Atomicity, Serialization And Recovery
In A Highly Available And Scalable Cluster-Based File System

Abstract

We discuss the consistency protocol in *island-based* file system, a cluster-based file system designed to provide highly available and scalable data storage and access to Internet applications. Our goal in maintaining the consistency is to minimize the efforts for porting applications from single systems to the cluster-based environment. The island-based design offers 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. We measure the impact of the consistency protocol on the performance and scalability of the 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 on 16 islands.

1. Introduction

Clusters are a cost-efficient platform for large applications such as file servers and web servers, because they offer two potential benefits over single large machines: high availability and incremental scalability [15]. High availability can be achieved by replicating data or functional components of a system across cluster nodes. The power and capacity of a cluster can be incrementally scaled by adding nodes to the cluster. The challenges of cluster computing, however, include supporting portability of applications from single systems to clusters and dealing with node failures and network partitions, as well as partitioning and replicating data and functionality in an effective way to achieve high availability and scalability.

In this paper, we consider the consistency maintenance of a large cluster-based file system, called *island-based* file system [16], as an approach to the portability of applications from single file systems to the cluster-based file system.

The island-based file system was designed to provide highly available and scalable data storage and access to Internet applications. The goal of availability is to maximize the percentage of client requests that succeed despite the failure of one or more servers in the cluster. Previous approaches in cluster-based data storage rely on redundancy to survive a small number of failures, but the system becomes largely unavailable if more failures occur. The island-based file system uses a failure isolation approach. It divides the nodes in the cluster into groups called *islands*. An island might be a single node, or a group of nodes that use redundancy within the island to mask failures. A self-contained, off-the-shelf and load-balanced file server runs in each island. The island-based design partitions files and directories and replicates certain parts of the directory system across islands in such a way that the server in each island can deliver data to clients independently of the failures in other islands. The failure-isolation approach is complementary to 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.

We evaluated the island-based design by statistical analysis of the access patterns of existing systems. The results show that the availability model in island-based file system is desirable to Internet applications because a temporary partial failure can be made unnoticeable to the majority of clients. We implemented a prototype island-based file system on a cluster of commodity PCs connected by Ethernet. Measurement with trace-based operation mixes shows that island-based system scales efficiently with cluster size.

A small portion of the directory system is replicated across islands to allow independent accesses to data in each island. As in any other systems where data replication and updates to replicated data are present, island-based file system faces the challenge of keeping its replicated directory system consistent across islands. Hazards could occur if the replication is not handled with care, because the structural integrity of the directory system is critical for the system to function correctly and to recover successfully from possible failures, and because directories tend to be shared by multiple clients more frequently than files.

The following are two examples of possible hazards in a general distributed file system where directories are replicated across servers and clients are allowed to access any replicas:

- 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 single system, only one, not both, of the operations would abort.

- 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 single 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 island-based file system 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 island-based file system.

The island-based design offers an opportunity for strong consistency without sacrificing performance in common cases. Since it strives to reduce shared state across islands for the purpose of failure isolation, the cost for maintaining consistency of shared state can potentially be reduced as well. Therefore, it is possible to achieve strong consistency for the small set of shared state while maintaining the overall performance and scalability of the system.

We discuss in this paper how to design a robust and efficient protocol for the synchronization of operations on replicated data in the face of node failures and network partitions. Before discussing our contributions in detail, we present a brief overview of prior work on replication and consistency issues in file systems and Internet applications.

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 schemes.

Wide-area distributed file systems such as Ficus [20], Coda [23] and Locus [22] 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 [18] and Echo [21] 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 [25], the overhead is reduced by distributing load across servers and amortizing the costs of individual operations with file sessions.

In recent cluster-based file systems like Frangipani [1] and xFS [2], 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 [3].

The Cluster-Based Scalable Network Services (SNS) [15] 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 [26] and database servers [27], on a cluster-based file system as well as on a single file system.

