Design And Implementation Of An Island-Based File System

Minwen Ji and Edward W. Felten
Department of Computer Science, Princeton University
{mji, felten}@cs.princeton.edu

Abstract

Reliability, availability and scalability are major concerns in the design of distributed file systems. We have built an *island-based* file system (*IFS*) called *Archipelago* to solve these problems by failure isolation and low-cost consistency maintenance. The building blocks of IFS are smaller self-contained file servers called *islands*. The main idea underlying island-based design is the *one-island principle*: as many operations as possible should involve exactly one island. The one-island principle improves partial reliability and availability because each island can function independently of other islands' failures. It allows IFS to scale efficiently with the system and workload sizes because consistency across islands can be maintained at a low cost. The data distribution strategies in existing file systems cannot satisfy the one-island principle without sacrificing load balance and scalability. We designed a new strategy in which data is distributed to islands at directory granularity by hashing the pathnames of directories. Certain metadata is replicated across islands in such a way that islands are self-contained and the cost for maintaining consistency across replicas is minimized.

We evaluated the data distribution strategy in IFS by statistical analysis of the access patterns and contents of existing file systems in use. We studied partial availability, load balance, replication cost and consistency cost in web access logs, UNIX file system call traces, snapshots of file system contents, and Windows NT file access traces. The results confirmed the assumptions we made in the design. In addition, we compared data loss of IFS and typical non-IFS in case of partial failures in analytic models. The IFS model has a significantly lower data loss ratio than non-IFS, at the cost of replicating a small amount of metadata.

We designed three protocols in Archipelago, the rebalance, consistency and recovery protocols, to make the island-based design a viable solution. We have implemented Archipelago on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The consistency and recovery protocols are tested with randomized failure injections. The performance measured in micro benchmarks and operation mixes shows little overhead of the consistency protocol on one-island operations; in one case, the speedup with 16 islands achieves 98.3% efficiency. A trace-driven study of the online reconfiguration of a web server running on Archipelago shows that data migration in the rebalance protocol is made transparent to the web server and imposes a performance penalty of only 4.5%.

1. Introduction

Reliability, availability and scalability are major concerns in the design of distributed file systems. An increasingly popular class of applications, the Internet servers, requires improved *partial* reliability and availability as opposed to the traditional *all-or-nothing* mode. Typical Internet servers, e.g. web servers, cache servers, email servers and news servers, serve a large number of independent clients who access a relatively small portion of the entire contents individually. In case of partial failures, those servers will prefer to remain available to as many clients as possible, rather than to go offline as a whole. This access pattern also implies that the goal of scaling the servers is to meet the needs of increased number of workloads or clients, rather than to improve the performance for individual clients. The large scale of these applications, typically tens to hundreds of PC's per site, requires that the overhead for maintaining shared state across loosely-coupled machines be kept low. Those servers need to be dynamically reconfigured to adapt to the changing requirements of workloads, and the reconfiguration needs to be made transparent to clients in terms of both functionality and performance. Locality and load balance are shown to be two important but conflicting issues in those servers [3].

The state-of-the-art file systems, i.e. cluster file systems built on top of shared virtual disks [1][5], use data redundancy in the virtual storage layer for high reliability, and distributed lock management for consistency across replicas. To our knowledge, no existing redundancy scheme can prevent a virtual storage server from failing with arbitrary multiple physical failures; those systems do not address the damage control in case of such a failure. Distributed lock management introduces considerable communication and synchronization overhead for certain access patterns. Those systems cannot provide locality due to the transparency of the storage layer. On the contrary, the traditional mounted file systems [9] [8] have independent local file servers as building blocks, hence provide failure isolation and locality, and require little consistency maintenance across individual servers. However, they cannot scale well due to the manual partition of data and load imbalance across servers.

We designed an i*sland-based* file system (*IFS*) called *Archipelago* to solve these problems by failure isolation and low-cost consistency maintenance. The building

blocks of IFS are smaller self-contained file servers called *islands*. An island is *self-contained* in the sense that it contains all the metadata and functions it needs to access the data stored in it. The main idea underlying island-based design is *the one-island principle*: as many atomic operations as possible should require the participation of exactly one island. By *atomic* operation we mean a basic file system interface function such as creating a file or reading file attributes. The one-island principle promises the following benefits:

- **Failure isolation**: The one-island principle improves partial reliability and availability by failure isolation because each island is self-contained and hence can function independently of other islands' failures. In other words, the failure of 1 out of n islands in IFS renders only $\frac{1}{n}$ data inaccessible. In non-self-contained systems, the data in a surviving server will be inaccessible if any server containing a piece of metadata needed to access the surviving data fails.
- **Low consistency cost**: The one-island principle allows IFS to scale efficiently with the system and workload sizes because consistency across islands can be maintained at a low cost.
- **Low reconfiguration cost**: Reconfiguration (addition or removal of islands, or dynamic load balancing) requires a minimal amount of data to be migrated between islands, rather than a full rearrangement of all data. Therefore, reconfiguration has little impact on client performance and can be made transparent to clients.

However, the one-island principle has several challenges. Certain state needs to be replicated across islands for them to function independently of each other's failure. The storage required for such replication should be kept low. Cross-island communication and synchronization is occasionally necessary to keep replicated state consistent, and should be minimized. An efficient algorithm is needed to determine which island a client should contact for each operation. It is undesirable to have a client visit multiple islands, e.g. for recursive name lookups, because this violates the one-island principle. Load needs to be balanced across islands, which distinguishes IFS from a mounted file system.

We designed a new data distribution strategy for IFS: data is distributed to islands at *directory granularity* (rather than block, file or sub tree granularity) by *hashing* the *pathnames* of the directories to island indices. Directory granularity is fine enough to allow load balance; most file system operations involve a single directory and hence satisfy the one-island principle. A hash function can be computed on the client machines without contacting any servers and pathnames are the only information that a client can possibly have without contacting any servers. A hash function inherently provides locality because it has a consistent mapping from directories to islands as far as the function itself does not change. We use a combination of universal hashing [10] and extendible hashing [11] to achieve load balance and low reconfiguration cost, i.e. locality can be traded for load balance at a low cost when necessary. 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 mounted file system, a replicated file system or a cluster 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, islands can function independently of others' failures and we can leverage the functions of the internal file systems. The consequence of storing data in skeleton hierarchies is the replication of the attributes of ancestor directories that will be needed when a descendent is being looked up.

We evaluated the data distribution strategy in IFS by statistical analysis of the access patterns and contents of existing file systems in use. In particular, we studied partial availability, load balance, replication cost and consistency cost in web access logs, UNIX file system call traces, snapshots of file system contents, and Windows NT file access traces. The results show that the majority of web clients access only 1 to 2 distinct directories; therefore, they are likely to survive a temporary partial failure in IFS in spite of the fact that a partial failure causes a random set of directories to be inaccessible. The storage needed for replicating the attributes of ancestor directories accounts for 0.3% to 7.7% of total storage. Load imbalance (average load per island divided by standard deviation of load) resulted from the hashing algorithm in IFS is 0.0001 to 0.0279. On average only 0.2% operations involve multiple islands (we call them *cross-island* operations) and need a consistency protocol. We also compared in analytic models the data loss in IFS in case of partial failures to that of cluster file systems (CFS's) built on top of shared virtual disks [20][1][5]. The IFS model has a significantly lower data loss ratio than CFS's under various comparable redundancy schemes, e.g. 20.4 times lower with 32 non-redundant storage servers, at the cost of replicating the attributes of ancestor directories.

We designed three protocols in Archipelago, the rebalance, consistency and recovery protocols, to make the island-based design a viable solution. A rebalance protocol is used for fast, fault-tolerant and transparent reconfiguration of the system, i.e. addition or removal of islands, or dynamic load balancing. A consistency protocol is used for the atomicity and serialization of cross-island operations in the face of failures. A recovery protocol is used for islands to recover from various

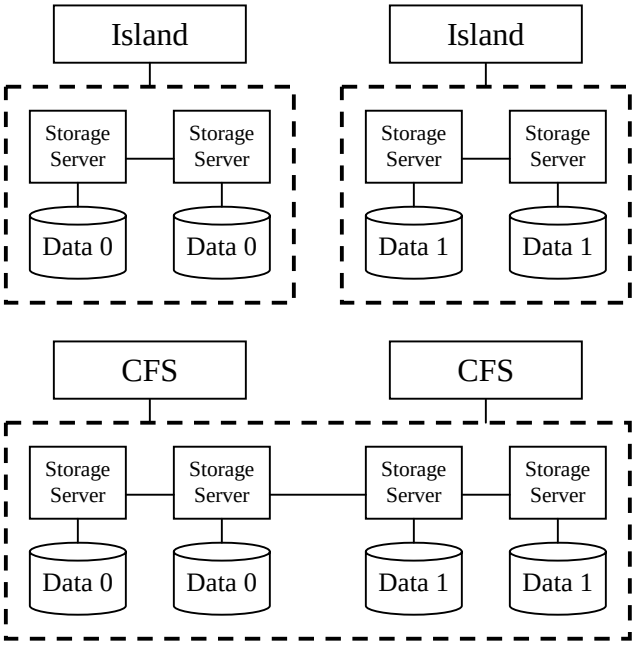


Figure 1. Mirrored IFS and CFS configurations in data loss models. Storages with the same labels contain the replicas of the same data.

combinations of failures back to consistent states.

We have implemented Archipelago on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The consistency and recovery protocols are tested with randomized failure injections. The performance measured in micro benchmarks and operation mixes shows little overhead of the consistency protocol on one-island operations; in one case, the speedup with 16 islands achieves 98.3% efficiency. A trace-driven study of the online reconfiguration of a web server running on Archipelago shows that data migration in the rebalance protocol is made transparent to the web server and imposes a performance penalty of only 4.5%.

The rest of the paper is organized as follows. Section 2 analyzes data loss models. Section 3 gives the basic system design. Section 4 discusses replication cost. Sections 5 describes hashing algorithms for load balance. Section 6 presents statistical results. Sections 7, 8 and 9 describe in details the rebalance protocol, consistency protocol and recovery protocol, respectively. Section 10 discusses other design issues. Section 11 reports the implementation status. Section 12 discusses the correctness testing. Section 13 and 14 present the performance measurements in micro benchmarks, operation mixes and the trace-driven study of an online reconfiguration. Section 15 references related work. Section 16 draws conclusions and proposes future work.

2. Analytical models for data loss

In this section, we shall compare the permanent or temporary data loss in IFS in case of partial failures to that of a typical class of non-IFS, cluster file systems (CFS) built on top of shared virtual disks [20][1][5]. We model the data loss due to independent storage server failures in comparable configurations of IFS and CFS, see Figure 1 for an example.

