A Low-Cost Consistency Protocol for Replicated Directory Data in Cluster-Based Storage Systems

Minwen Ji
Princeton University
mji@cs.princeton.edu

Abstract

We discuss a low-cost consistency protocol for replicated metadata in cluster-based storage systems. Our goal in maintaining the consistency is to minimize the efforts for porting applications from single systems to the cluster-based environment. The access patterns to the replicated metadata in recent cluster-based storage systems offer a new opportunity for strong consistency without sacrificing performance in common cases. We design and implement a protocol for the atomicity, serialization and recovery of operations in the face of arbitrary sequences of failures. The correctness of the protocol is checked with randomized failure injections into a prototype implementation. We measure the impact of the consistency protocol on the performance and scalability of an example cluster-based file system. The measurement of micro benchmarks shows that the protocol adds little overhead to common operations; while the measurement of trace-based operation mixes shows a speedup of 15.7 in a cluster of 16 nodes.

1 Introduction

In many cluster-based storage system, e.g. file systems [26] [1] [17], memory/cache systems [7] [8] and virtual disks [20], a small amount of metadata is replicated across nodes in the cluster for fast local lookups. The access pattern to the replicated metadata in those systems can be characterized as “read mostly, write critically”. For example, the manager map in xFS [1], which maps each file’s index number to the file’s manager, who manages cache consistency and disk location of the file, is globally replicated. The manager map will be looked up upon the opening of each file, while it will change only to correct a load imbalance or when a machine enters or leaves the system. When it does change, however, it is critical that the change is made in an atomic, serializable and recoverable way, because the integrity of metadata is critical for the system to function correctly and to recover successfully from possible failures, and because corrupted metadata affects not only individual files or clients but also the entire system.

In this paper, we focus on two recent systems, an island-based file system [17] and an affinity-based management system for clustered in-memory databases [8]. The replicated metadata in these two systems shares the same “read-mostly, write-cirtically” access pattern as that in other cluster-based storage systems. However, in these two systems, the replicated metadata includes not only system internal metadata but also *directory data*, which is created by clients to organize or locate other data, such as directories in file systems and search keys in databases.

To motivate the design of a consistency protocol for the replciated direcotry data, we give the following two examples of possible hazards in a distributed file system where directories are replicated across servers and clients are allowed to access any replicas:

1. An empty directory a is replicated in cluster servers 1 and 2; client B deletes directory a in server 1 and server 1 propagates the deletion request to server 2; simultaneously, client C creates a sub directory d in a in server 2 and server 2 propagates the creation request to server 1; the deletion is aborted in server 2 because a is not empty and the creation is aborted in server 1 because a no longer exists; in a consistent system, only one, not both, of the operations would abort.
2. A directory a with a file b is replicated in servers 1 and 2; client C, the owner of a, changes a’s permission from 700 to 755 (world-readable) in server 1 and server 1 propagates the change to server 2; client D successfully reads file b in server 1 but, shortly after, it gets a ”permission denied” error when it tries to list the content of directory a in server 2. In a consistent system, client D is expected to have access to directory a as well after it successfully reads file b.

The hazards occur because the file system operations are not serialized, or clients observe the results of the operations in conflicting orders; the consequence is that the system no longer behaves in the same way as its single-system counterparts and start generating confusing or incorrect answers to clients' requests. Furthermore, the chance for such hazards is highly magnified when failures, such as server crashes and network partitions, are present.

Our goal in maintaining the consistency of the replicated directory data is to minimize the efforts for porting applications from single, tightly-coupled and/or small-scale systems to large cluster-based environments. In particular, we want to eliminate as many hazards as possible that a cluster environment might introduce. Meanwhile, we do not want the consistency protocol to have an intolerable impact on the performance and scalability otherwise achievable in these two systems.

Both the island-based design and affinity-based management offer an opportunity for strong consistency without sacrificing performance in common cases. Since they strive to reduce data sharing across nodes and the replicated directory data exibits "read-mostly, write-critically" pattern, the cost for maintaining consistency of shared data can potentially be reduced as well. Therefore, it is possible to achieve strong consistency for the small set of shared data while maintaining the overall performance and scalability of the system.

We discuss in this paper how to design a low-cost protocol for the synchronization of operations on replicated directory data in the face of node failures and network partitions.

2 Related work

File system replication and consistency issues have been studied in a wide variety of contexts. Consistency guarantees vary largely from system to system due to the differences in their system structures and replication models.

Wide-area distributed file systems such as Ficus [24], Coda [25] and Locus [28] employ optimistic one-copy availability, in which any data may be updated as long as some copy, including the client cache, is available. Strong semantics such as serialization of operations on replicated data are traded for availability and performance in those systems. The systems choose to guarantee "eventually" consistent data instead, i.e. they allow temporary inconsistency and try to detect it, which must then be resolved by applications or users. (The exception is that Ficus can automatically reconcile conflicting updates to its directories.)

Harp [21] and Echo [14] use primary-copy scheme with logging for replication, where clients can access only the primary copy. Harp is able to guarantee the atomicity and serialization of updates with write-behind logging. Since it handles updates to data and metadata in the same way, it relies on in-memory logging and uninterruptible power supply (UPS) to reduce the overhead of the consistency protocol. In a recent distributed file system [27], the overhead is reduced by distributing load across servers and amortizing the costs of individual operations with file sessions.

In recent cluster file systems like Frangipani [26] and xFS [1], data redundancy is provided in the virtual block device layer, not in the file system layer. Locking scheme is used in the block device layer for consistency of data replicas. Since updates to data and metadata are handled in the same way in the block device layer, those systems typically use fast system area network such as ATM for aggressive communications across data replicas [20].

The replication model generalized from we systems is similar to the models in typical replicated databases [4] [3]. However, our consistency requirement differs, primarily because we do not require transactional semantics for file accesses or persistence for in-memory data. Therefore, we believe that a light-weight protocol can be designed for the model in our systems.