3. Our contributions

The context of system structure and replication scheme in which our consistency protocol is considered differs from the contexts in previous studies. In island-based file system, only certain directory attributes, but not the directory contents (the lists of names and addresses of sub directories and files) or files, are replicated across islands. The degree of replication varies by directories based on their usage, and changes dynamically as the

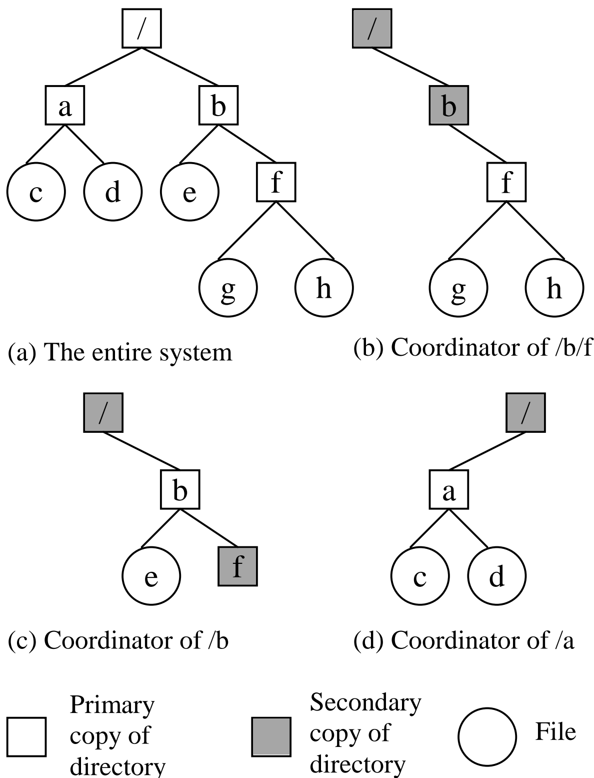

Figure 1. Replication of directories. Figure (a) is the image of an entire system. (b) (c) and (d) are the images of the internal file systems in three other islands. Shaded directories in the figure represent replicas that contain only attributes and partial contents or no contents.

usage changes.

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 strategy in the island-based design 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 the implemented prototype.

4. Overview of island-based design

In this section, we summarize the features in island-based file system that are relevant to the consistency protocol we present in the rest of the paper. A separate publication [16] gives details of the island-based design, implementation and evaluation.

The main idea underlying the island-based design is the *one-island principle*: as many file system operations as possible should require the participation of exactly one island. The one-island principle offers good failure isolation because each island can function independently of other islands' failures. It also allows island-based systems to scale efficiently with the system and workload sizes because communication and synchronization across islands are reduced.

The target application of island-based file system is the data storage for those Internet applications 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.

We designed a new data distribution strategy for island-based file system: data is distributed to islands at *directory granularity* by *hashing* the *pathnames* of the directories to island indices. We call the file system running inside each island 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, we store directories 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 we can leverage the built-in functions of the internal file systems.

The consequence of storing data in skeleton hierarchies is the replication of directories. To reduce shared state across islands, we replicate only directory attributes that are needed when a descendent of the directory is looked up. Such attributes include name, security, read-only tag and compressed tag. Updates to those attributes need to be propagated to all replicas. The overhead of updates is acceptable since those attributes rarely change [16]. 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 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*.

In the rest of the paper, we discuss the consistency protocol for these cross-island operations in the face of partial failures.

5. Replication model

We define a few terms below to assist in generalizing the replication model in island-based file system. For each replicated *object* (a set of attributes of a directory), a particular island is chosen as the *coordinator* of the cross-island operations 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 or *replicas*. Figure 1 illustrates the replication in island-based file system. Each file system operation originates in a single island. Each cross-island operation on an object must originate in its coordinator and be propagated to other islands (called *involved* islands) that have a replica of the object. All objects in an island are readable by operations originated in or propagated to this island. However, primary copies of objects in an island can be updated only by the operations that originate in this island, and secondary copies of objects in an island can be updated only by the operations that are propagated to this island from the coordinator.