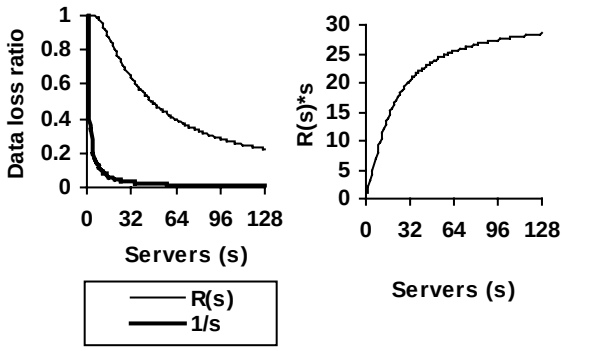


Figure 2. Data loss ratios in non-redundant IFS (1/s) and CFS (R(s)) with the loss of 1 out of *s* servers, and the ratio of R(s) to 1/*s*.

We make the following assumptions in the analytic models. Data is randomly distributed across multiple storage servers at block granularity in CFS since physical block location in the shared virtual disks is transparent to CFS. Data is randomly distributed across islands at directory granularity in IFS. The CFS model does not replicate ancestor directories; therefore, the inaccessibility of a directory implies the inaccessibility of all its descendents. We also assume whole file accesses, i.e. the inaccessibility of a part of a file causes the whole file to be counted as lost. We assume that the directory hierarchy is a complete tree of height *h*. This is a conservative assumption because as the hierarchy gets more irregular, more files will have longer pathnames and hence have more chances to be inaccessible in CFS. Each directory has *d* sub directories and *f* files, and the directory itself has a fixed size equal to the block size *bs*, hence fits in a single server. Each file also has a fixed size *fs*. We ignore the impact of lost inodes in CFS, i.e. we assume that they are replicated everywhere. In a model with *s* storage servers, there are *s* such trees, and the root of each tree is a sub directory of the root in the entire system.

We compare the data loss ratios in IFS and CFS under various redundancy schemes, which are based on the non-redundant model below.

2.1 Non-redundant model

In this model, each island in IFS runs on a single storage server. With the failure of 1 out of *s* servers, non-redundant IFS permanently or temporarily loses $\frac{1}{s}$ data,

according to the self-contained property of islands.

We compute the data loss ratio with the failure of 1 out of *s* storage servers in non-redundant CFS as follows. The amount of data in a tree of height *i* (a tree with a single node is of the height 0) is

$$T(i) = \frac{d^{i+1}-1}{d-1} \cdot bs + \frac{d^{i}-1}{d-1} \cdot f \cdot fs .$$

The expected probability of a file being inaccessible is

$$F = 1-(1-\frac{1}{s})^{\left\lceil \frac{fs}{bs} \right\rceil}.$$

The expected amount of data loss in the tree of height *i* is

$$L(i) = \frac{1}{s} \cdot T(i) + (1-\frac{1}{s}) \cdot (d \cdot L(i-1) + f \cdot F \cdot fs) .$$

That is, if the root of a tree happens to be stored in the failed server (with the probability $\frac{1}{s}$), the whole tree will be inaccessible; otherwise, (with the probability $(1-\frac{1}{s})$) the amount of data loss will be the sum of the expected amount *L*(*i*-1) of data loss in each of the *d* sub trees plus the expected number $f \cdot F$ of lost files times the file size *fs*. Similarly, the amount of data loss in a system with *s* servers and *s* sub trees of height *h* in the root directory is

$$TL(s) = \frac{1}{s} \cdot s \cdot T(h) + (1-\frac{1}{s}) \cdot s \cdot L(h) .$$

The data loss ratio is $R(s) = \frac{TL(s)}{T(h) \cdot s}$. See Appendix A for the complete solution to *L(h)*. We choose a set of typical parameters based on previous studies of file system contents [44] [42]: *h=8, d=2.5, bs=4096, f=10, fs=98304*. That is, on average, each server stores 2542 directories and 10166 files, or about 1 GB data.

Figure 2 shows the data loss ratios in non-redundant IFS ($\frac{1}{s}$) and CFS (*R(s)*) with the failure of 1 out of *s* storage servers as a function of *s*, and the ratio of *R(s)* to $\frac{1}{s}$, i.e. $R(s) \cdot s$. IFS significantly reduces the data loss ratio at the cost of replicating ancestor directories.

We also analyzed the sensitivity of *R(S)* to other parameters within practical ranges; the results show that *R(S)* increases with *h* (height of the tree), *d* (number of sub directories per directory), *f* (number of files per directory) and *fs* (file size), and decreases with *bs* (block size). With the failures of *k* servers, IFS loses $\frac{k}{s}$ data and CFS loses $k \cdot R(s)$ data.

2.2 Redundancy schemes with grouping

Many existing redundant storage systems are divided into groups and data redundancy is applied within groups, but not across groups. It results either from the nature of the redundancy scheme, such as mirroring pairs, or from performance optimization, such as RAID-5 striping groups [5] [43]. A CFS running on a shared storage system with *s* redundancy groups can be compared to an IFS with *s* islands, each of which runs on a single redundancy group of the same scheme. See Figure 1 for a mirrored example. If we treat each group as a single server, we can use the non-redundant model to compute the data loss with the failure of a group in both systems. Since the mean time to failure of a group is reduced by the same factor in both systems, the ratio of data loss in CFS to data loss in IFS is still $R(s) \cdot s$.

2.3 Redundancy schemes without grouping

In general, IFS can achieve as high reliability as CFS with an arbitrary redundancy scheme by being configured as a single island with a storage system of the same redundancy scheme. The actual gain in reliability needs to be analyzed on a case-by-case basis. Below we compare the data loss in IFS running on mirrored storage with that of CFS running on shared chained-declustering storage [7].

In this model, each system has $2 \cdot s$ storage servers. In IFS, each of *s* islands runs on top of 2 mirrored servers; in CFS, the replica of the data in each server is evenly distributed to the other servers. With the failures of 2 out of $2 \cdot s$ servers, IFS loses $\frac{1}{s}$ data with the probability $\frac{1}{s-1}$ (if the 2 failed servers happen to be in the same island); CFS loses $\frac{1}{s^2}$ storage with the probability 1. Interpreting *R(s)* as the data loss ratio with the loss of $\frac{1}{s}$ storage, the expected data loss ratios of IFS and CFS are $\frac{1}{s-1} \cdot \frac{1}{s}$ and $R(s^2)$, respectively. It is worth noting that the data loss ratio of CFS running on mirrored storage is $\frac{1}{s-1} \cdot R(s)$ and $R(s^2) > \frac{1}{s-1} \cdot R(s)$. That is, chained-declustering has a higher expected data loss ratio than mirroring.

2.4 Partial availability for applications

The models in previous sections show that, in comparable redundancy schemes of IFS and CFS, with the failures of the same number of servers, IFS has a significantly lower data loss ratio than CFS, at the cost of replicating ancestor directories. In other words, if the data is permanently lost, IFS will cause a lower cost for reconstructing the data at application level or manually;

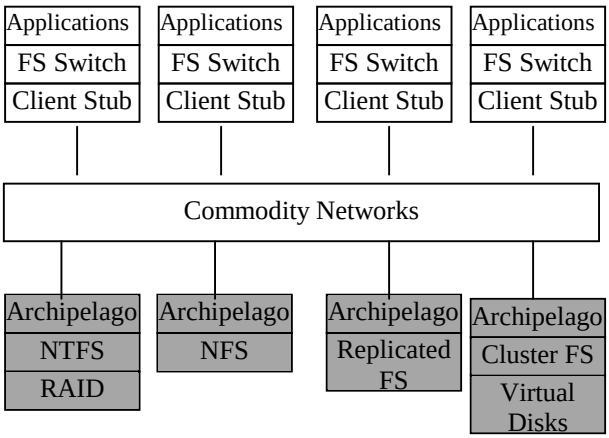


Figure 3. Overview of Archipelago. Shaded boxes are islands or servers and non-shaded boxes are clients.

if the data loss is only temporary, IFS maintains a higher availability.

If a client application needs to access multiple directories or files and any of the directories or files is lost, the application will fail as a whole. The availability of IFS with partial failures depends on the number *n* of distinct directories applications access. For example, with the failure of 1 out of *s* islands in non-redundant IFS, the expected probability that an application will *not* be affected is $(1-\frac{1}{s})^n$. The availability of CFS depends on the accessibility of the directories and files applications access and all their ancestor directories; therefore, the partial availability of CFS is always no higher than that of IFS.

We are going to collect the histograms of users and accesses by the numbers of distinct directories they involve in our statistical analysis. See Section 6.

3. System overview

Figure 3 gives an overview of Archipelago in a typical configuration. An island consists of a *server* process running on top of an internal file system. *Client* applications view the Archipelago as a single system and access it through local file system switches and *stubs*. Islands and clients are connected by commodity networks such as Ethernet and can be geographically distributed.

3.1 Directory granularity

Our first design decision concerns the granularity to use in data distribution. It should both allow load balance and satisfy the one-island principle

The obvious granularity choices are bytes, blocks, files, directories and sub trees. Although byte and block granularities are good for even distribution of data, they are not candidates for IFS because most file system operations involve multiple bytes or blocks, hence violate the one-island principle. Sub tree granularity, as used in mounted file systems, is self-contained and requires little state sharing across servers. However, sub trees cause load imbalance as some sub trees grow faster than others. Few existing systems distribute data at file granularity, but some distribute cached data or metadata at file granularity [3] [4] [19]. File granularity can potentially achieve better load balance than sub tree granularity because files are smaller than sub trees. Every file system operation that involves a single file satisfies the one-island principle. However, some frequent operations like "ls" or "dir" involve multiple files in a directory, which led us to choose the directory granularity instead.

3.2 Hashing pathnames

Our second design choice concerns how directories are assigned to islands or how a client decides which island to contact for each operation. The one-island principle implies that the client should go directly to the island that can satisfy the client's request.

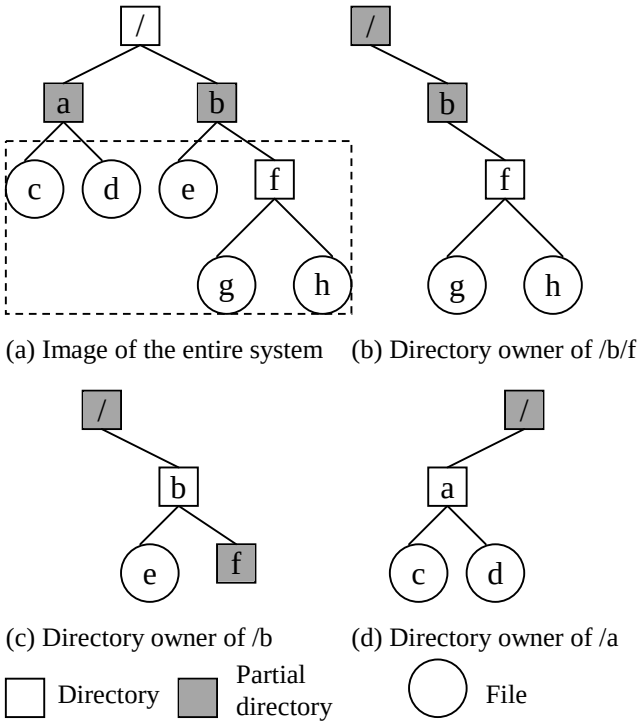
We decided to distribute directories to islands by hashing the full pathnames of directories to island indices. We chose hashing because a hash function can be computed on the client machines without contacting any servers and hence satisfies the one-island principle. We chose to hash the pathname instead of a low-level integer identifier like an inode number because the client always knows the pathname but it might not know the inode number without contacting a server. A hash function inherently provides locality because it has a consistent mapping from directories to islands as far as the function itself does not change.