The Cluster-Based Scalable Network Services (SNS) [9] provide an architecture and programming model for building Internet services that are willing to trade consistency for availability. Our approach is complementary to theirs in that, while their system can be used for creating new, scalable Internet services on loosely-coupled clusters, we strive to make it easy to run existing applications, such as the well-adopted web servers [15], database servers [16] and database applications in a cluster as well as on a single machine.

3 Our contributions

The context of system structure and replication model in which our consistency protocol is considered differs from the contexts in previous studies. In the island-based file system, only certain directory attributes, but not the directory contents or files, are replicated across islands. The degree of replication varies by directories based on their usage, and changes dynamically as the usage changes. In the clustered in-memory database, only search keys, but not massive data, are replicated, and the replicated data is stored in memory only. The contributions of this paper are the following:

1. We design a consistency protocol that offers stronger semantics than "eventual" consistency, and hence increases the likelihood that applications can be ported from single systems to cluster-based systems with few modifications.

2. The overhead of the consistency protocol is reduced by taking advantage of the data distribution strategies in the target systems and using a light-weight non-locking algorithm, rather than using additional hardware or fast network.

3. We take arbitrary sequences of failures into consideration and use a recovery procedure based on a finite state machine model to handle the failures. We check the correctness of the protocol by randomized failure injection into a prototype implementation.

4 Overview of two cluster-based storage systems

In this section, we summarize the characteristics of two cluster-based storage systems that are relevant to the discussion of consistency protocol design in the rest of the paper. The motivation, reasoning and design details of the two systems are outside the scope of this paper. Interested readers should refer to separate publications [17] [8].

4.1 Island-based file system

An *island-based* file system is a clustered file system with a failure isolation mechnism that partitions and replicates data and metadata across cluster nodes in such a way that the server in each node can deliver data to clients independently of the failures in other nodes. This approach is complementary to existing redundancy-based methods: redundancy can mask the first few failures, and failure isolation can take over and maintain availability for the majority of clients if more failures occur. The building blocks of such a file system are self-contained, load-balanced and off-the-shelf file servers called *islands*. The main idea underlying the island-based design is the one-island principle: as many operations as possible should involve exactly one island. The one-island principle offers improved availability because each island can function independently of other islands' failures. It also helps the file system scale efficiently with the system and workload sizes because communication and synchronization across islands are reduced.

The target applications of island-based file system are those Internet services that prefer to serve as many clients as possible rather than to go entirely offline when partial failures are present, that are medium to large scale, e.g. tens to hundreds of commodity PC's connected by commodity local area networks, and that expect occasional node failures and network partitions. Examples include email, Usenet newsgroup, e-commerce, web caching, and so on.

In an island-based file system, data is distributed to islands at directory granularity by hashing the pathnames of the directories to island indices. The file system running inside each island is called the *internal* file system. An internal file system can be an instance of any existing file system such as a local file system, a replicated file system or even another cluster-based file system. Inside each island, directories are stored in a skeleton hierarchy. The skeleton hierarchy in an island contains the directories hashed to this island index and their ancestor directories up to the root, and is stored in the unmodified internal file system as a normal tree. This way, the data stored in each island is made self-contained and the built-in functions of the internal file systems can be leveraged.

The consequence of storing data in skeleton hierarchies is the replication of directory attributes that are needed when a descendent of the directory is looked up. Such attributes include name, security, read-only tag, compressed tag, etc. Updates to those attributes need to be propagated to all replicas. The overhead of updates is acceptable since those attributes rarely change [17]. The replication scheme is a usage-based adaptive scheme, i.e. We replicate attributes for directories that are more frequently used to a higher degree. Directory contents or files are not replicated across islands, but data redundancy can be used inside each island to improve reliability.

The majority of operations in an island-based file system, such as CreateFile, WriteFile and ReadDirectory, involve exactly one island and are called *one-island* operations. The following operations involve the replicated directory attributes and are called *cross-island* operations: CreateDir, RemoveDir, SetDirAttr, SymLinkDir, DeleteLinkDir and RenameDir.

The consistency protocol presented in this paper shall handle the cross-island operations in the face of partial failures.

4.2 Affinity-based management system for clustered in-memory databases

An *affinity-based management* (*ABS*) system for clustered in-memory databases was designed for the following application server infrastructure. A number of *processing nodes* in the infrastructure run web servers and application servers, and each web server or application server is capable of processing any HTTP or application-specific requests. The application servers store all their persistent data in a shared, on-disk database, which is called the *master database*. Redundancy may be applied inside the master database for high reliability and scalability. A cluster of in-memory databases are transparently situated between the application servers and the master database, caching partial or all data from the master database for speedy access from the application servers. *In-memory databases* [10] are optimized in many aspects, such as buffer management, retrieval and indexing, specifically for memory-resident data, and is lightweight enough to be hosted on the same machine as application servers. An individual in-memory database in the cluster is called a *cache server*. Data accessed by a query will be loaded from the master database into a cache server before the query is executed in that server. Data will stay in the cache server until it is evicted by a cache replacement policy or invalidated by a synchronization protocol.

The task of ABS is to automatically and dynamically partition and replicate data across individual in-memory databases in the cluster and directs queries to the right databases, in order to maximize effective cache capacity and minimize synchronization cost. The basic method in ABM is to divide the execution of each query into two stages and to use the result from the (local) first-stage execution to determine the (remote) destination of the second-stage execution. The first stage is computation-intensive; the function and data needed for the first-stage execution are replicated on each machine where a database client is installed. The second stage is data-intensive; data accessed during the second-stage execution is partitioned across individual in-memory databases at row granularity. The first stage determines the set of data that the query will likely access, which is then used to determine the destination of the query in its second stage. The second stage completes the query execution and generates results. The intention of the two-stage execution is to determine the destination of each query with the knowledge of the data to be accessed.