The island-based design favors the consistency maintenance in the following ways:

- The majority of operations are one-island operations and hence are guaranteed to be atomic and serialized by the internal file systems.
- All cross-island operations on the same object are coordinated by a single island, hence synchronization can be done with centralized control per object, which eases the protocol design.
- The single coordinator property ensures that no conflicting updates will occur even in case of network partitions.

It is worth noting that the replication model adopted in island-based file system is similar to the models in typical replicated databases [17] [18]. However, our consistency requirement differs, primarily because we do not require transactional semantics for file accesses. Therefore, we believe that an approach to the consistency of island-based file system will be valuable to other file systems in this model as well.

6. Consistency protocol design

In this section, we present the consistency protocol in island-based file system.

A strawman approach to the consistency of replicated directories across islands 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 [24], majority consensus [1] and/or versioning [8].

Unfortunately, such a scheme can seriously weaken the availability and scalability offered by the island-based design. 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. In consequence, there will no longer be "one-island" operations in island-based file system since almost every operation needs to contact multiple islands for locking involved directories. 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 [5], two-phase commit [6], logging [7] and finite-state-machine-based recovery to serialize the cross-island operations while keeping the synchronization for one-island operations local. Our methodology takes three steps. First, we guarantee that each cross-island operation is atomic; second, we serialize cross-island and one-island operations in common cases; third, we ensure the serialization of cross-island operations during a recovery from failures.

6.1 Atomicity

The basic consistency guarantee our protocol offers is the *atomicity* of the cross-island operations, i.e. clients would never observe the intermediate state of any operation. In other words, once a client observes the result of a cross-island operation in an island, it would always observe the result of that operation in other involved islands afterwards.

We use a vector of logical clocks for the atomicity of cross-island operations. Each island has its *local* logical clock and each cross-island operation coordinated by this island increases the clock by 1, or *generates* a new clock value. Each island or client maintains a vector of all islands' clocks. Each request to an island carries the sender's current clock vector for synchronization with the receiver's vector before the request is processed, and

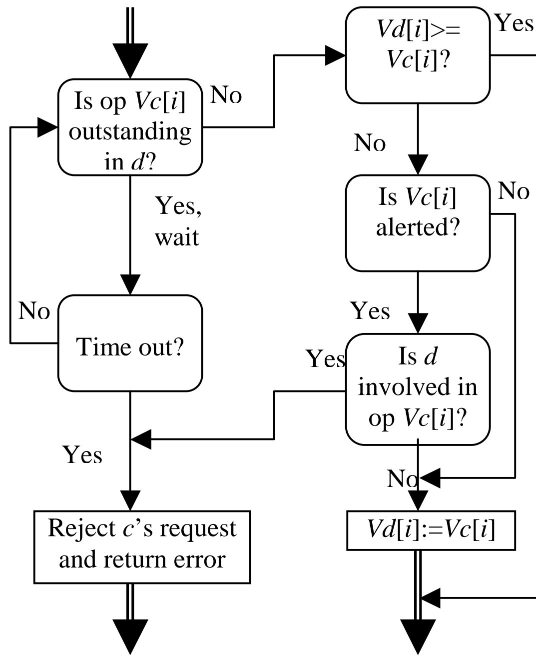

Figure 2. Synchronization of a client *c*'s clock *Vc*[*i*] with the island *d*'s clock *Vd*[*i*]. Op *Vc*[*i*] is the cross-island operation that generated the clock value *Vc*[*i*] in its coordinator, island *i*.