The potential problems with hashing are load imbalance (where too many directories are hashed to a single island) and high reconfiguration cost (because naïve hashing results in a fixed mapping from directories to islands). To address these problems, we use a combination of universal hashing [10] and extendible hashing [11]. We will describe the algorithms in more detail in Section 5.

3.3 Skeleton hierarchy

Having decided to distribute data at directory granularity by hashing the pathnames, we faced the third design choice: how to store directories inside an island.

We decided to store directories in a *skeleton hierarchy* in the internal file system with the cost of replicating ancestor directories as necessary. The skeleton hierarchy in an island contains the directories hashed to this island and their ancestor directories up to the root. See Figure 4 for an illustration. The skeleton hierarchy is stored in the



(a) Image of the entire system (b) Directory owner of /b/f

(c) Directory owner of /b (d) Directory owner of /a

Figure 4. Skeleton hierarchy. Figure (a) without the directories and files inside the dashed rectangle is the image of the directory owner of root directory /. (b) (c) and (d) are the images of the internal file systems in three other islands. / is replicated in all islands. /b is replicated in its parent owner (a), directory owner (c) and the directory owner of its sub directory /b/f. /a and /b/f are replicated only in their parent owners and directory owners because they are leaf directories. Shaded directories in the figure represent replicas that contain only attributes and partial contents or no contents.

unmodified internal file system as a normal tree. Therefore, IFS can inherit most functions from its internal file systems, such as metadata structures, disk allocation, I/O scheduling, caching, locking, security, recovery, etc.

The alternative is to store directories in a flat table indexed by the pathnames [4]. However, it will be hard to control accesses using directory security attributes because accesses no longer pass the recursive permission checks of ancestor directories.

The consequence of storing data in skeleton hierarchies is the replication of ancestor directories. The next section explains why the replication cost is low.

4. Replication and consistency

Directories are stored in skeleton hierarchies inside islands with replicated ancestor directories. We introduce two terms, *directory owner* and *parent owner*, to help us explain the replication in IFS. The *directory owner* of a

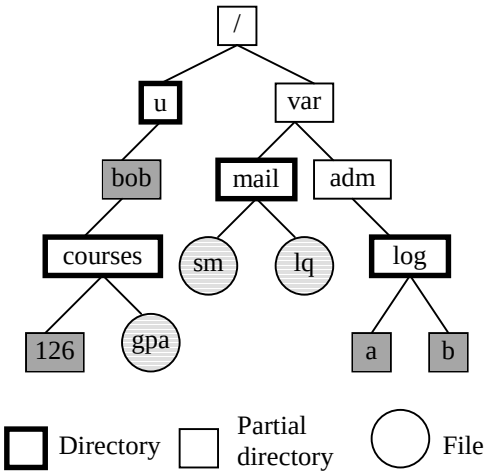


Figure 5. Directory owner and parent owner. This is an image of the internal file system in an island that is the directory owner of the highlighted directories or the parent owner of the shaded files and directories. Partial directories are replicas that contain only attributes and partial contents or no contents.

directory is the island to which the directory is hashed. The *parent owner* of a file or directory is the directory owner of its parent directory. A file resides in exactly one island, its parent owner. A directory, however, will be replicated in two islands at the time of creation, its directory owner and its parent owner (unless the two owners happen to be the same). A *non-leaf* directory, one that has sub directories, will be replicated in all the owners of its descendent directories. Figure 5 gives an illustration of the directory and parent owners.

However, only the directory *attributes*, not the directory *contents*, need to be replicated. Directory attributes include name, size, security, time stamps, read-only tag, compressed tag, etc. Directory contents are the lists of names and addresses of sub directories and files. Adding or removing files in a directory owner does not affect other replicas of the directory since they modify the contents, but not the attributes. Changes to directory attributes will, however, affect other replicas. Therefore, we want to replicate only those attributes that are needed when a descendent of the directory is being looked up.

We divide directory attributes into two categories, *static* attributes and *dynamic* attributes, based on their access patterns. A static attribute is more frequently read than written, and a dynamic attribute is more frequently written than read. We apply *read-one-write-all* policy to static attributes and *read-one-write-one* policy to dynamic attributes. Attributes such as name, security, read-only tag and compressed tag are static. Updates to static attributes of non-leaf directories will be broadcast to all islands so that read requests for these attributes

satisfy the one-island principle. A cross-island protocol is needed for the atomicity and serialization of those operations in the face of partial failures. Attributes such as size and time stamps are dynamic. To avoid the cost of keeping dynamic attributes consistent across replicas, we do *not* replicate these attributes, but read and write them in a single island, the directory owner. Therefore, there is no consistency problem with dynamic attributes.

The following operations in IFS involve multiple islands and require a cross-island protocol:

- *CreateDir* and *RemoveDir* (coordinated by the directory owner and involving the parent owner): when a directory is created or removed, it is created or removed in both the directory owner and parent owner. (A directory is replicated in other islands on demand and removed when it becomes empty.)
- *SetDirAttr* (coordinated by the directory owner and involving all islands): when the static attributes of a non-leaf directory are changed, the change is made in all islands that have a replica of the directory.
- *SymLinkDir* and *DeleteLinkDir* (coordinated by the parent owner of the symbolic link and involving all islands): when a symbolic link to a directory is created or deleted, it is created or deleted in all islands. See Section 10 for details.
- *RenameDir* (coordinated by the directory owner and involving multiple islands): when a non-leaf directory is renamed, all subdirectories might be hashed to different islands, and hence need to be migrated to their new owners. See Section 10 for details.

We are going to find the answers to the following questions in our statistical analysis of the access patterns and contents in the existing file systems in use (see Section 6):

- What is the upper bound of the storage required for the replication of static directory attributes?
- Can the consistency of IFS be maintained at a low cost? (By "low consistency cost" we mean minimized number of cross-island operations that need a consistency protocol, rather than optimized performance for individual operations of this type.)

5. Hashing, load balance and rebalance

The goals of our hashing algorithm are fast directory-to-island mapping, load balance and low reconfiguration cost. We take two steps in hashing a pathname to an island: first, hashing the pathname to a *bucket* (an integer value) with a universal hash function; second, hashing the bucket to an island with an extendible hashing table.

5.1 Universal hashing

Universal hash functions were presented by Carter and Wegman [10] and have the property of *input independent* distribution. We chose the function they called H_3 for our first step of hashing because it can hash a bit string to an integer bucket by boolean operations in expected time linear in the string length.

We shall quantitatively analyze the directory and workload distributions as the result of the universal hash function as follows. Assuming that objects *O* are to be distributed to units *U*, we define the *imbalance* I_{OU} as the standard deviation of objects *O* in units *U* divided by the average objects in each unit. I_{OU} is 0 if the distribution is perfectly even. Let *B* be the number of buckets, *D* be the number of directories, *W* be the workload, and *S* be the number of islands. Theoretically, the imbalance in directory distribution across buckets is $I_{DB} = \sqrt{\frac{B-1}{D}}$ (see Appendix B) and can be made small if we choose the number *B* of buckets to be much smaller than the number *D* of directories in the file system. The imbalance in workload distribution across buckets is $I_{WB} = I_{DB} \cdot \sqrt{I_{WD}{}^2 + 1}$, where I_{WD} is the imbalance in workload distribution across directories. See Appendix B for derivations. A universal hash function does not have control on the workload distribution across directories. Therefore, for load balance purpose it is not sufficient to simply assign a bucket to each island, i.e. to make $I_{WS} = I_{WB}$.

5.2 Extendible hashing

We use a second step of hashing to assign buckets to islands so that workload is balanced across islands.

We use a variation of the standard extendible hashing structure [11] as follows. The H_3 hash function is configured to generate values over a relatively large range, e.g. 32-bit binary integers. An *extendible hash table* is constructed to map the H_3 values to island indices. The size of the table is $H \geq 2^{c \cdot \lceil \log S \rceil}$, where *c* is a constant and *c*>1 so that the average number of table entries per island is large enough to balance the workload. *H* grows with the number *S* of islands. The hash table is indexed by the highest $c \cdot \lceil \log S \rceil$ bits of the H_3 values and each table entry is assigned to an island. Initially, each island is assigned an equal number of table entries. As load imbalance increases or islands are added to or removed during system reconfiguration, the table entries are reassigned to islands to rebalance the load. (Note that table entries will not be reassigned when islands leave or join the system due to failures and recoveries.)

We use a greedy algorithm that attempts to balance both workloads (as first priority) and table entries (as second priority) across islands. We define an *overloaded* island as one whose workload is above average or one that is to be removed from the system, and an *underloaded* island as one whose workload is below average or one that is

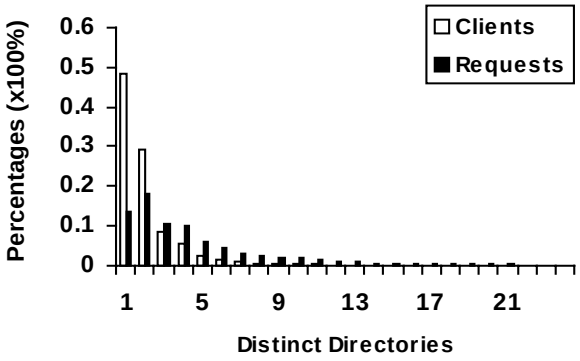


Figure 6. Histograms of clients and requests by distinct directories in the web traces. Accesses to more than 24 directories account for 0.4% clients and 19.3% requests, and are omitted in the graph for readability.

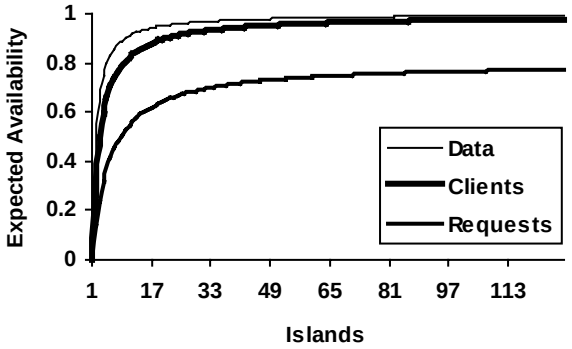


Figure 7. Expected availability for data, clients and requests in the web traces with the failure of 1 out of *s* islands. The x axis is the number *s* of islands. The y axis is (1-1/s) for data and $\sum p(n)\cdot(1-1/s)^n$ for clients and requests, where *n* is the number of distinct directories and *p(n)* is the percentage of clients or requests in each category of *n*.

newly added to the system. For each overloaded island, remove its table entries until no more entries can be removed without underloading the island. Sort the entries removed from the overloaded islands in descending order of their workloads. For each entry in the sorted order, assign it to the least loaded island and updates the workload of the island accordingly.