The replicated data for the first-stage execution is the columns or search keys used in the selection conditions of queries, which are typically of short data types, such as integers, timestamps, short character strings, etc. The replication enables the database clients to locally execute first-stage queries. And yet the space required for replicating search keys is strictly less than the space required for replicating the entire table. Any update to the database will be executed in the master database before the update operation is completed. Any update that results in modifications to the replicated keys, such as insertion, deletion or update operations on the replicated columns, will be broadcast to all replicas.

The consistency protocol presented in this paper shall handle the updates to replicated keys in the face of partial failures.

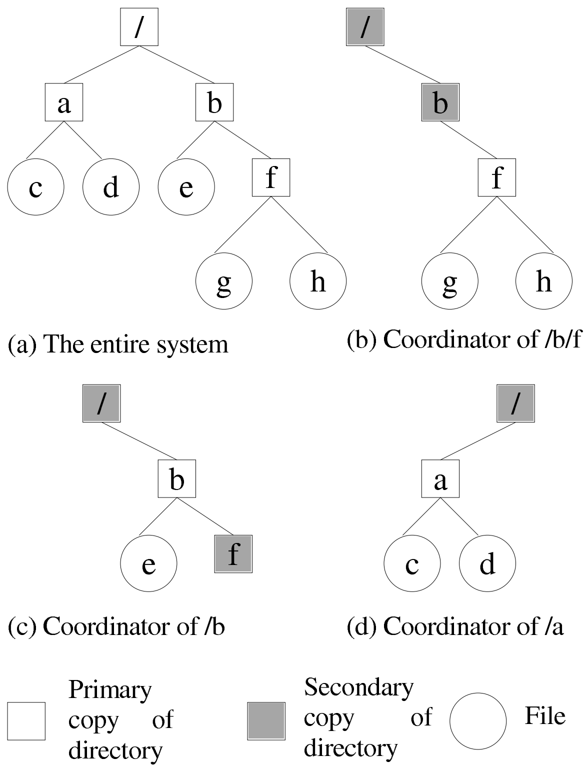


Figure 1: Replication of directories. Figure (a) is the image of an entire system. (b) (c) and (d) are the images of the internal file systems in three other islands. Shaded directories in the figure represent replicas that contain only attributes and partial contents or no contents.

5 Replication model

We define a few terms below to assist in generalizing the replication model in the island-based file system and the clustered in-memory database. For each replicated object (a set of attributes of a directory or the entire set of search keys), a particular node (the directory owner or the master database server) is chosen as the coordinator of the global operations or updates on this object, or simply called the coordinator of the object. The copy of an object in its coordinator is called the primary copy and the other copies are called secondary copies. Any node (an island or a pre-executor) that has a copy of the object is called a replica of the object. Each operation originates from a single replica. Each update must originate from the coordinator and be propagated to other replicas. All objects in a node are readable by operations originated from or propagated to this node. Figure 1 illustrates the replication in the island-based file system.

The cache servers in the clustered in-memory database are a special case: they do not have a copy of the replicated keys unless they are co-located with the pre-executors, but they participate in the consistency protocol as replicas because it is implicitly assumed that the cache servers have a consistent view of the database state with the pre-executors. To determine the set of cache servers involved in an update, the distributor is consulted before the commit of the update starts.

6 Consistency protocol design

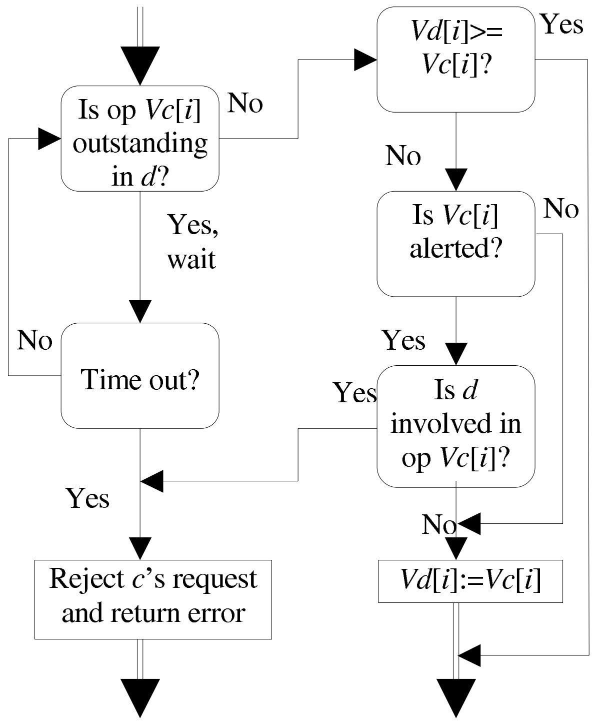
A strawman's approach to the consistency of replicated directories across replicas is to lock a directory before operating on it. Locking schemes, especially ones with multi-reader-single-writer locks, are a typical approach to the consistency on replicated data in general. To avoid deadlocks and to handle partial failures and network partitions, a locking scheme often needs to be used in combination with other mechanisms such as timeout [23], majority consensus [26] and/or versioning [2].

Unfortunately, such a scheme can seriously weaken the availability and scalability. In the island-based file system, since each operation implicitly involves recursive lookup and permission checking with the ancestor directories, the ancestor directories need to be locked for the operation as well. A lock on each directory requires at least two round-trip messages, acquiring the lock and releasing/revoking the lock, to and from the coordinator of the directory. Consequently, there will no longer be "one-island" operations in the island-based file system since almost every operation needs to contact multiple islands for locking involved directories. The same overhead is also inevitable in the clustered in-memory database. If We use a global lock for the entire system rather than a lock per replicated object, We can reduce the communication cost for locking, but We also reduce the parallelism offered by the cluster structure.