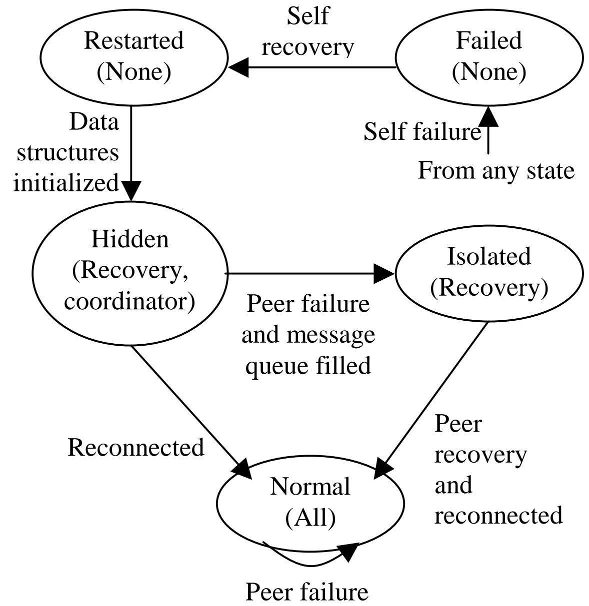

Figure 3. State transitions of an island in response to various failures and recoveries. The types of requests accepted in each state are listed in parenthesis. Each transition is labeled with the event that triggers the transition. "Reconnected" is the event that the recovering island has reconnected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

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>=*V*1, if and only if *V*2[*i*]>=*V*1[*i*], 0<=*i*<*n*, where *n* is the number of islands.

We maintain the following invariants:

1) The local commit of a cross-island operation and the update of the local clock are atomic in each island, which is guaranteed with a local lock in that island.
2) A coordinator does not release the new clock value to a client until it has notified all involved islands of the operation, i.e. until the operation is either *outstanding* or *committed* in all involved islands. This is guaranteed with a two-phase commit [6]: the coordinator notifies all involved islands of the operation in phase 1, then locally commits the operation and updates the clock, and asks involved islands to commit the operation in phase 2.
3) A request cannot be processed in an island if the request carries a clock that is generated by an outstanding operation in that island. Based on invariants 1 and 2, this invariant means that once a client observes the result of an operation in at least one island, it will always observe the result of that operation in other involved islands afterwards. This is guaranteed by the clock synchronization algorithm in Figure 2, which is an extension to Lamport's algorithm [5].

The three invariants above guarantee that an island will never expose the intermediate state of any operation to clients. Invariant 2 ensures that synchronization on 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 involved island is inaccessible due to either an island 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, an island 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 an involved island after phase 1, the operation will be outstanding in the involved island till the coordinator reconnects. This type of failure will be detected by a timeout in the clock synchronization in the involved server. See Figure 2. The alerted bit will be cleared once the islands 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 cross-island operations, i.e. clients observe the results of all operations in the same order in all islands. All the cross-island operations on the same object are coordinated by the same island, hence can be serialized by a local mutex in that island, unless an involved island failed.

The serialization in case of failures is guaranteed by *write-ahead logging* [7]. The coordinator always writes a record with its clock vector to stable storage before it locally commits a cross-island operation. Only after the operation is committed in all involved islands, the record can be removed from the log.

When an island 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 one-island 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 an involved island 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 islands and clients never decrease and always increase upon cross-island operations, even with network partitions. Therefore, recovering islands 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 islands in the ascending order of their clock vectors.

If two clients interact with each other by accessing the same objects in the file system, then the operations by the two clients are serializable in the face of failures. For example, if two clients, $c1$ and $c2$, access the same object at time $t1$ and $t2$ ($t1<t2$) and receive the clock vectors $V1$ and $V2$ respectively, then $V1<=V2$ because the vectors are issued by the same island; therefore, $c1$'s operations before $t1$ (with vectors $<V1$) and $c2$'s operations after $t2$ (with vectors $>V2$) are serializable.

Clients that do not interact through accesses in the file system might have *concurrent* clock vectors. We say two vectors $V1$ and $V2$ are concurrent if and only if there exist i and j, $i!=j$ and $0<=i,j<n$, such that $V1[i]<V2[i]$ and $V1[j]>V2[j]$, where n is the number of islands. During a failure recovery, concurrent vectors will be sorted with a simple tie resolution rule consistent across all islands, 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 file system was concerned [5].

6.3 Recovery

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

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 island.