The reassignment is *monotonic* since entries will only be moved from overloaded islands to underloaded islands. Therefore, only a minimal amount of data needs to be migrated for load rebalance, e.g. only $\frac{1}{S}$ data needs to be migrated when an *S*th island is added to the system. Migration can be done in parallel in all islands since we need not worry about an island becoming full during the migration. The rebalancing algorithm will be scheduled to run when the load imbalance exceeds a threshold so that no island will become full during normal operations. Since the greedy algorithm attempts to evenly distribute table entries across islands in addition to workloads, the imbalances in directory, file and byte distributions across islands are suppressed.

We use the number of bytes that have been accessed since the last rebalance as the measure of actual workload. This can be recorded as the system is running, and requires space for a counter per hash table entry. Since the hash table size *H* in our algorithm is a constant factor of the number *S* of islands, the space required for the workload measurement of the entire system is *O(S)*. The hash table is replicated in all islands and clients, requiring *O(S)* space on each machine. Both space requirements are small.

The rebalancing algorithm will work well as far as no table entry contains more than $\frac{1}{S}$ of total workloads. Otherwise, the hot spot has to be removed either manually or by using a scalable or replicated internal file system [1] [41].

We are going to find the answers to the following questions in our statistical analysis (see Section 6):

- How evenly in practice can the universal hash function we chose distribute pathnames to buckets, considering the fact that pathnames extensively share common prefixes?
- How evenly in practice can the extendible hashing algorithm distribute workload to islands in spite of hot spots?

6. Statistical analysis

In this section, we study the partial availability, load balance and consistency cost by collecting statistics from existing file systems in use. Although the IFS design was motivated by the access patterns of Internet services, we evaluated it in a more generic context.

6.1 Partial availability for applications

As discussed in the previous section, the availability of IFS with partial failures depends on the number of distinct directories client applications access.

We computed the histograms of clients and requests by the distinct directories they touched from the access logs of the web server running on our site. We assume that IFS acts only as a content provider to the web server, i.e. accesses to control information or executables of the web server itself do not count in our statistics. We grouped the HTTP requests into clients by the hostnames or IP

addresses in the requests, and within each client, we grouped requests into directories by the URLs in the requests and maintained a counter for the total number of requests. We computed the histograms from two months' traces, July 1998 (137248 clients and 1304975 requests in total) and January 1999 (166804 clients and 1297428 requests in total). We kept the distinct directories and total number of requests for each client up to an hour, updated the histograms and cleared all clients' records in the end of each hour, and restarted recording for the next hour. The histograms were cumulated for the two months. See Figure 6. The results show that the largest portion (48.3%) of clients accessed only 1 distinct directory in an hour and the largest portion (17.9%) of requests were issued by clients who accessed 2 distinct directories in an hour. Requests are more scattered across categories because larger categories tend to have more accesses and hence weigh more. We computed the histograms for other time windows ranging from 30 minutes to 8 hours, but there was no significant difference across time windows.

Figure 7 shows the expected availability of IFS for data, clients and requests, respectively. Since the majority of web clients access a small number of distinct directories, the expected availability for this class of clients is high in spite of the fact that a partial failure in IFS causes a random set of directories to be inaccessible.

We also computed the histograms of application groups and file system calls by the distinct directories they touched from the file system traces taken on a file server in Hewlett-Packard Labs for the week starting September 24, 1999, which consisted of 5,995,712 pathname-based low-level file system calls such as open(). The users of that file server are 5 to 10 researchers who access files through applications like emacs, g++, netscape and shells on UNIX workstations. We grouped file system calls by process ids and divided process ids into "application groups" by the fork() system calls. Each application group is associated with a window or session manager, but some are finer-grained because we do not know about the fork() events that happened before the tracing program started. In the traces we used, 183,915 fork() events were recorded and 5,170 groups were identified. We computed the histograms for the time windows ranging from 1 minute to 1 hour. We use the overall histogram of application groups below since there was no significant difference across time windows. Similar to the web traces, the largest portions, 26.2% and 14.8%, of application groups accessed 1 and 2 distinct directories, respectively; different from the web traces, more groups accessed a larger number of distinct directories, e.g. 17.3% groups accessed more than 24 directories. As time window increases, more file system calls were counted in larger categories of distinct directories. For example, in 5 to 10 minute windows, the largest portion (17.6%) of calls were in the category of 1 distinct directory; in the 1-hour windows, the largest portion (44.4%) of calls were in the category of 7801 distinct directories. The users of those application groups will be affected by a lasting partial failure in IFS, for IFS was not designed for that class of users.

6.2 Replication cost and load distribution

We took snapshots of five UNIX and Linux file systems in use, using the shell command "ls -l -A -R /". The five systems provide file services to a web server ("web"), a research project ("project"), a CD-ROM image server ("cdroms"), a department ("department") and a university ("university"), respectively. Most of the systems consist of multiple partitions that are mounted together via NFS. For our statistical studies, we pretended that each system is a single file tree stored in IFS.

Replication cost: We computed the *upper bound of the replication storage*, i.e. storage for replicating all non-leaf directory attributes and all symbolic links to directories in all islands. Let D be the number of directories, F be the number of files, I be the inode size in bytes, and T be the total number of bytes for directory and file contents. Then the storage required for the entire system without replication, the *net storage*, is $I \cdot (D+F)+T$ bytes. Let S be the number of islands, N be the number of non-leaf directories, Q be the number of symbolic links to directories, and L be the size of a symbolic link. Then the upper bound of the replication storage is $I \cdot N \cdot (S-1)+(I+L) \cdot Q \cdot (S-1)$.

	Web	Pro-ject	Cd-roms	Depart ment	Univer -sity
Directo-ries (D)	5938	16233	25195	178662	178974
Files (F)	104186	222958	228326	3377478	1653946
Contents (T) (GB)	4.74	11.01	14.55	105.9	51.27
DirLinks (Q)	183	450	1010	3339	35698
Non-leaves(N)	1863	4189	10102	46639	45427
Islands (S)	1	3	4	31	15
Rep. (GB)	0.000	0.036	0.130	5.815	4.809
Rep. (percent)	0.0%	0.3%	0.8%	4.7%	7.7%

Table 1. Replication cost. Each column is an existing file system in use. Row "Rep. (GB)" shows the upper bound of the replication storage, computed as $I \cdot N \cdot (S-1)+(I+L) \cdot Q \cdot (S-1)$. Row "Rep. (percent)"

shows the percentage of the upper bound of replication storage to the total storage. The net storage is computed as $I \cdot (D+F)+T$. The replication storage in the web system is zero because there is only one island for the web file system.

Based on existing system configurations, we assumed that the capacity of each island was roughly 4 GB, hence the number of islands for each file system is the total number of bytes in the system divided by 4 GB. We computed for each system the upper bound of replication storage with *I*=4KB and *L*=1KB. See Table 1. The percentage of replication storage to total storage ranges from 0.3% to 7.7%. Given the decreasing costs for storage devices nowadays, the replication cost is insignificant.

Load imbalance: With the same snapshots of the five file systems, we computed the load imbalances described in Section 5 and Appendix B, and compared them with their theoretical expectations. Since the access logs of the five systems are not all available to us, the number of bytes, instead of accessed bytes, was used as the measure of workload in our study. See Table 2.

To compare I_{DB} with its theoretical expectation $\sqrt{\frac{B-1}{D}}$ and across different systems, we fixed the number B of buckets or extendible hash table entries to be 256. The directory distribution across buckets is very close to its theoretical expectation. The byte distribution across buckets is less close to its expectation probably because of the inaccurate assumption of pairwise independency between directory workloads. (See Appendix B.) The byte distribution across directories is determined by the usage of the systems and is considerably uneven. The second step of hashing, i.e. the extendible hash table, was designed to balance the workload, i.e. the number of bytes in our study, across islands. The number of islands for each system is the same as in Table 1. Table 2 shows that bytes are evenly distributed across islands. The extendible hashing algorithm is independent of the inputs; therefore, it can also evenly distribute actual workload across islands if the input is the actual workload recorded in real systems.

	Web	Project	Cdroms	Depart-ment	Univer-sity
I_{DB}	0.21	0.13	0.10	0.04	0.04
I_{DB}	0.19	0.13	0.10	0.04	0.04
I_{WD}	5.93	15.56	17.58	11.95	17.70
I_{WB}	1.14	2.03	1.76	0.48	0.71
I_{WB}	1.23	1.94	1.81	0.68	0.71
I_{WS}	0	0.0004	0.0001	0.0279	0.0087

Table 2. Load imbalances in five file systems. I_{DB} is the imbalance in directory distribution across buckets; I_{WD} is the imbalance in byte distribution across directories; I_{WB} is the imbalance in byte distribution across buckets; I_{WS} is the imbalance in byte distributions across islands. The imbalance value is 0 if the distribution is perfectly even. I_{WS} is 0 in the web system because there is only one island for the web system. The shaded row of I_{DB} is the theoretical expectation $\sqrt{\frac{B-1}{D}}$ of I_{DB}; the shaded row of I_{WB} is the theoretical expectation $I_{DB} \cdot \sqrt{I_{WD}{}^2 + 1}$ of I_{WB}.

Hot spots: Table 3 shows the hot spots in various distributions in terms of largest/average sizes. We observed the following properties in all five systems: the largest directory, one that contains the most bytes, has 81.30% to 99.99% of its bytes stored in a single file, which in turn is the largest file in the entire system; the largest file is small compared to the entire system, hence it does not prevent a good overall load balance across islands. It is worth noting that the relatively high imbalance in the departmental file system is due to the fixed number 256 of hash table entries: the largest table entry accounts for more than $\frac{1}{S}$ of total bytes. In our implementation, the table size grows with the number of islands.

	Web	Project	Cdroms	Depart-ment	Univer-sity
H_{DB}	1.68	1.47	1.30	1.11	1.12
H_{WD}	243.1	1843.6	939.1	5703.3	3077.1
H_{WB}	10.76	29.56	13.47	9.54	6.50
H_{WS}	1	1.0003	1.0002	1.0385	1.007

Table 3. Hot spots. H_{DB} is the largest/average bucket size in directories; H_{WD} is the largest/average directory size in bytes; H_{WB} is the largest/average bucket size in bytes; H_{WS} is the largest/average island size in bytes. The value is 1 if the distribution is perfectly even. H_{WS} is 1 in the web system because there is only one island for the web system.

6.3 Operation breakdown

We shall analyze the expected consistency cost of IFS below. By "low consistency cost" we mean minimized number of cross-island operations that need a consistency protocol, rather than optimized performance

for individual operations of this type.