We use a novel combination of logical clock synchronization [19], two-phase commit [12], logging [13] and finite-state-machine-based recovery to serialize the updates while keeping the synchronization for one-island operations or read-only queries local. Our methodology takes three steps. First, we guarantee that each update is atomic; second, we serialize updates and other operations in common cases; third, we ensure the serialization of updates during a recovery from failures.

6.1 Atomicity

The basic consistency guarantee our protocol offers is the atomicity of the updates, i.e. clients would never observe the intermediate state of any update. In other words, once a client observes the result of an update in a node, it would always observe the result of that operation in other replicas afterwards.



Is op *Vc*[*i*] outstanding in *d*?
No
Yes, wait
No
Time out?
Yes
Reject *c*'s request and return error
Vd[*i*]>= *Vc*[*i*]?
Yes
No
Is *Vc*[*i*] alerted?
No
Yes
Is *d* involved in op *Vc*[*i*]?
Yes
No
Vd[*i*]:=*Vc*[*i*]

Figure 2: Synchronization of a client c's clock Vc[i] with the island d's clock Vd[i]. Op Vc[i] is the cross-island operation that generated the clock value Vc[i] in its coordinator, island i.

We use a vector of logical clocks for the atomicity of updates. Each coordinator has its local logical clock and each update coordinated by this node increases the clock by 1, or generates a new clock value. Each replica or client maintains a vector of all coordinators' clocks. Each request to a replica carries the sender's current clock vector for synchronization with the receiver's vector before the request is processed, and returns the receiver's vector to the sender after the request is completed. We say vector V_2 is equally or more up-to-date than vector V_1, or $V_2 \geq V_1$, if and only if $V_2[i] \geq V_1[i], 0 \leq i < n$, where n is the number of coordinators.

We maintain the following invariants:

1. The local commit of an update and the increase of the local clock are atomic in each coordinator, which is guaranteed with a local lock in that coordinator.

2. A coordinator does not release the new clock value to a client until it has notified all replicas of the operation, i.e. until the operation is either outstanding or committed in all replicas. This is guaranteed with a two-phase commit [12]: the coordinator notifies all replicas of the operation in phase 1, then locally

commits the operation and updates the clock, and asks replicas to commit the operation in phase 2.

3. A request cannot be processed in a replica if the request carries a clock that is generated by an outstanding operation in that replica. Based on invariants 1 and 2, this invariant means that once a client observes the result of an operation in at least one replica, it will always observe the result of that operation in other replicas afterwards. This is guaranteed by the clock synchronization algorithm in Figure 2, which is an extension to Lamport's algorithm [19].

The three invariants above guarantee that a replica will never expose the intermediate state of any operation to clients. Invariant 2 ensures that synchronization in a replica for reads does not need communication with the coordinator, if no network partition is present.

We make an exception to invariant 2 to handle network partitions. If any replica is inaccessible due to either a node crash or network partition during phase 1 of the commit, the coordinator updates its clock with an alerted bit set, which will be propagated to the clients together with the clock. During the clock synchronization with a client, a replica must ask for a confirmation from the coordinator about its involvement in an alerted operation that it has not seen but the client has. If the coordinator crashed or disconnected from a replica after phase 1, the operation will be outstanding in the replica till the coordinator reconnects. This type of failure will be detected by a timeout in the clock synchronization. See Figure 2. The alerted bit will be cleared once the nodes reconnect and all outstanding operations are either committed or aborted.

6.2 Serialization

The higher-level consistency guarantee our protocol offers is the serialization of the updates, i.e. clients observe the results of all operations in the same order in all replicas. All the updates on the same object are coordinated by the same node, hence can be serialized by a local mutex in that node, unless a replica failed.

The serialization in case of failures is guaranteed by write-ahead logging [13]. The coordinator always writes a record with its clock vector to stable storage before it locally commits an update. Only after the operation is committed in all replicas, the record can be removed from the log. (However, the secondary copies in the clustered in-memory database will be completely lost during node crashes. The write-ahead logging is not necessary in this case because, during the recovery of a pre-executor after a crash, a complete snapshot of the replicated keys, rather than updates that occurred after the crash, will need be copied from the master database.)

When a replica b is reconnected, the coordinator a sends to b a list of operations that involved b but have not been committed on b. The operations will be committed in b in ascending order of their clocks (V[a]'s), i.e. in the same order as if b had not been disconnected from a. Note that b needs not know about the local operations on the same objects that were done while it was disconnected from a because it would not have known those operations even if it had not been disconnected.

If a client thread issues at most one request at a time, all the operations by the same thread are serializable even if a replica failed. Consecutive operations by the same thread are guaranteed to have ascending clock vectors because, with the logical clock synchronization (Figure 2), the clock vectors in all replicas and clients never decrease and always increase upon updates, even with network partitions. Therefore, recovering replicas are able to commit the operations by the same client thread in the same order as if it had not failed, by sorting the operations from all coordinators in the ascending order of their clock vectors.

If two clients interact with each other by accessing the same objects, then the operations by the two clients are serializable in the face of failures. For example, if two clients, c_1 and c_2, access the same object at time t_1 and t_2 ($t_1 < t_2$) and receive the clock vectors V_1 and V_2 respectively, then $V_1 \leq V_2$ because the vectors are issued by the same replica; therefore, c_1's operations before t_1 (with vectors $< V_1$) and c_2's operations after t_2 (with vectors$> V_2$) are serializable.

Clients that do not interact through accesses to the same objects might have concurrent clock vectors. We say two vectors V_1 and V_2 are concurrent if and only if there exist i and j, $i \neq j$ and $0 \leq i, j < n$, such that $V_1[i] < V_2[i]$ and $V_1[j] > V_2[j]$, where n is the number of coordinators. During a failure recovery, concurrent vectors will be sorted with a simple tie resolution rule consistent across all replicas, which does not necessarily reflect the real-time ordering. The reordering of concurrent operations would not be observable and could not cause problems as long as the replicated objects were concerned [19].

