CITI Technical Report 06-03

# Hierarchical Replication Control

*Jiaying Zhang*

jiayingz@eecs.umich.edu

*Peter Honeyman*

honey@citi.umich.edu

## Abstract

We present a hierarchical locking algorithm that dynamically elects a primary server in a replicated file system at various granularities. We introduce two lock types: shallow locks that control a single file or directory, and deep locks that lock everything in the subtree rooted at a directory. Experimental results show that for typical use cases, deep locks can make the overhead of replication control negligible, even when replication servers are widely distributed.

May 5, 2006

Center for Information Technology Integration
University of Michigan
535 W. William St., Suite 3100
Ann Arbor, MI 48103-4978

# Hierarchical Replication Control

Jiaying Zhang
jiayingz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

## 1. Introduction

Global scientific collaborations are characterized by elect member organizations sharing resources — compute and storage grids, instruments, and access — in dynamic virtual organizations [1, 2]. Rapid advances in storage and network technologies present new opportunities for creating and sharing massive data sets in these global virtual organizations. Concurrent advances in Internet middleware infrastructure — notably, near-universal deployment of NFSv4 [3, 4] — offer virtual organizations immense opportunities along a spectrum that includes supercomputer clusters at one end and global distribution at the other.

The performance and reliability advantages of data replication are especially relevant to global scientific collaboration. Collaborative access often requires shared access, so replicated data servers must specify and adhere to policies for concurrent update, such as ordered writes or a strict one-copy view. Collaborating scientists also need to know that data created or modified in replicated storage is durable.

Experience with a mutable replication extension to NFSv4 has shown that the needs of scientific collaborations are a good match for replicated storage, but the computations themselves, sometimes based on codes targeted for platforms of yore, can introduce extreme or peculiar behavior that affects performance [8].

Our replication extension to NFSv4 coordinates concurrent writes by selecting a primary server [7]. Unlike the conventional primary copy approach, we do not assign primary servers in advance, and allow any client to choose any relevant server when it opens a file. With no writers, the system has the performance profile of systems that support read-only replication (e.g., AFS): use a nearby server, support transparent client rollover on server failure, etc. Unlike read-only systems, we support concurrent access with writers. Performance penalties are slight, and are induced only when writers are active.

The system works as follows. When a server receives an update request, it forwards the request to the primary server for that object. If there is none, the server becomes primary by notifying other replication servers to forward any update requests for that object. The primary server lazily distributes updates to other servers. When the last writer is done, the primary server notifies the other replication servers that there is no longer a primary server for the object.

The protocol for electing a primary server can be delayed waiting for acknowledgments from slow or distant replication servers. To reduce the performance penalty, we are looking at ways to amortize costs over more requests. One way is to allow a primary server to assert control over a directory and its constituent entries, and beyond that to the entire subtree rooted at a directory.

Becoming the primary server for an object resembles the acquisition of a lock for the object distributed among all the replication servers. Electing a primary server with the granularity of a single file allows high concurrency and fine-grained load balancing, but a coarser granularity is more suitable for applications whose updates exhibit high temporal locality and are spread across a directory or a file system. The problem is therefore to find an appropriate locking granularity that balances the performance and concurrency tradeoff.

Lock granularity has been studied in database systems and distributed systems. Many modern transactional systems use hierarchical locking [5] to improve concurrency and performance of simultaneous transactions. In distributed file systems, Frangipani [11] uses distributed locking to control concurrent accesses among multiple shared-disk servers. For efficiency, it partitions locks into distinct lock groups and assign them to servers by group, not individually. Y. Lin, etc., study the selection of lease granularity when distributed file systems use leases to provide strong cache consistency [6]. To amortize leasing overhead across multiple objects in a volume, they propose volume leases that combine short-term leases on group of files (volumes) with long-term leases on individual files. Farsite [12], a decentralized distributed file system, uses content leases to govern which client machines currently have control of a file's content. A content lease may cover a single file or an entire directory of files.

In the next section, we look at some design choices to choose appropriate granularities during primary server election and the impact they have on performance in a replicated file system that intends global scale.

## 2. Lock granularity

We introduce two lock types: shallow and deep. A server holding a shallow lock on a file or a directory is the primary server for that file or directory. A server holding a deep lock on a directory **D** is the primary server for **D** and all of the files and directories in **D,** and holds a deep lock on all the directories in **D**. In other words, a deep lock on **D** makes the server primary for everything in the subtree rooted at **D**.

The two diagrams in Figure 1 describe a heuristic for supporting deep locks.