Given the finite set of possible failures and the infinite set of possible sequences of the failures, we find it a good practice to model the recovering island as a finite state machine, in which each state corresponds to a set of behaviors that are allowed in the recovering island, and each state transition is triggered by a failure or recovery event. Figure 3 shows the state transitions of an island in response to the possible failures and recoveries. An island 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 island is allowed to process in that state. The types of requests an island receives include *client* requests (from the clients), *coordinator* requests (from the coordinators of cross-island operations), *recovery* requests (from the recovering or reconnecting islands), etc.

In the *normal* state, an island processes all requests. A self failure in any state causes the island to transit to the *failed* state, in which no requests, of course, are processed. When it is recovered, an island 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 islands and to synchronize replicated state with other islands by log exchanges. In the hidden state, the island rejects all client requests so that inconsistency, if it is present in the island, is not visible to clients. The island accepts requests from other recovering or reconnecting islands so that both can make

progress. It also accepts requests from the coordinators of new cross-island operations and stores them in a *message queue* for sorting with other operations when all have arrived. If the queue becomes full, the island 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 island in the buffer.)

When all other islands have reconnected and exchanged logs with it, the island 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 other islands that it has rejected. After it commits all involving operations, it transits to the normal state.

6.4 Summary

Our consistency protocol guarantees the following serializations for cross-island operations in the face of arbitrary sequences of node failures and network partitions:

1) All operations on the same object are serializable.
2) All operations by the same client thread are serializable.
3) Operations by different clients are serializable if the clients interact with each other by accessing the same object(s) in the file system.

In addition, the ordering relations of operations are *transitive*, i.e. if operation 1 is observed to happen before 2 and 2 before 3 then 1 is observed to happen before 3, because the ordering relations of clock vectors are transitive, i.e. if $V1<V2$ and $V2<V3$ then $V1<V3$.

The protocol has little impact on common cases or one-island operations since all operations that read replicated data or read/write non-replicated data can be processed in a single island without communication to others.

Under this protocol, island-based file system never exposes the intermediate state of cross-island operations to clients and clients never observe the results of cross-island operations 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 file systems to island-based file system with few modifications.

7. Implementation

We have implemented the consistency protocol as a library in a prototype of island-based file system called *Archipelago*. Archipelago [16] was implemented on a cluster of Pentium II PCs running Windows NT 4.0. NTFS [9] is used as the internal file system. 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. The library consists of approximately 1800 lines of C++ code.

8. Correctness testing

The basic algorithms in our consistency protocol, i.e. logical clock synchronization [5], two-phase commit [6] and logging [7], have been widely used in existing systems. The correctness of our system relies mostly on the details in implementation, which are hard to model or check using existing tools [12] [14]. Therefore, we do not attempt to theoretically prove the correctness of our consistency protocol. Instead, we use a randomized test engine to test the correctness of Archipelago in the face of failures. The test engine is extended from a model checker originally developed in Hewlett-Packard Labs [13]; the model checker is based on the input/output automata (IOA) [11]. 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 [12] [14], 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.

The test engine consists of three components, *terminators*, *network partitioner* and *clients*. The terminators are independent threads or processes, one for each island. Each terminator injects *crash* or *reboot* events to its associated island at intervals randomly chosen within given ranges. It simulates a crash of the island by killing the server process of that island, and the reboot of the island by forking a new server process for that island. The network partitioner is an independent thread that simulates network partitions between islands. At random intervals, it randomly chooses a pair of islands and sends a message to both islands 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 [11]. 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 island at any given time; there is no gap and no overlap in the clocks of the operations coordinated by the same island; the island i always has a more or equally up-to-date clock $V[i]$ than any other islands or clients; etc.. These assertions have been surprisingly helpful in our preliminary experiments. Liveness assertions such as that an island 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.