6.3 Recovery

We have designed a recovery procedure for replicas to recover from arbitrary sequences of failures back to consistent states. Table 1 shows the possible failures for an individual replica and how the replica can be recovered from those failures.

Given the finite set of possible failures and the infinite set of possible sequences of the failures, we find it a good

Failures	Definitions	Examples	Recoveries
Self Failures	Any failures that stop the island itself from functioning	Software failures, machine crashes, disk failures, power failures	Rerun software, reboot machines, repair disks, restore power
Peer Failures	Any failures that make other islands inaccessible from this island	Self failures of other islands, network partitions	Recover other islands, repair networks

Table 1: Possible failures and recoveries for an individual replica

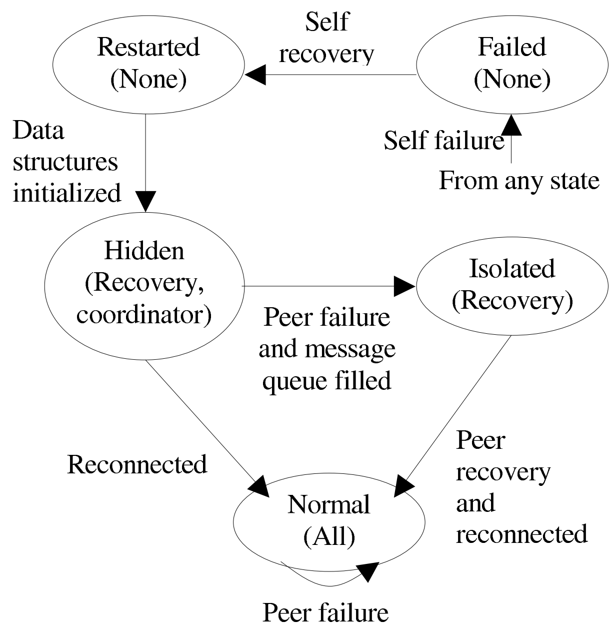

Restarted (None)
Self recovery
Failed (None)
Data structures initialized
Self failure
From any state
Hidden (Recovery, coordinator)
Isolated (Recovery)
Peer failure and message queue filled
Reconnected
Peer recovery and reconnected
Normal (All)
Peer failure

Figure 3: State transitions of an island in response to various failures and recoveries. The types of requests accepted in each state are listed in parenthesis. Each transition is labeled with the event that triggers the transition. "Reconnected" is the event that the recovering island has reconnected to and resynchronized with all other islands.

practice to model the recovering replica as a finite state machine, in which each state corresponds to a set of behaviors that are allowed in the recovering replica, and each state transition is triggered by a failure or recovery event. Figure 3 shows the state transitions of a replica in response to the possible failures and recoveries. A replica can be in one of the 5 states, normal, failed, restarted, hidden and isolated. Each state is distinguished from others by the types of requests the replica is allowed to process in that state. The types of requests a replica receives include client requests (from the clients), coordinator requests (from the coordinators of updates), recovery requests (from the recovering or reconnecting replicas), etc.

In the normal state, a replica processes all requests. A self failure in any state causes the replica to transit to the failed state, in which no requests, of course, are processed. When it is recovered, a replica transits from the failed state to the transient restarted state, in which it initializes necessary data structures while rejecting all requests. It automatically transits to the hidden state after all data structures are initialized. In the hidden state, it attempts to reconnect to other nodes and to synchronize replicated state with other nodes by log exchanges. In the hidden state, the replica rejects all client requests so that inconsistency, if it is present in the replica, is not visible to clients. The replica accepts requests from other recovering or reconnecting nodes so that both can make progress. It also accepts requests from the coordinators of new updates and stores them in a message queue for sorting with other operations when all have arrived. If the queue becomes full, the replica transits from the hidden state to the isolated state, in which it accepts no more coordinator requests. (Note that the buffer for keeping outstanding operations in the normal state will never be filled because there is at most one outstanding operation per coordinator in the buffer.)

When all nodes have reconnected and exchanged logs with it, the replica commits all the operations stored in the message queue in the ascending order of their clock vectors. If it is in the isolated state, it asks for new operations from coordinators that it has rejected. After it commits all pending operations, it transits to the normal state.

7 Implementation

We have implemented the consistency protocol as a library, which consists of approximately 1800 lines of C++ code. We have used the library in a prototype of the island-based file system called *Archipelago*. Archipelago [17] runs on a cluster of Pentium II PCs with Windows NT 4.0. Cross-island operations in the prototype call the consistency protocol library for the functions on logical clock synchronization, two-phase commit, logging, etc.

At startup, the island servers call the recovery procedure in the library. Win32 RPC (Remote Procedure Call) is used for cross-island communication. Due to its speciality and simplicity, the clustered in-memory database needs to call only a subset of the functions in the library.

8 Correctness

As discussed in the previous sections, the combination of logical clock synchronization, two-phase commit and write-ahead logging maintains the following invariants in the face of failures:

1. All updates on the replicated directory data are atomic.

2. All updates on the replicated directory data are serialized.

3. In most cases, read-only operations can be processed locally, i.e. without contacting other replicas for synchronization purpose.

The correctness of the systems that use this consistency protocol largely relies on the details in implementation, which are hard to model or check using existing tools [6] [5]. Therefore, we use a randomized test engine to test the correctness of the protocol instead. The test engine is extended from a model checker originally developed in Hewlett-Packard Labs [11]; the model checker is based on the input/output automata (IOA) [22]. We extended the tool so that it checks the implementation of a system, rather than a simulation written in IOA style. Unlike the tools that exhaustively search the state space [6] [5], the randomized testing tools cannot prove that a system is correct. Instead, it helps identifying incorrect parts of a system by injecting various sequences of events to the system and analyzing the results. Such events typically could not possibly be experienced in real workloads or manual tests during a short period of time.

