used to detect it are described in section 4.3. Because the actual execution order of shared writes cannot be determined at the tee, their results are uncomparable.

5.1. Comparison module

The comparison module consists of a thread group with a queue containing pointers to request table entries, each representing a request. Workers grab a unit of work, forward the request to the developmental server, wait for responses from both servers, then execute the comparison logic on the response pair.

5 In the case of requests that potentially take or return large amounts of data, like READ, WRITE, READDIR and READDIRPLUS, the amount of this data that gets logged is limited.

6. Evaluation

In all the following experiments, PostMark was used as the load generator, running on one client or multiple independent clients controlled from a central location. [client configuration, tee configuration, network configuration]. Unless specified otherwise, the reference server ran the Linux 2.6.5 kernel NFS server on a Debian 3.0 system. [ref server configuration].

[PostMark provides a rather biased workload – use SPEC SFS as well]

6.1. Relay & NFS plugin

To measure overhead on user activity, throughput (requests/second) and latency (difference between time the response was sent to the client and time the request was received from the client) are measured in the following experiments. In each experiment, each quantity is measured versus number of clients.

1. No tee, clients talk directly to reference server – baseline
2. Relay only – overhead of relay component
3. Relay and NFS plugin – overhead of plugin component

6.2. Synchronization

The effectiveness of the synchronization algorithms is characterized by how quickly the objects that make up the server's working set can be synchronized. Call this the “convergence rate”. To assess the convergence rate of the various combinations of core synchronization algorithm and priority assignment policy, we measure the proportion of requests that are comparable versus time, taking into account those that are uncomparable due to write sharing. For basic synchronization, only priority assignment policy 2 is used. For synchronization with change tracking, each priority assignment policy is used. This results in a total of 5 experiments.

Core algorithm

1. Basic synchronization (restart from scratch if object changes)
2. Synchronization with change tracking

Priority assignment policy

1. weighted moving average (WMA) of access frequency
2. WMA of access frequency with parent updates
3. random
4. zero (this produces FIFO behavior)

6.3. Comparison

Accuracy of comparison is defined as the proportion of implementation faults that are detected (assume that the faults produce detectable discrepancies). To measure this, the following 2 experiments will be conducted.

1. Use the unmodified Linux 2.6.5 kernel nfsd as the reference server. Augment the developmental server with a simple fault injection mechanism that injects “faults” both at the filesystem layer and the NFS server layer. For instance:
    1. Filesystem layer: change/remove/create files and directories
    2. NFS server layer: perturb response fields while maintaining a count of the number of requests affected, by request type

Run a workload through the tee to the 2 servers. Compare the number of discrepancies detected with the number of faults injected, either by object or by request type.

2. Run a workload through the tee to 2 identical servers. The number of discrepancies detected should be zero.

To measure the performance of the comparison procedure, rate of comparison and rate of request receival (both in requests/sec) are measured while varying the load factor.

An interesting additional experiment would be to measure the discrepancies detected while running a workload through the tee using two different existing NFS implementations as reference and developmental server, for instance the FreeBSD kernel nfsd and the Linux kernel nfsd.

7. Future work

Idle detection & rate throttling. Detecting periods of system idleness would allow the resources consumed by the synchronization and comparison modules to be controlled by limiting their rate of processing. This would help reduce the overhead on user activity.

Visualization & analysis tools. Making sense of the discrepancies logged is a task deliberately left to programs outside the tee for performance reasons. Tools that specialized in the analysis and user-friendly display of this data would complement the tee nicely.

8. Conclusion

[depends on the results obtained]

It is hoped that the tee will serve both as a proof of concept regarding a comparison-based approach to system verification, and as a valuable testing tool in the Self-* development cycle.

References

[1] Ganger, G., Strunk, J., Klosterman, A. 2003. Self-* Storage: Brick-based Storage with Automated Administration. Carnegie Mellon University Technical Report, CMU-CS-03-178, August 2003.

[2] Strunk, J. 2003. Private communication.

[3] Callaghan, B., Pawlowski, B., Staubach, P. 1995. NFS Version 3 Protocol Specification (RFC 1813). Sun Microsystems, Inc. June 1995.