Events	Parameters (% or seconds)	Numbers of Events
CreateDir	3.2279 %	1565
CreateFile	2.8244 %	1369
DeleteFile	1.9206 %	974
DeleteLinkDir	0.8070 %	221
ReadDir	11.2169 %	5273
ReadFile	13.1536 %	8162
RemoveDir	2.4209 %	1469
ResolveLinkDir	7.3434 %	530
SetDirAttr	5.6488 %	2609
SetFileAttr	21.9819 %	14970
SymLinkDir	0.8070 %	227
WriteFile	28.6475 %	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.

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 cross-island operations 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 a cross-island operation crashed after it notified the involved islands of the operation, but before it logged the operation on disk. Therefore, the operation was aborted in the coordinator, but outstanding in the involved islands. When the involved islands received the next operation from the same coordinator later, the assertion of at most one outstanding operation per island failed. The fix to this bug is to clear the relevant buffers of outstanding operations upon reconnection of two islands.

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 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

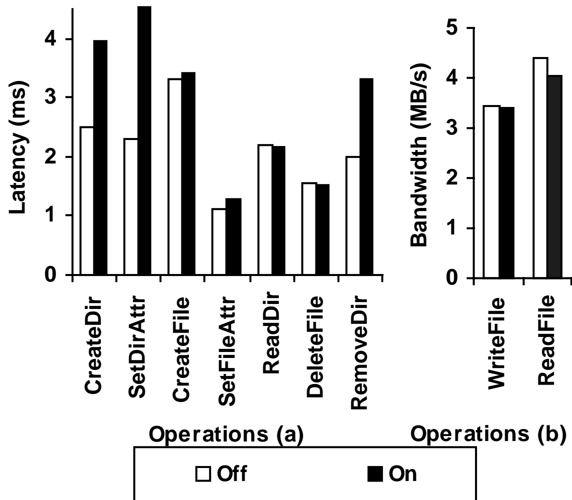

Figure 4. Single client performance. A single client runs the micro benchmarks with the consistency protocol turned on and off, respectively. The y-axis in (a) is the latency in milliseconds measured at the client side. Lower columns represent better performance. The y-axis in (b) is the bandwidth in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

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 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 [10]. Therefore, the overall scalability depends on the actual operation breakdown in the system.

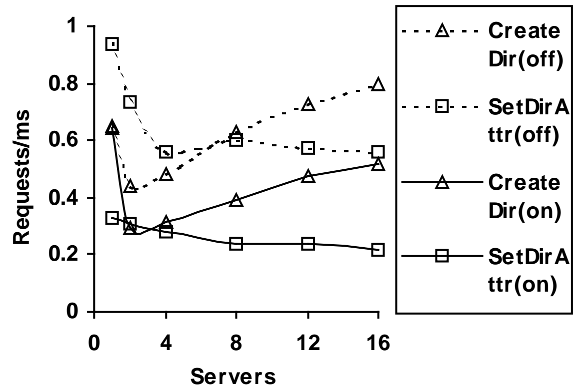

Figure 5. Impact of the consistency protocol on cross-island operations. The CreateDir(off) and SetDirAttr(off) curves are the throughputs (requests/ms) of the two operations with the consistency protocol turned off. The CreateDir(on) and SetDirAttr(on) curves are the throughputs in the same benchmark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

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 [4]. 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 [16].

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

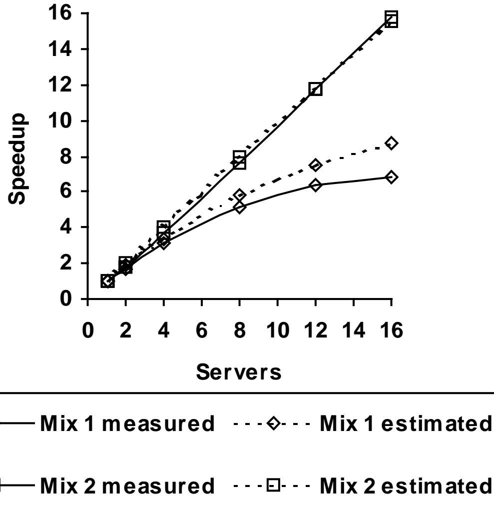

Figure 6. Speedup of throughputs of randomized operation mixes. The four curves are the measured speedup of operation mix 1, estimated speedup of operation mix 1, measured speedup of operation mix, and estimated speedup of operation mix 2, respectively. The speedup is calculated as the absolute throughput (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.