Archipelago is tested with the randomized test engine. The test engine consists of three components, terminators, network partitioner and clients. The terminators are independent threads or processes, one for each replica. Each terminator injects crash or reboot events to its associated replica at intervals randomly chosen within given ranges. It simulates a crash of the replica by killing the server process of that replica, and the reboot of the replica by forking a new server process for that replica. The network partitioner is an independent thread that simulates network partitions between replicas. At random intervals, it randomly chooses a pair of replicas and sends a message to both replicas to tear down or to reestablish the connections between them. Since multiple pairs can be disconnected this way, a sequence of such events can generate complicated partitions. The clients are multiple threads that share the same set of objects (files, directories and symbolic links) in Archipelago. Each client generates workloads on the file system by repeatedly issuing a randomly chosen request with given frequencies on a randomly chosen object.

The IOA formal language has an interface for defining models for safety and liveness checking [22]. A safety model specifies a property that must hold at any time, while a liveness model specifies an event that must eventually occur. A prototype of the interface was implemented in the original tool, but We have not ported it to the test engine yet. Instead, we check the safety of the protocol by manually inserting assertions to key parts of the code. A few examples of the assertions are: there is at most one outstanding operation coordinated by each node at any given time; there is no gap and no overlap in the clocks of the operations coordinated by the same node; the coordinator i always has a more or equally up-to-date clock V[i] than any other replicas or clients; etc.. These assertions have been surprisingly helpful in our preliminary experiments. Liveness assertions such as that a replica will eventually transit from the failed state to the normal state in the recovery procedure will be added once the system has passed the simpler tests.

The test engine takes parameters such as the interval ranges of failure/recovery events, and the relative frequencies of operations. We selected the intervals in such a way that they both allow a sufficient workload in each state of the system, and allow the overlap of failure/recovery events to exercise the recovery procedure. We exaggerated the frequencies of updates from real workloads by two orders of magnitude to stress the consistency protocol. We tested Archipelago with 4 islands in the randomized test engine. Table 2 shows the parameters and results in our latest test. After surviving through 28 node crashes and 7 network partitions, Archipelago failed one of the assertions and caused the test engine to halt.

We found 14 non-obvious bugs in the protocol during two days of testing Archipelago. The bugs are all at implementation detail level and do not invalidate the overall protocol design. An example of the bugs We found is following. The coordinator of an update crashed after it notified the replicas of the operation, but before it logged the operation on disk. Therefore, the operation was aborted in the coordinator, but outstanding in the replicas. When the replicas received the next operation from the same coordinator later, the assertion of at most one outstanding operation per coordinator failed. The fix to this bug is to clear the relevant buffers of outstanding operations upon reconnection of two nodes.

Events	Parameters	Numbers of Events
CreateDir	3.23%	1565
CreateFile	2.82%	1369
DeleteFile	1.92%	974
DeleteLinkDir	0.81%	221
ReadDir	11.22%	5273
ReadFile	13.15%	8162
RemoveDir	2.42%	1469
ResolveLinkDir	7.34%	530
SetDirAttr	5.65%	2609
SetFileAttr	21.98%	14970
SymLinkDir	0.81%	227
WriteFile	28.65%	16394
Crash	60 to 120 sec	28
Reboot	8 to 16 sec	24
Partition	15 to 30 sec	7
Reconnection	2 to 4 sec	4

Table 2: Parameters and results in testing Archipelago in the randomized test engine. The parameters are the given frequencies for normal operations and the given interval ranges for failure/recovery events. For example, each time a client randomly chooses an operation, the probability that CreateDir is chosen is 3.2279%; the terminator waits for an interval randomly chosen from 60 to 120 seconds each time before it kills the server process. The results are the actual numbers of successful operations or events in the test. The actual numbers are different from the specified values due to randomization, race conditions and simulated failures. The operations SymLinkDir, ResolveLinkDir and DeleteLinkDir are creating a symbolic link to a directory, reading the directory entries in a symbolic link to a directory and deleting a symbolic link to a directory, respectively.

Both the development of the test engine and the correctness checking of Archipelago are in a very early stage. However, the preliminary results are encouraging, and we believe that randomized failure injection is a promising approach to checking the implementation correctness of a complicated system.

9 Performance

In this section, we present the results of measuring Archipelago with the following metrics: 1) overhead of the consistency protocol in simple cases; 2) impact of the consistency protocol on the scalability of cross-island operations; 3) impact of the consistency protocol on the overall scalability of Archipelago.

The machines used in our experiments have Pentium

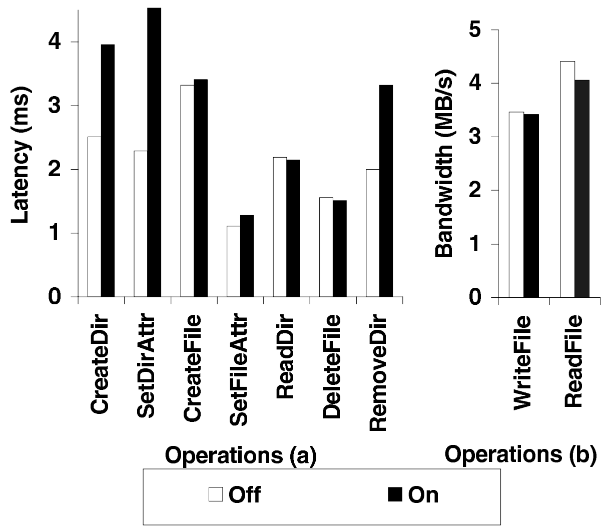


Figure 4: Single client performance. A single client runs the micro benchmarks with the consistency protocol turned on and off, respectively. The y-axis in (a) is the latency in milliseconds measured at the client side. Lower columns represent better performance. The y-axis in (b) is the bandwidth in MB/s in the WriteFile and ReadFile operations measured at the client side. Higher columns represent better performance.