Previous studies of file system traces indicated that the cross-island operations are rare. Traces taken on the Sprite system [39] show that setattr, rmdir and mkdir account for only 0.7%, 0.03%, and 0.02% of total operations, respectively. The SPEC SFS or LADDIS benchmark [12] generates an operation mix based on NFS client workload studies, which consists of 1% setattr operations, 1% remove operations and 2% create operations. Recent traces taken on NFS clients [38] consist of 0.092% chmod, 0.015% chown, 0.003% symlink, 0.015% readlink, 0.013% rename, 0.013% mkdir, and 0.012% rmdir. The majority of the operations in all those studies are reading attributes, reading files, writing files and reading directories, which account for 84% to 96% of total operations. Some of the operations in those studies, e.g. setattr and chmod, were not recorded for files and directories separately; therefore, the percentages of those operations on directories will be even lower than reported.

It is well known that file access patterns are always specific to the operating systems where the traces were taken. Since we implemented IFS on Windows NT as opposed to UNIX, in which the Sprite and NFS traces were taken, we felt it important to study the file access patterns in NTFS. We chose 7 workstations running Windows NT 4.0 and collected statistics on operations by running a trace program on each workstation. The users of the workstations include three graduate students, a software engineer, a home user and several lab users. The trace programs were run for 2 to 7 days and collected 30,391 to 480,385 total events.

The trace program forks a thread to wait on each file system related event such as FileAdded through the NTFS event notification interface ReadDirectoryChangesW [40]. The events are not necessarily one-to-one mapped to file system operations, and there is no detailed documentation on the mapping. Hence we present the raw events in Table 4 and infer the operation breakdown with the empirical rules: reads to files and directories are not detected if the reads hit in cache; writes to files and directories are not detected until the cache is flushed; an attribute-change event comes with a name-change, size-change, or security-change event; reading attributes as well as reading contents changes last access time if it does not hit in cache.

Table 4 shows that, on average, one-island operations account for 99.8% of total operations. The slow operations in IFS, e.g. setting directory attributes, renaming directories, creating symbolic links to directories, are rare. Therefore, the amortized cost for keeping replicated state consistent across islands is low in IFS.

No.	Events	Average	Standard Deviation
1	Total Events	244408	140571
2	FileAdded	3.34%	1.70%
3	FileRemoved	2.38%	1.70%
4	FileRenamed	0.41%	0.31%
5	DirAdded	0.04%	0.07%
6	DirRemoved	0.03%	0.07%
7	DirRenamed	0.00%	0.00%
8	FileAttrModified	26.8%	10.8%
9	FileWritten	35.5%	11.3%
10	FileAccessed	16.3%	8.60%
11	FileSecurityModified	0.03%	0.04%
12	DirAttrModified	0.07%	0.07%
13	DirWritten	1.23%	1.59%
14	DirAccessed	13.9%	17.8%
15	DirSecurityModified	0.00%	0.00%
16	FileLinkModified	0.16%	0.08%
17	FileLinkRead	0.09%	0.10%
18	DirLinkModified	0.00%	0.00%
19	DirLinkRead	0.001%	0.002%

Table 4. Percentages of file system events in NTFS traces. Row 1 (Total events) shows the total number of events in each trace. Rows 2 through 19 show the percentage of each event. Shaded events correspond to cross-island operations in IFS. The FileLinkModified (row 16) and DirLinkModified (row 19) events include creating, removing, writing and setting attributes on symbolic links to files and directories, respectively. The FileLinkRead (row 17) and DirLinkRead (row 19) events are resolving symbolic links to files and directories, respectively. The column "Average" shows the percentage of each event averaged over all traces. The column "Standard Deviation" shows the standard deviation of the percentages of each event in each trace. Events not shown in the table have zero percentages.

Given the probabilities of one-island ($P1$), two-island ($P2$) and all-island (Pa) operations, where $P1+P2+Pa=1$, we can predict the speedup efficiency at large scale with a simple model. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup efficiency with n servers is $1/(1+$*overhead-per-operation*$)$, where *overhead-per-operation* is the average number of server-to-server RPCs per operation and equals $(2-1)*2*P2+(n-1)*2*Pa$. (The factor 2 results from the two-phase commit protocol. See Section 8.) Two-island operations include CreateDir, RemoveDir, ReadFileLink and ReadDirLink; all-island operations include SetDirAttr, SetDirSecurity, SymLinkDir and RenameDir. Some operations, e.g. SetDirSecurity and SymLinkDir, did not show up in our statistical experiments; we inferred their percentages from other statistics [38]. The resulting percentages are $P1=99.768\%$, $P2=0.161\%$ and $Pa=0.071\%$. From the

speedup efficiency model above, we know that, with the efficiency higher than 50%, the system can scale up to 702 islands.

The rest of the paper describes the protocol design, implementation and performance measurements of Archipelago.

7. Rebalance protocol

When load imbalance across islands exceeds a threshold as the system ages or when islands are permanently added to or removed from the system, hash table entries need to be reassigned to islands and data needs to be migrated between islands to rebalance the load. (Note that rebalance will not be invoked when islands leave or join the system due to failures and recoveries.) We describe the protocol in details below.

An island is designated as the coordinator in each rebalance. Each island has a unique identifier ranging from 0 to n-1, where n is the number of islands in the current configuration. If no islands are added or removed during a rebalance, island 0 is the coordinator. Only the highest numbered islands can be removed from or added to the system during a reconfiguration. If r islands (numbered n-r through n-1) are to be removed, island n-r will be the coordinator; if a islands (numbered n through n+a-1) are to be added, island n+a-1 will be the coordinator. Given the current configuration and its own identifier, a coordinator always knows which other islands are to be added or removed.

The rebalance is committed in two phases. Each configuration is associated with a version number, and each committed rebalance increases the version number by 1. First, the coordinator attempts to collect workload statistics from all islands, each island logs a "preparing rebalance" message in permanent storage. If any island is inaccessible, the coordinator aborts and notifies the system administrator; otherwise, the coordinator constructs a new hash table that rebalances the workload across the islands in the new configuration, and publishes the new configuration file including the new hash table and increased version number at a well-known location. Second, the coordinator sends a "committing rebalance" message to all islands including the added or removed ones, and then all islands load the new configuration file from the well-known location.

Once the rebalance is committed, each island checks whether it is the source or destination of the monotonic data migration by comparing the old and new hash tables. The destination islands simply log a "rebalance completed" message and return to normal state. Each source island forks a thread, called the *migrator*, to migrate the directories that are no longer hashed to its own index to their new owners. Migration can be done in parallel in all islands since we need not worry about an island becoming full during the migration. The migration will be resumed as necessary with the information recorded in the log, should an island crash during the rebalance. When it finishes, the migrator logs the "rebalance completed" message and exits.

There are two forms of migration during the rebalance: background migration and on-demand migration. The migrators move data in the background. If a new owner receives a request for a file that has not been migrated yet, it issues a request to the old owner to move the file immediately. We call this *on-demand migration*. This is a better approach than waiting for the migrator in the old owner to initiate the movement because waiting could lead to deadlock. However, on-demand migration can cause three types of race conditions: (1) the migrator could not find a file because that file had already been migrated on demand; (2) the migrator tries to move a file but the file has already been created in the destination island by on-demand migration; (3) a file could not be moved because it was in use by another thread. To cope with the on-demand migration, the migrator repeatedly scans the internal file system, detects the race conditions and temporarily skips the suspect directories and files. The same error detection scheme applies to situations where destination islands crash during the rebalance. Client accesses during migration will *not* directly cause race conditions because clients are never allowed to access files or directories in their old owners once the rebalance is committed. They can only access files or directories in their new owners *after* the files or directories have been migrated either in background or on demand.

The hash table is replicated on all clients' machines as well as in all islands, along with the version number. The table size is proportional to the number of islands. Clients' copies of the hash table are updated *lazily*: each request from a client carries the client's current version number, and a client will be asked to load the new configuration file from the well-know location when its version number is found to be out of date. Islands act as clients when they communicate with each other; therefore, the same scheme applies to islands that crash or disconnect from the coordinator before they receive the "committing rebalance" messages: upon first contact to any updated island, the out-of-date islands are forced to load the new configuration file.

A rebalance will be invoked when the load imbalance exceeds a threshold so that no island could become full during normal operations. We expect that a reasonable threshold can be set so that the rebalance occurs at a nondisruptive frequency, e.g. once every month.

8. Consistency protocol

Since certain states, e.g. static directory attributes, are replicated across island, a cross-island protocol is

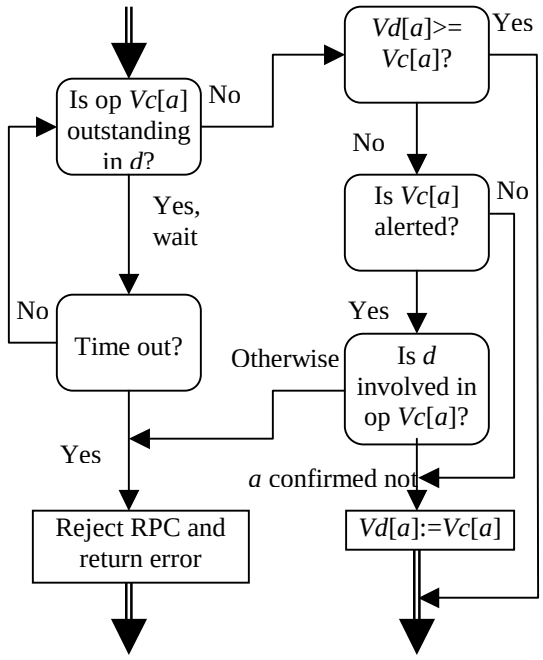


Figure 8. Synchronization of an RPC client *c*'s clock *Vc*[*a*] with the island *d*'s clock *Vd*[*a*], where 0<=*a*<*n*. Op *Vc*[*a*] is the cross-island operation that generated *Vc*[*a*].

necessary to keep the replicas consistent.

A typical approach to maintaining consistency on replicated data is to acquire globally exclusive locks on an object before making changes to the object. To avoid deadlocks and to handle partial failures and network partitions, such a locking protocol needs to be used in combination with other mechanisms such as timeout [28], majority consensus [1] and/or versioning [16], and could be quite complicated to implement.

The island-based design eases the consistency maintenance in two ways:

- The majority of operations involve a single island, hence do not require a cross-island protocol for consistency.
- 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 of the protocol ensures that no conflicting updates will occur even in the face of network partitions.

The lack of conflicting updates in the face of network partitions in Archipelago largely relaxes the synchronization constraint. We designed and implemented a protocol that guarantees serialization as well as atomicity of cross-island operations, and was easy to implement. We handle node crashes and network partitions in a generic way, i.e. we always assume the worst case for safety purpose. Therefore, we do not need to determine the precise type of failure. Although some measures are overkill for one of the failure types, they do not hurt the overall performance because failures are much rarer than normal operations. Correctness and hence simplicity are more critical in the design of such a protocol.