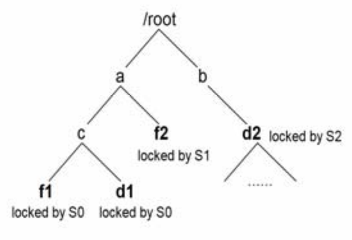
```
Update request for a file or a directory from a client

if the object is locked or any of its ancestors is deep locked then
        if locked by self then serve the request
        else forward the request
else if the object is a file then lock it
else if any descendant of the directory is locked by another server then lock it
else deep lock the directory
```

```
Lock request from a peer server

grant a request for a deep lock if
        the object is not locked or is locked by the peer server, and
        no descendant of the object is locked by a server other than the peer, and
        no ancestor of the object is deep locked by a server other than the peer

grant a request for a shallow lock if
        the object is not locked or is locked by the peer server, and
        no ancestor of the object is deep locked by a server other than the peer
```

**Figure 1. The locking protocol used in the election of a primary server.**

| | | |
|---|---|---|
| **Ancestry Entry** | | an ancestry entry has the following attributes |
| id | = | unique identifier of the directory |
| array of counters | = | set of counters recording which servers lock the directory's descendants |

**The data structure of entries in the ancestry table**



**Ancestry Table**

| Id | counter array | | |
|---|---|---|---|
| | S0 | S1 | S2 |
| root | 2 | 1 | 1 |
| a | 2 | 1 | 0 |
| b | 0 | 0 | 1 |
| c | 2 | 0 | 0 |

**Suppose the displayed replication unit has three replication servers: S0, S1, and S2. S0 currently locks file f1 and directory d1. S1 currently locks file f2. S2 currently locks directory d2. The right table shows the content of the ancestry table maintained on each replication server.**

**Figure 2. The structure and maintenance of entries in the ancestry table.**

When a replication server receives a deep lock request, it checks if the referred directory has any descendant currently locked by a different server. To avoid scanning the directory tree when receiving the request, we do some bookkeeping when locking objects.

Each replication server maintains an ancestry table for locked files or directories. An entry in the ancestry table corresponds to a directory that has one or more locked decedents. Figure 2 provides the data structure of entries in the ancestry table and an example that illustrates how the ancestry table is maintained.

The data structure of an ancestry entry contains an array of counters, each of which corresponds to a replication server. E.g., if there are three replication servers in the system, an entry in the ancestry table contains three counters accordingly. Whenever a lock is granted or revoked, each server updates its ancestry table by scanning each directory along the path from the locked object to the root, adjusting counters for the server that owns the lock. A replication server also updates its ancestry table appropriately if a locked file or directory is moved, linked, or unlinked during directory modifications.

With the ancestry table, the replication server can tell if a directory subtree holds a locked object in one lookup: It first finds the mapping entry of the directory from its ancestry table, and then looks over the entry's counter array. If any counter of a replication server, except the one that issues the lock request, has a non-zero value, the replication server knows that a different server currently locks some descendant of the directory. In that case, it rejects the deep lock request.

Deep locks reduce the number of locks in the system at the cost of traversing the path to the root when processing a lock request. To evaluate the performance benefit provided by this strategy, we compare the time to run SSH build benchmark in two series of experiments. In the first series of experiments, the system uses shallow locks only during primary server election. In the second series of experiments, the system uses both shallow locks and deep locks.

The SSH-Build benchmark [9] is constructed as a replacement for the Andrew file system benchmark [10]. It consists of three phases. The *unpack* phase decompresses the tar archive of SSH v3.2.9.1. This phase is relatively short and is characterized by metadata operations on files of varying sizes. The *configure* phase builds various small programs that check the configuration of the system and automatically generates header files and Makefiles. The *build* phase compiles the source tree and links the generated object files into the executables. The last phase is the most CPU intensive, but it also generates a large number of temporary files and a few executables.

Figure 3 presents the measured performance when using deep locks versus without using deep locks. The results show dramatic improvement: deep locks make the overhead of replication control negligible, even when replication servers are widely distributed.