II 300 MHz processors, 128 MB main memories and 6.4 GB Quantum Fireball IDE hard disks for use by Archipelago. The PCs are connected by a 3COM SuperStack II 100Mbps Ethernet hub. The PCs run Windows NT Workstation 4.0 and the hard disks for Archipelago are formatted in NTFS.

9.1 Single client performance

We use a set of micro benchmarks that consists of 9 phases and each phase exercises one of the file system operations: CreateDir, SetDirAttr, CreateFile, SetFileAttr, ReadDir, WriteFile, ReadFile, DeleteFile and RemoveDir. The data set for the micro benchmarks is an inflated project directory that consists of 3600 directories, 3876 files and 154.4 MB of data in files. The 3876 files are stored in 540 directories and the rest of the directories are empty. Disk space is pre-allocated for each file in the CreateFile phase. The transferred block size in the WriteFile and ReadFile phases is 64 KB or the file size, whichever is smaller. Each test is run more than 3 times and the results shown in this section are the averages.

We run the micro benchmarks with a single client and two servers (or islands) in Archipelago. We turn on and off the consistency protocol to measure its overhead

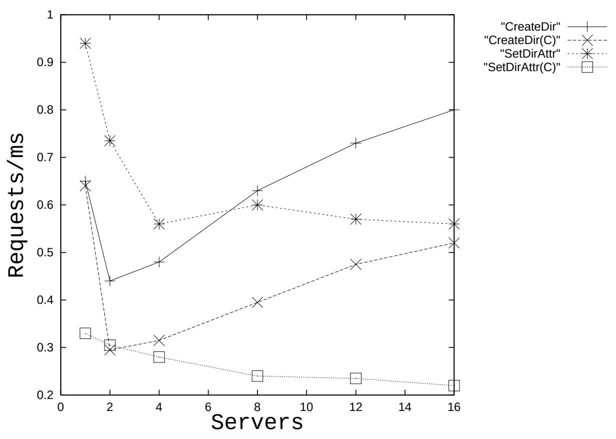


Figure 5: Impact of the consistency protocol on cross-island operations. The CreateDir(off) and SetDirAttr(off) curves are the throughputs (requests/ms) of the two operations with the consistency protocol turned off. The CreateDir(on) and SetDirAttr(on) curves are the throughputs in the same benchmark but with the consistency protocol turned on.

on individual operations. With the protocol turned off, the coordinators of cross-island operations merely propagate updates to involved islands without guarantee of atomicity, serialization or recoverability. The protocol increases the RPCs between servers for cross-island operations by a factor of 2 for two-phase commit and requires a log write per successful cross-island operation. Figure 4 shows the bandwidth in WriteFile and ReadFile and the response times in other operations, all measured at the client side. As expected, the consistency protocol adds considerable overhead to cross-island operations (CreateDir, SetDirAttr and RemoveDir), but does not have a significant impact on one-island operations.

9.2 Scalability of cross-island operations

We run the same micro benchmarks with 1 to 16 servers and clients. We turn on and off the consistency protocol to measure its impact on the scalability of individual operations. Figure 5 shows the throughputs of two cross-island operations, CreateDir and SetDirAttr. (RemoveDir is similar to CreateDir.) The throughput of CreateDir scales at roughly the same rate with or without the protocol because each CreateDir operation involves a constant number (two) of islands in Archipelago. The throughput of SetDirAttr does not scale in either case because each operation involves all islands. The protocol does not have a noticeable impact on one-island operations, which are not shown here. Therefore, the overall scalability depends on the actual operation breakdown in the system.

9.3 Scalability of operation mixes

We run a benchmark of randomized operation mixes with the consistency protocol turned on to measure the overall scalability of Archipelago. The benchmark is extended from the SPEC SFS or LADDIS benchmark [18]. Since Archipelago is implemented on top of NTFS, the operation mix in our benchmark uses NTFS API and is based on the traces we took on Windows NT workstations [17].

We run the benchmark with 1 to 16 clients and servers. The pre-created data set includes 2000 directories, 2000 files, and 100 symbolic links shared by all clients, and the same numbers of private objects (directories, files and symbolic links) per client. The client repeatedly does an operation that is randomly chosen at specified frequencies. For each operation, the client randomly chooses an object, either from the existing shared or private objects, or by generating a new name in an existing directory, depending on the operation. The WriteFile operation writes a random number (chosen from 0 to 1 MB) of bytes to the file; both WriteFile and ReadFile operations transfer up to 8KB per request so that the operation time is comparable to those of other operations. Each client maintains its own view of the shared objects and its private objects, but does not synchronize with other clients on the creation and deletion of the shared objects. Therefore, an operation on a shared object might fail if it conflicts with a previous operation on the same object from another client [18]. After the data set is pre-created, all clients run the randomized operation mix for 10 minutes. The throughput is calculated as the total number of successful operations by all clients divided by 10 minutes.

We run the benchmark with two different operation mixes. Mix 1 exaggerates the cross-island operations and mix 2 is closer to the measured breakdown. We record the actual client operations and server-to-server RPCs in the benchmarks, and estimated the speedups of the overall operation mix accordingly. Table 3 shows the recorded operation mixes and Figure 6 shows both the measured speedups and estimated speedups. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup with n servers is $\frac{n}{1+OverheadPerOperation}$, where $OverheadPerOperation$ is the total number of server-to-server RPCs divided by the total number of successful client operations.