8.1 Atomicity

The basic consistency guarantee our system provides 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. (One-island operations are guaranteed by the internal file systems to be atomic.)

We use a vector of logical clocks for the atomicity of cross-island operations. Each island has its own 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 (through a *remote procedure call* or *RPC*) to an island carries the sender's current clock vector for synchronization with the receiver's vector before the RPC is processed, and returns the receiver's vector to the sender after the RPC 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.

Let island *a* be the coordinator of a cross-island operation, *v* be the new clock value generated by the operation and the identifier for the operation itself, island *b* be any other island involved in the operation *v*, *c* be a client, *d* be any island, *Va*, *Vb*, *Vc* and *Vd* be the clock vectors of *a*, *b*, *c* and *d*, respectively. We maintain the following invariants in the usage of the clock vectors for the atomicity of *v*:

1. Operation *v* locally committed in *a* ⇔ *Va*[*a*]=*v*; and operation v locally committed in *b* ⇔ *Vb*[*a*]=*v*. That is, the local commit of an operation and the update of the coordinator's clock are atomic in each island.
2. *Vc*[*a*]=*v* ⇨ *b* has already been notified of operation *v*. That is, 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.
3. *c*'s request that carries *Vc* (*Vc*[*a*]=*v*) can be processed in *d* ⇨ operation *v* is not outstanding in *d*. 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.

Invariant 1 is maintained by guarding the local commit of an operation and the update of the coordinator's clock in each involved island with a local lock in that island. Invariant 2 is maintained by a two-phase commit protocol [23]: 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. Invariant 3 is maintained by the clock synchronization shown in Figure 8, which is an extension to Lamport's algorithm [22]. Because of invariant 2, an island can determine whether it is involved in an operation during the clock synchronization without contacting the coordinator of that operation if no network partition is present.

The three invariants above guarantee that an island will never expose the intermediate state of any operation to clients and does not require an involvement checking with the coordinator for each operation that the island has not seen but a client has, if no network partition is present.

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. Therefore, a network partition, if there is any, can be detected and the RPC will be rejected to avoid inconsistency. 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 8.

8.2 Serialization

All the one-island operations on the same object are done in the same island, hence are serialized in the internal file system. 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, append-only logging* [14]. The coordinator always writes a record with its clock vector to disk 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. We always keep the last record on disk even if it has been committed until a new one overwrites it for two reasons. First, it is important for an island to "remember" its latest clock after it recovers so that operations in this island before and after the crash carry consistent clocks. Second, this scheme saves us two extra writes to disk per operation, one for recording the latest clock, the other for marking the new end of file.

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 naturally serialized unless an involved island failed. If the coordinator for an operation failed, the client stub will return an error but will not leave the system in an inconsistent state, i.e. it is in a *fail-stop* mode. If an involved island d other than the coordinator a failed, the client will observe the operation as completed and proceed with successive operations. When it recovers, d will receive the lists of operations to commit from all surviving islands. Consecutive operations by the same thread are guaranteed to have ascending clock vectors because, with the logical clock synchronization (Figure 8), the clock vectors in all islands and clients never decrease and always increase upon cross-island operations, even with network partitions. That is, d will be 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, $c1$ and $c2$, interact with each other by accessing the same object in the file system 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$>V2$) can be serialized during a failure recovery.

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 $V1[j]>V2[j]$. 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 far as the file system was concerned [22].

To summarize, the consistency protocol guarantees the following serializations for cross-island operations:

1. All operations on the same object are serialized, i.e.

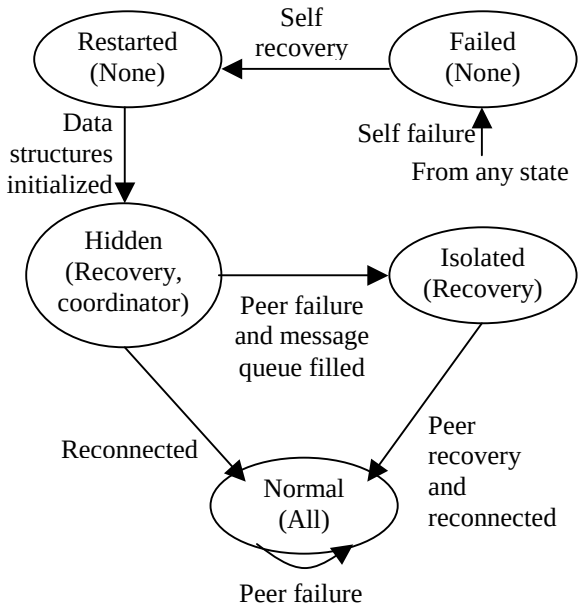


Figure 9. State transitions of an island in response to various failures and recoveries. The types of requests accepted in each state are listed in parenthesis. Each transition is labeled with the event that triggers the transition. "Reconnected" is the event that the recovering island has reconnected to and exchanged logs with all other islands, and has committed all operations in logs in ascending order of their clock vectors.

clients observe them in the same order in all islands.

2. All operations by the same client thread are serialized, i.e. clients observe them in the same order in all islands.
3. Operations by different clients can be serialized 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 $V1<V3$.

9. Recovery protocol

A recovery protocol is designed for islands to recover from various combinations of failures back to consistent states. Table 4 shows the possible failures for an individual island and how the island can be recovered from those failures. We assume that each island stores its log and data in the same internal file system and that no data or log was lost after the island is recovered from a self failure. If the internal file system loses data during a failure, human intervention will be required to reconstruct the data.

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 4. Possible failures and recoveries for an individual island.

Figure 9 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. If a client request is rejected due to a disallowing state in the island, the client stub will keep resending the request till the island transits to an allowing state.

In the *normal* state, an island processes all requests. When an involved island is found to be inaccessible during a cross-island operation, the coordinator island sets the alerted bit in its clock and still processes all requests. The alerted island needs to keep the partially committed operations in its on-disk log till they are committed in all involved islands; if the involved islands are inaccessible for a long time, the on-disk log might fill the internal file system. The island does not need to transit to a new state in this case because the situation of full disks is handled by the internal file system, i.e. any client requests that require new space, for data or for log, will fail in the internal file system. A record in the log can be deleted once the operation has been committed in all involved islands. When there is no more partially committed record in the log, the alerted bit is cleared from the clock.

A self failure in any state causes the island to transit to the *failed* state, in which no requests, of course, are processed.

As discussed briefly in the previous section, a failed or disconnected island will exchange logs with other islands upon reconnection to those islands. The following invariants are maintained in the state transitions of a

recovering island *r*:

1. No operations in logs can be committed in *r* until all logs from other islands have been received and operations in all logs have been sorted in ascending order of their clock vectors (with a tie resolution rule for concurrent vectors). That is, operations serialized in real time will be committed in *r* in the same order as if *r* had not failed.
2. No client requests or requests that indirectly affect clients' view of the system state will be processed in *r* until all operations in logs have been committed in *r*. That is, the inconsistent state of *r*, if there is any, is invisible to clients.

When it is recovered, an island transits from the failed state to the transient *restarted* state, in which it initializes necessary data structures such as the hash table 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 using the logs. The island sends to all other islands the lists of cross-operations in its log that involved the receiver, receives from other islands the lists of operations that involved itself, and stores them in a *message queue*. To tolerate failures during recovery, a retransmitted log replaces any outstanding or logged operations received from the same island earlier. In the hidden state, the island rejects all client requests since it might be in an inconsistent state. It 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, but stores them in the message queue instead of committing them synchronously to reserve the ordering. If the message queue becomes full, the island transits from the hidden state to the *isolated* state, in which it deletes the new operations from the message queue to make room for old operations and 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 needs to ask for the new operations from other islands that it has rejected. After it commits all involving operations, it transits to the normal state. Then it checks in the log whether any data migration for rebalance or for renaming directories was in progress at the time it failed and resumes the migrators as necessary.

10. Other design issues

Archipelago inherits most functions from its internal file systems, such as metadata structures, disk allocation, I/O scheduling, caching, locking, security, recovery, etc.; therefore, we are not concerned about all the low-level details in file system design and implementation. However, certain functions in internal file systems need to be extended to adapt to a distributed environment.

10.1 Symbolic links and renaming directories

Symbolic links in Archipelago are implemented as files containing a pathname to a file or directory. Symbolic links to files are easy to manage because they cause at most a redirection from the owner of the symbolic link to the owner of the target file. However, a pathname with symbolic links to directories will not be hashed to the proper island. To solve this problem, we replicate all symbolic links to directories in all islands. Upon receiving a request for a file or directory that is not found locally, an island checks whether any components of the pathname are symbolic links to directories, without contacting other islands. If none of them is, it returns an error; otherwise, it redirects the request to the real owner after resolving the symbolic links. Similar to the replication of static directory attributes, the replication of symbolic links to directories does not require much space, and the creation, modification and deletion of symbolic links, which will involve all islands, are rare operations. See Section 6.

Renaming a directory in Archipelago is an expensive operation because all the subdirectories below the renamed directory are likely to be hashed to different islands. We try to hide the latency of such an operation by using a symbolic link and a thread similar to the migrator described in Section 7. A symbolic link is created with the new directory name, pointing to the old directory, a migrator thread is forked, and then the rename operation returns as if it is completed. The migrator recursively moves subdirectories and files from their old owners to their new owners in the background. If a request arrives for a file that has not been moved yet, the symbolic link in the pathname will be resolved and the file will be migrated on demand. If a directory is renamed again before the migration completes, accesses to this directory will require multiple symbolic link resolutions. The symbolic links will be removed after the migration completes.

10.2 Security, caching and heterogeneity

We designed and implemented a security model in Archipelago, using the security facilities available in existing file systems and communication protocols, namely access control lists, permission bits, authentication and impersonation. A client is authenticated with its credentials when a connection to an island is established. A thread is forked in an island upon each request from the client. The thread extracts the client's credentials from the authenticated connection and impersonates the client when it processes the request. In this way, file accesses in the request are checked with the client's credentials against the access

control in the internal file systems.

Server-side caching is done in the internal file systems automatically. Archipelago inherently provides locality by hashing, i.e. client requests will always be sent to the server that might have cached the requested data in memory, as far as rebalance is not in progress. Most of the client-side caching protocols in previous work [8] [24] can be adopted in Archipelago. We have not implemented a client-side caching protocol, but we do not expect the island-based design to add any difficulty to the implementation.

In a heterogeneous environment, differences in the internal file systems, such as file attributes and authentication protocols, make the implementation of an integrated file system considerably challenging [25]; but previous work has demonstrated the viability of providing file services across platforms [26].

11. Implementation

We have implemented a prototype of Archipelago on a cluster of Pentium II PCs running Windows NT 4.0. NTFS [16] is used as the internal file system. NTFS uses extensive caching and name indexing for better performance and logs metadata changes for local recoverability. An access control list is associated with each file or directory to check access rights. NTFS can be configured to run on a group of disks with parity striping for high reliability.