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 [4]. 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 $n/(1+overhead_per_operation)$, where the *overhead per operation* is the total number of server-to-server RPCs divided by the total number of successful client operations.

	Mix 1 (%)	Mix 2 (%)
CreateDir	0.9297	0.0522
CreateFile	4.0314	3.5661
DeleteFile	2.7731	2.4353
DeleteLinkDir	0.9850	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.

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

10. Conclusion

Clusters are a fact of life -- people are already using them to provide scalable services. An important question for people who are building cluster-based services to understand is how to design software that supports painless porting of applications from single systems to clusters, without weakening the availability and scalability offered by the cluster structure. Our experience suggests that this is indeed possible.

We design and implement a protocol for the atomicity, serialization and recovery of cross-island operations in the face of arbitrary sequences of failures in island-based

file system. We build a randomized test engine to check the correctness of the protocol, and study the impact of the consistency protocol on the performance and scalability of the system in micro benchmarks and trace-based operation mixes. We conclude that it is possible to distribute data in a cluster-based file system under such protocols that the system can both achieve good failure isolation and strong consistency, and scale efficiently with the cluster size.

References

[1] 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.

[2] 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.

[3] 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.

[4] B. E. Keith, and M. Wittle, "LADDIS: the Next Generation in NFS File Server Benchmarking", in Proceedings of USENIX Summer Technical Conference, June 1993.

[5] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, in Communications of the ACM, July 1978.

[6] J. Gray, “Notes on Database Operating Systems”, in Operating Systems: An Advanced Course, 1978.

[7] 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.

[8] B. Gronvall, A. Westerlund, and S. Pink, "The Design of a Multicast-based Distributed File System", in Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999.

[9] H. Custer, “Inside the Windows NT File System”, Microsoft Press, 1994.

[10] M. Ji, and E. W. Felten, “Design and Implementation of an Island-Based File System”, Technical Report 610-99, Department of Computer Science, October 1999.

[11] N. Lynch, and M. Tuttle, “An Introduction to Input/Output Automata”, CWI-Quarterly, 2(3), September 1989.

[12] 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.

[13] R. Golding, J. Wilkes, and A. Veitch, private communications, August 1999.

[14] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction”, in Proceedings ACM Symposium on Principles of Programming Languages, January 1992.

[15] 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.

[16] M. Ji, E. W. Felten, R. Wang, and J. P. Singh, "Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services", to appear in Proceedings of 4th USENIX Windows Systems Symposium, August 2000.

[17] 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.

[18] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, "Update Propagation Protocols for Replicated Databases", in Proceedings of ACM SIGMOD, 1999.

[19] B. Liskov, S. Chemawat, R. Gruber, P. Johnson, L. Shrira and M. Williams, “Replication in the Harp file system”, in Proceedings of the 1991 Symposium on Operating System Principles, Oct. 1991.

[20] G. J. Popek, R. G. Guy, T. W. Page Jr., J. S. Heidemann: "Replication in Ficus Distributed File Systems", in Proceedings of Workshop on the Management of Replicated Data, November 1990.

[21] A. 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.

[22] B. 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.

[23] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", in IEEE Transactions on Computers 39(4), April 1990.

[24] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart, “A Coherent Distributed File Cache with Directory Write-Behind”, in ACM Transactions on Computer Systems, Vol. 12, No. 2, May 1994.

[25] P. Triantafillou and C. Neilson, "Achieving Strong Consistency in a Distributed File System", in IEEE Transactions on Software Engineering, Vol. 23, No. 1, January 1997.

[26] Apache Web Server, http://www.apache.org

[27] MySQL Database Server, http://www.mysql.com