Operation mix 1 scales at a less than ideal slope due to the relatively large number of cross-island operations. For example, with 16 servers, the average overhead per operation is 0.8. The difference between the estimated speedup and measured speedup is due to the assumption of equal RPC processing times and local operation times. Operation mix 2 is closer to the measured breakdown, i.e. contains a smaller number of cross-island operations;

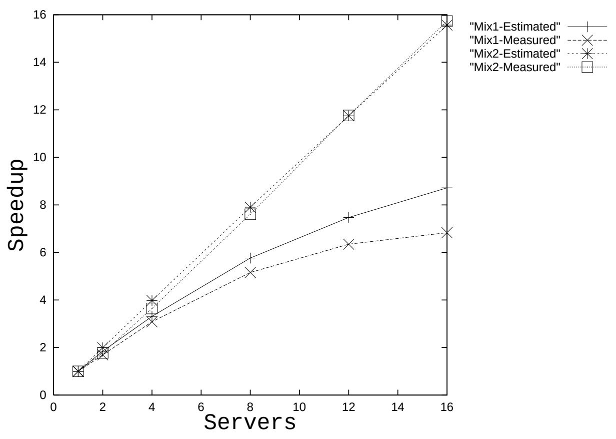


Figure 6: Speedup of throughputs of randomized operation mixes. The four curves are the measured speedup of operation mix 1, estimated speedup of operation mix 1, measured speedup of operation mix, and estimated speedup of operation mix 2, respectively. The speedup is calculated as the absolute throughput (requests/sec) divided by the throughput of 1 server. The throughput of 1 server is 75.6 requests/sec in operation mix 1 and 80.1 requests/sec in operation mix 2, respectively.

	Mix 1 (%)	Mix 2 (%)
CreateDir	0.9297	0.0522
CreateFile	4.0314	3.5661
DeleteFile	2.7731	2.4353
DeleteLinkDir	0.985	0.0128
ReadDir	14.4505	15.6528
ReadFile	14.1343	15.2778
RemoveDir	0.7543	0.0162
ResolveLinkDir	1.7205	0.1014
SetDirAttr	1.0383	0.0713
SetFileAttr	26.6085	29.2835
SymLinkDir	1.0089	0.0109
WriteFile	31.5656	33.5194
Successful	45360 to 309960	48042 to 756120
Total	48042 to 325534	48043 to 780260
Throughput (requests/sec)	75.6 to 516.6	80.07 to 1260.2

Table 3: Operation mixes. Each percentage in this table is the number of successful requests on each operation divided by the total number of successful requests, averaged over 1 to 16 clients and servers. The total numbers of requests and throughputs grow with the numbers of clients and servers for the fixed 10 minutes period; the ranges are shown in the last three rows in the table.

it scales nearly ideally in both estimated and measured throughputs. For example, it reaches a speedup of 15.7 on 16 islands.

10 Conclusions

We have designed and implemented a protocol for the atomicity, serialization and recovery of updates on replicated directory data in the island-based file system and the clustered in-memory database. We build a randomized test engine to check the correctness of the protocol in the face of arbitrary sequences of failures. The impact of the consistency protocol on the performance and scalability of the system is studied in micro benchmarks and trace-based operation mixes.

The protocol has little impact on common cases or local operations since all operations that read replicated data or read/write non-replicated data can be processed in a single replica without communication to others. Under this protocol, the replicas never expose the intermediate state of updates to clients and clients never observe the results of updates in conflicting orders; therefore, the chance for hazards introduced by the cluster environment is largely reduced, and it is possible to port applications from single systems to the cluster-based systems with few modifications.

We conclude that it is possible to distribute data in a cluster under such a protocol that the system can both achieve high availability and strong consistency, and scale efficiently with the cluster size.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems and Principles*, December 1995.

[2] Anonymous authors. *The Common Internet File System (CIFS) Specification Reference*. Microsoft, 1996.

[3] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proceedings of ACM SIGMOD*, 1999.

[4] P. Chundi, D. J. Rosenkratz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of 12th International Conference on Data Engineering*, 1996.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings ACM Symposium on Principles of Programming Languages*, January 1992.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[7] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems and Principles*, December 1995.

[8] Suppressed for anonymous review. Affinity-based management of clustered in-memory databases for application servers. *Submitted for publication*, March 2001.

[9] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Octobor 1997.

[10] H. Garcia-Molina and K. Salem. Main memory database systems. In *IEEE Transactions on Knowledge and Data Engineering*, December 1992.

[11] R. Golding, J. Wilkes, and A. Veitch. Private communications, August 1999.

[12] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, 1978.

[13] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, November 1987.

[14] Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the echo distributed file system. In *Proceedings of Workshop on Management of Replicated Data*, November 1990.

[15] http://www.apache.org. Apache web server.

[16] http://www.mysql.com. Mysql database server.

[17] M. Ji, E. W. Felten, R. Wang, and J. P. Singh. Archipelago: An island-based file system for highly available and scalable internet services. In *Proceedings of 4th USENIX Windows Systems Symposium*, August 2000.

[18] B. E. Keith and M. Wittle. Laddis: the next generation in nfs file server benchmarking. In *Proceedings of USENIX Summer Technical Conference*, June 1993.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, July 1978.

[20] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[21] Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[22] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, Vol. 2 No. 3, September 1989.

[23] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, Vol. 12 No. 2, May 1994.

[24] G. J. Popek, R. G. Guy, T. W. Page Jr., and J. S. Heidemann. Replication in ficus distributed file systems. In *Proceedings of Workshop on the Management of Replicated Data*, November 1990.

[25] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. In *IEEE Transactions on Computers 39(4)*, April 1990.

[26] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Octobor 1997.

[27] P. Triantafillou and C. Neilson. Achieving strong consistency in a distributed file system. *IEEE Transactions on Software Engineering*, Vol. 23 No. 1, January 1997.

[28] Walker, G. Popek, R. English, C. Kline, and G. Thiel. The locus distributed operating system. In *Proceedings of 9th ACM Symposium on Operating Systems Principles*, October 1983.