An Archipelago server runs on each machine and forms an island. Each client accesses files through a local stub, which forwards the request to a server through Windows remote procedure call (Win32 RPC). The tasks of the server include authenticating clients, validating clients' versions of the hash table, synchronizing clients' clock vectors, and processing clients' requests in the internal file system. The functions of the stub include hashing a pathname to an island, updating local copies of the hash table, synchronizing the clock vectors with servers, maintaining secure RPC connections to servers, tolerating network failures and making file locations transparent to clients.

The server is implemented as a user-level process. The stub is implemented as a dynamic link library (DLL) that intercepts file system calls. Therefore, client-to-server and server-to-server communications can take advantage of user-level networking in the future [27]. All file system calls on NT go through a system DLL, *kernel32.dll*, and we replace this DLL with our own, which forwards a call to the stub DLL if the file is in Archipelago, or to the original *kernel32.dll* otherwise. We have tested the feasibility of the intercepting approach; however, due to the large number of functions in *kernel32.dll*, it requires more debugging effort to make the new *kernel32.dll* work with existing applications seamlessly. In our experiments (Section 13), we linked the benchmark programs directly with the client stub DLL without the new *kernel32.dll* for ease in running the benchmarks. It is expected to have little impact on the performance results since a call wrapper in an additional DLL takes little time compared to regular file system operations, disk accesses and communications.

The server and stub are implemented in C++, and consist of 3088 and 5415 lines of code, respectively. The server program is linked with the stub library for code reuse purpose. In addition, there are 24042 lines of automatically generated C code for RPC and system call interception. The amount of manually written code in Archipelago is small; therefore, the system is relatively easy to test and maintain.

12. Correctness testing

We do not attempt to theoretically prove the correctness of our consistency and recovery protocols. The basic algorithms, i.e. logical clock synchronization, two-phase commit and logging, 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 [30] [36]. 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 based on the input/output automata (IOA) [29], which was originally developed in Hewlett-Packard Labs [35]. We extended the tool so that it checks the real implementation of a system, rather than a simulation written in IOA style. Unlike the tools that exhaustively search the state space [30] [36], the randomized testing tools *cannot* prove that a system is correct. Instead, it helps identifying incorrect parts of a system by injecting various combinations of events to the system and analyzing the results. Such events typically could not possibly be experienced in real workloads or manual tests in a short time.

The test engine consists of three components, *terminators*, *network partitioner* and *clients*. The terminators are independent threads or processes, one for each island in Archipelago. Each terminator injects crash or reboot events to its associated island at intervals randomly chosen within specified 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, this simple form of simulation 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 repeatedly does a randomly chosen operation with specified frequencies on a randomly chosen object. The number of clients is set to be the same as the number of islands in each test.

The IOA formal language has an interface for defining models for *safety* and *liveness* checking [29]. 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 protocols by manually inserting assertions to key parts of the code and observing the results. 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 in 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 protocol will be added once the system has passed the simpler tests.

The test engine takes parameters such as the lower and upper bounds of various event intervals, and the relative frequencies of operations. We selected the intervals in such a way that they both allow a sufficient number of client operations in each state of the system, and allow the overlap of various independent events to exercise the recovery protocol. We exaggerated the frequencies of cross-island operations from real workloads by two orders of magnitude to exercise the consistency protocol. We tested Archipelago with 4 islands in the randomized test engine. For an early stage of correctness testing, it is preferable to run all 4 server processes on the same machine because it significantly eases debugging. Table 5 shows the parameters and results in testing Archipelago in the randomized test engine for the first 2 days.

Events	Parameters (% or seconds)	Results
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 5. Parameters and results in testing Archipelago in the randomized test engine for the first 2 days. The parameters are the specified frequencies for normal operations and the specified lower and upper bounds of intervals 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 between clients and simulated failures. After surviving through 28 node crashes and 7 network partitions, Archipelago failed one of the assertions and caused the test engine to halt. 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.

We found 14 non-obvious bugs in the protocols during the first 2 days of testing Archipelago. As expected, the bugs are all at implementation detail level and do not invalidate the overall protocol designs. These bugs could not be repeated in normal states or simple forms of failures. 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 the involved islands saw a second operation with the same clock from the same coordinator later. The assertion of at most one outstanding operation per island failed. The fix was 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. The preliminary results are encouraging, hence we are continuing to invest more time in this aspect.

13. Performance

In this section, we present the results of running micro benchmarks and operation mixes on Archipelago in various configurations. The 23 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 an Intel Express 510T Ethernet 100Mbps 24-port switch and run in full-duplex mode. The PCs run Windows NT Workstation 4.0 and the hard disks for

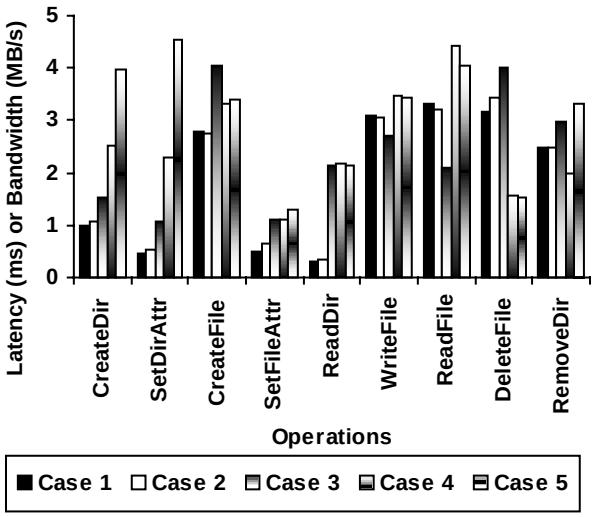


Figure 10. Single client performance. A single client runs the micro benchmarks in 5 cases: directly on NTFS (1), in the same address space as an Archipelago server (2), on a separate machine from the server (3), with two servers (4), and with the consistency protocol turned on in case 4 (5), respectively. The y-axis is the bandwidth in megabytes/second for the WriteFile and ReadFile operations, and the latency in milliseconds for the other operations. Both numbers are measured at the client side. Note that higher columns for the WriteFile and ReadFile represent better performance while lower columns for the other operations represent better performance.

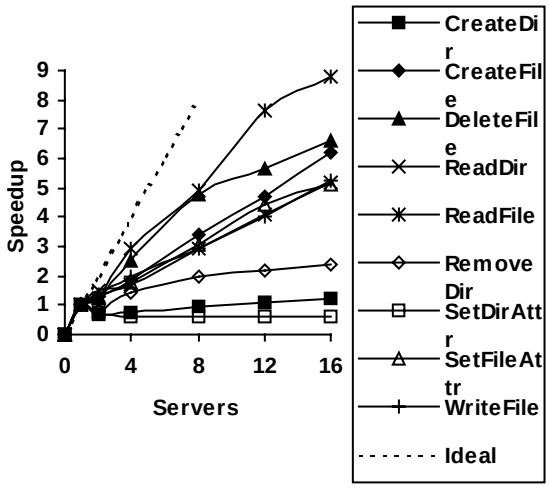


Figure 11. Speedup of throughputs on private data as a function of the number of servers. Number of servers = number of clients = number of private data sets. The speedup is calculated as the absolute throughput divided by the throughput with 1 server. The dotted line represents the speedup with 100% efficiency.

Archipelago are formatted in NTFS.

13.1Micro benchmarks

The set of micro benchmarks consists of 9 phases and each phase exercises one of the file system calls: CreateDir, SetDirAttr, CreateFile, SetFileAttr, ReadDir, WriteFile, ReadFile, DeleteFile and RemoveDir. The basic data set for the micro benchmarks is a project directory that consists of 90 directories, 646 files and 77.2 MB of data in files. We duplicated the directories 40 times, the files 6 times and the contents 2 times, respectively. The 9 resulting phases are: create 3600 directories, set 3600 directory attributes, create 3876 files with pre-allocated space in 540 directories, set 3876 file attributes, read 6634 directory entries, write 154.4 MB data in 1292 files or 180 directories, read 154.4 MB data in 1292 files, delete 3876 files, and remove 3876 directories. The transferred block size in the WriteFile and ReadFile phases is 64 KB or the file size, whichever is smaller. With the data set inflated, the results of the micro benchmarks are reasonably stable. Each test was run more than 3 times and the results shown in this section are the averages.

Other operations, such as moving a file and reading a symbolic link, were implemented with the operations in these micro benchmarks; hence, we did not include them in the tests. We did not intentionally flush the file cache in NTFS during the tests because we would like to treat NTFS, the internal file system, as a functional black box. However, the amounts of data in the WriteFile and ReadFile phases were large enough to overflow the cache.

Single client performance

We ran the micro benchmarks with a single client in 5 cases: directly on NTFS (1), in the same address space as an Archipelago server (2), on a separate machine from an Archipelago server (3), with two Archipelago servers, all on separate machines (4), and with the consistency protocol turned on in case 4 (5). Figure 10 shows the bandwidth in WriteFile and ReadFile and the response times in other operations, all measured at the client side.

The difference between case 1 and 2 is the overhead of computing hash functions. This overhead is low compared to the operation time itself. The difference between case 2 and 3 is the communication (RPC) time between the client and the server. We used Win32 RPC on top of TCP/IP on 100 Mbps switched Ethernet. In our experiments, the average round-trip RPC latency for small messages (~256 bytes) is 0.48 ms and the average one-way large data (64 KB) transfer rate in RPC is 8.67 MB/s. The performance decreased from case 2 to case 3 by an amount comparable to the RPC overhead. The difference between case 3 and case 4 is that the cross-island operations CreateDir, RemoveDir and SetDirAttr

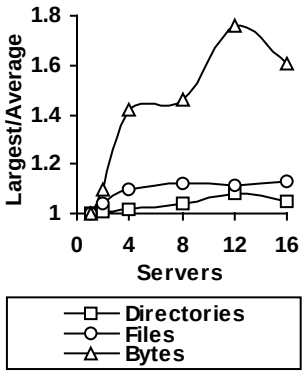


Figure 12. Load imbalances. The y-axis is the largest number of directories, files or bytes stored in a server divided by the average number over all servers. The y-value is 1 if load is perfectly balanced.

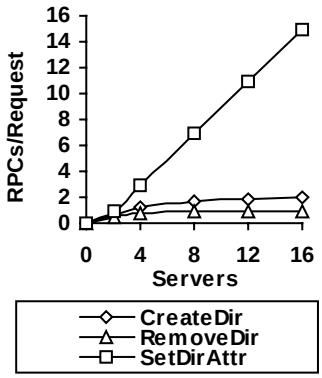


Figure 13. Number of server-to-server RPCs per request as a function of the number of servers. Operations that do not show up in this figure have zero server-to-server RPCs.

involve 1 island in case 3 and 2 islands in case 4. Therefore, the response times for those operations were increased from case 3 to case 4 except RemoveDir. Operations such as ReadFile and RemoveDir were faster in case 4 because there was more total cache space in case 4. The difference between case 4 and case 5 is the overhead of the consistency protocol. The consistency protocol slows down the cross-island operations but does not have a noticeable impact on one-island operations. The response times of CreateFile are larger than those of CreateDir in all cases because the client pre-allocated space for each file in the CreateFile phase.