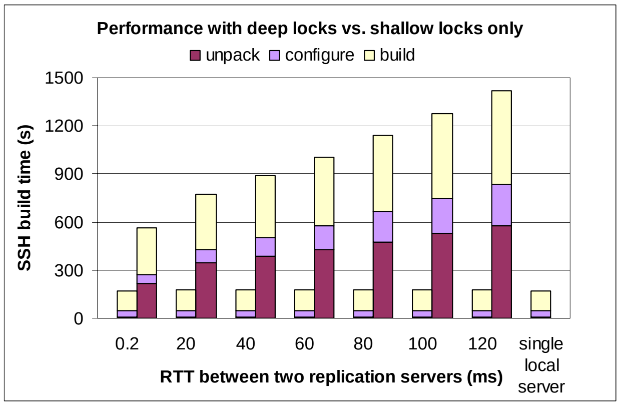


**Figure 3. Comparing deep and shallow locks.** The first column shows clock time when the primary server uses deep locks. The second column shows the time when the primary server uses only shallow locks. For the deep lock runs, a primary server relinquishes its role if it receives no further client updates in two seconds.

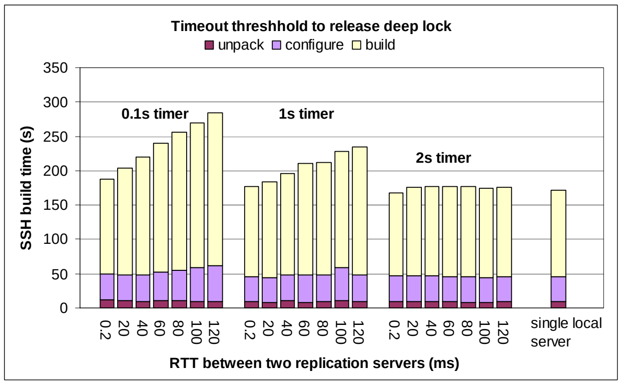


**Figure 4. Comparing timeout values to release a deep lock.** The diagram shows the time to build SSH in the presented replicated file system when we set the timeout to release a deep lock to 0.1 second, 1 second, and 2 seconds.

The introduction of deep locks introduces a performance and concurrency tradeoff. On the one hand, because a primary server can process any client update under a deep locked directory immediately, it significantly improves performance when an application issues a burst of updates. On the other hand, it increases the possibility of conflicting updates, i.e., concurrent updates received on different replication servers, due to false sharing.

We use two strategies to reduce false sharing. First, we postulate that the longer a server remains primary, the more likely it is that it will receive conflicting updates, so we start a timer on a server when it grants a deep lock. The primary server resets its timer if it receives a subsequent client update request under the locked directory before timeout. When the timer expires, the primary server relinquishes its role. Initial experiments that measure the time to build SSH, shown in Figure 4, suggest that a timer value approximately a couple seconds captures most of the busty updates.

Second, when the primary server receives a client write request for a file under a deep locked directory, it distributes a new lock request for that file to other replication servers. The primary server can process the write request immediately without waiting for replies from other replication servers since it is already the primary server of the file's ancestor. However, with the file locked, subsequent writes on that file no longer reset the timer of the locked directory. Thus, a burst of file writes has little impact on the duration that a primary server holds a deep lock. It also allows us to use a longer timeout threshold for an open file, further reducing the number of replication control messages distributed in the system.

## 3. Conclusion

Consistent mutable replication in large-scale distributed file systems is widely regarded as being too expensive for practical systems, yet with a little engineering, we show that it can have negligible impact on application performance. One of the ways to reduce the cost of replication is to use a form of hierarchical locking to coordinate concurrent updates among replication servers. Our experiments show that hierarchical locking is very effective in shaving overhead, especially when timeouts are introduced.

## References

[1] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke. "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *J Network and Computer Applications* (2001).

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann (1998).

[3] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," *2nd Intl. Conf. on System Administration and Network Engineering*, Maastricht (2000).

[4] Sun Microsystems, Inc., "NFS Version 4 Protocol," RFC 3010 (2000).

[5] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," *IFIP Working Conf. on Modeling in Data Base Management Systems* (1976).

[6] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. "Volume Leases for Consistency in Large-Scale Systems," *IEEE Trans. on Knowledge and Data Engineering* (1999).

[7] J. Zhang and P. Honeyman, “Naming, Migration, and Replication for NFSv4,” *5th Intl. Conf on System Administration and Network Engineering,* Delft (2006).

[8] J. Zhang and P. Honeyman, “Reliable Replication at Low Cost,” Technical Report 06-01, Center for Information Technology Integration (2006).

[9] T. Ylonen, “SSH - Secure Login Connection Over the Internet,” *6th USENIX Security Symp.*, San Jose (1996).

[10] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” *ACM ToCS* (1988).

[11] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A Scalable Distributed File System,” *SOSP* (1997).

[12] A. Adya, W.J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, R.P. Wattenhofer, “FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment”, *OSDI* (2002).