Scalability on private data

There are multiple clients in this set of tests, each running an instance of the micro benchmarks on its own private data set. Before each phase, all clients are synchronized at a barrier. Each server ran on a separate machine and 1 to 3 clients ran on the same machine. The number of clients was configured to be the same as the number of servers. Given the 23 machines connected by the 24-port Ethernet switch, we scaled the number of servers and clients up to 16 each. We have tested Archipelago with 25 servers on an Ethernet hub and expect the system to be able to scale to larger configurations. In this paper we present only the results of scaling from 1 to 16 servers.

We measured throughput at the server side, i.e. the total number of bytes requested divided by the time for all servers to complete, for WriteFile and ReadFile, and the total number of requests divided by the time for all servers to complete for other operations. To compare the scalability across operations, we calculated the speedup as the absolute throughput divided by the throughput with 1 server. The 1 server case is the same as case 3 in Figure 10. Figure 11 shows the speedup of throughputs on private data as a function of the number of servers. Most operations scale linearly with the number of servers, but at a less than ideal slope. The overhead results from load imbalance and communications.

The directory, file and byte operations are distributed across 3600, 540 and 180 directories, respectively. The load distribution is expected to be less than ideal due to the small size of the data sets (compared to the size of an entire file system). We calculated the load imbalance as the largest load divided by the average load of servers. See Figure 12. We expect the operations to scale better in real systems with the rebalance protocol, which will be studied in Section 14.

Figure 13 shows the average server-to-server RPCs per request measured in the tests. The two-phase commit protocol was turned off in this set of tests; therefore, the actual numbers of RPCs will be doubled with the protocol turned on. One-island operations do not show up in Figure 13 because they require no server-to-server RPCs. The number of RPCs for SetDirAttr grows linearly with the number of servers; therefore, the speedup curve for SetDirAttr in Figure 11 is nearly flat. The numbers of RPCs for CreateDir and RemoveDir are nearly constants; therefore, these two operations do scale with the number of servers, but slower than the one-island operations.

Impact of the consistency protocol

We turned on the consistency protocol, i.e. clock synchronization, two-phase commit and logging, and reran the micro benchmarks. As expected, the protocol does not have noticeable impact on the one-island operations. Figure 14 shows the throughputs of two cross-island operations, CreateDir and SetDirAttr. (RemoveDir is similar to CreateDir.) The protocol increases the RPCs between servers for cross-island operations by a factor of 2 and requires a log write per successful cross-island operation. As expected, the consistency protocol brings considerable overhead to cross-island operations. The throughput of SetDirAttr does not scale with or without the consistency protocol. The throughput of CreateDir scales at roughly the same rate with or without the protocol.

13.2 Operation mixes

We ran a new benchmark of randomized operation mixes to measure the overall scalability of Archipelago. The new benchmark is similar to the SPEC SFS or LADDIS benchmark [12], but with the extensions to shared

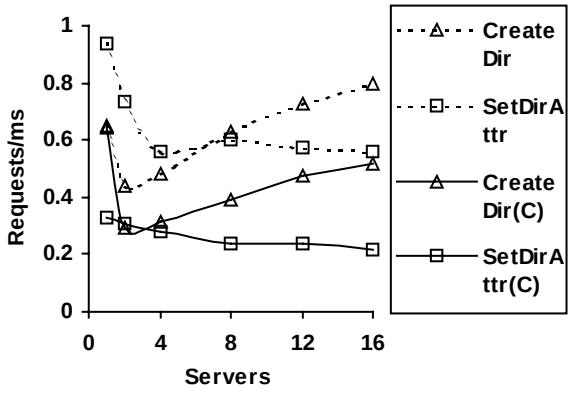


Figure 14. Impact of the consistency protocol on cross-island operations. The curves CreateDir and SetDirAttr are the throughputs (requests/ms) in the experiment in Figure 11. The curves CreateDir(C) and SetDirAttr(C) are the throughputs in the same benchmark but with the consistency protocol turned on. The curves Delays-C and Delays-S are the average delays per request for CreateDir and SetDirAttr with the protocol turned on, respectively.

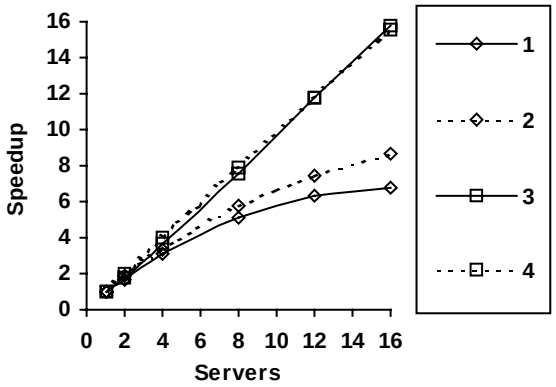


Figure 15. Speedup of throughputs of randomized operation mix. Curves 1, 2, 3 and 4 are actual speedup of operation mix 1 (see Table 3), estimated speedup of operation mix 1, actual speedup of operation mix 2, and estimated speedup of operation mix 2, respectively. The speedup is calculated as the absolute 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.

objects, hierarchical directory structures, randomized pathnames, variable file sizes and scalable workloads. Since Archipelago is implemented on top of NTFS, the operation mix in our benchmark uses NTFS API and is based on the operation breakdown we measured on Windows NT workstations. See Section 6.

We ran the benchmark with 1 to 16 clients and servers on 1 to 16 machines. Each client runs on the same machine as a server, but accesses random files, directories and symbolic links across the entire system. The pre-created data set includes 2000 shared directories, 2000 shared files, 100 shared symbolic links, and the same numbers of private objects 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 lists of the shared objects and its private objects, but does not synchronize with other clients on the creation and deletion of the objects in the shared directories. Therefore, an operation on a shared object might fail if it conflicts with a previous operation on the same object from another client. 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 ran the benchmark with two different operation mixes. Mix 1 exaggerates the cross-island operations and mix 2 is closer to the measured breakdown. The mixes cover a number of typical operations from each category, i.e. one-island, two-island and all-island. Uncovered operations in the measured breakdown are replaced by operations in the same category, e.g. the operation of reading a symbolic link to a file counts for 0.09% in our measured breakdown and is replaced in the mix with the same number of operations that read a symbolic link to a directory. We recorded the actual client operations and server-to-server RPCs in the benchmarks, and estimated the speedups of the overall operation mix accordingly. Table 6 shows the recorded operation mixes and Figure 15 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 $n/(1+overhead\text{-}per\text{-}operation)$, 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

Table 6. Operation mixes. The actual numbers of operations generated in the benchmarks are slightly different from the specified frequencies due to randomization and failed requests. 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 grow with the numbers of clients and servers for the fixed 10 minutes period; the ranges are shown in the last two rows in the table. See Table 2 for explanations for certain 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. Load is well balanced across servers in both operation mixes; the largest/average requests per server are below 1.1 in all cases. 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. It is worth noting that mix 2 scales better than the pure one-island operations in Section 13.1 because considerable load imbalance is present in that benchmark due to the small number of working directories.

14. Case study: Online reconfiguration of a web server

We simulated on top of Archipelago the web server running on our site and measured the performance of online reconfiguration. The file system that the web server originally runs on consists of 5934 directories, 103,426 files and 4.74 GB of contents. It was first copied to an Archipelago with two islands. We added and then removed two islands to the system and studied the performance of data migration and its impact on the performance of web accesses. The hardware used in this set of tests is the same as in previous tests. Table 7 shows the statistics in the addition and removal of two islands without client accesses.

The web server was a Netscape Enterprise Server 3.5.1 running on Solaris 2.6. The hardware for the web server was a Sun Ultraserver-2 with 256 MB of memory and 1 Gbps fiber network connection. The web server kept access logs, which include pathnames of accessed pages, time stamps, client IP addresses, etc. We used the access log for trace-driven study.

Reconfiguration	Addition	Removal
Time (minutes)	26.04	26.03
Migrated (GB)	2.58	2.58
Migrated (files)	52152	52134
Migrated (dirs)	2964	2954
Bytes Before(GB)	2.52, 2.64, 0, 0	1.29, 1.29, 1.29, 1.29
Bytes After (GB)	1.29, 1.29, 1.29, 1.29	2.58, 2.58, 0, 0

Table 7. Statistics in the addition and removal of two islands without client accesses. The row "Time" shows the elapsed time in minutes since the reconfiguration started till the migration of data was completed in all islands. The next three rows show the migrated bytes, files and directories during the reconfiguration, respectively. The last two rows show the byte distribution across four islands before and after the reconfiguration, respectively. (We use the number of bytes as the measure for server loads for simplicity in these experiments.)

We simulated the web server with 16 threads on separate machines, reading the access log and issuing requests to Archipelago as clients. The absolute time stamps in the log were ignored and the traces in the log were consumed as fast as possible. Each thread issued 3000 requests in each test and the overall consumed traces in each test were taken from 00:01:34 to 18:01:48 on March 1, 1999. 699 MB of data in 48000 files were accessed in each test, of which 86 MB of data and 7218 files were distinct.

We ran the simulation in 5 different cases relevant to the addition of two islands and measured the impact of data migration on client performance. The migration was expected to affect client accesses in three ways. First, the background migrators compete with the clients for resources like disk and network bandwidths. Second, when the clients try to access files in the new islands, some files have to be migrated on demand from the old islands. Third, on-demand migration causes race conditions.

In addition to running the simulation before and after the reconfiguration, we ran the simulation in 3 cases during the reconfiguration to separate the impacts of different sources. First, we ran the simulation in the beginning of the reconfiguration to see the impacts of both background migration and on-demand migration. Second, we ran the simulation again, later in the same reconfiguration; since all the requested files had been migrated on demand in the first simulation, the slowdown in this case came solely from the migrators' competition. Third, we reran the reconfiguration and simulation with the migrators disabled to see the slowdown solely from on-demand migration. Table 8 shows the results of the simulated web accesses in the

five cases.

The results show that the migrators had a minor impact on the client performance. In case 4 (migrators only), the migrators consumed only 7% of the overall disk bandwidth and imposed a performance penalty of only 4.5%. The percentages are dependent on the relative numbers of migrators to clients, i.e. 2 to 16 in this case. Client bandwidth was nearly halved by on-demand migration because the amount of data transferred to satisfy a request was doubled. The disadvantage of disabling migrators is that the first accesses to files in the new islands will always require on-demand migration and will see a significant performance drop. Additionally, without migrators, a system administrator cannot tell when exactly the migration is completed. Therefore, enabling migrators is a good idea.

We also recorded the number of race conditions caused by on-demand migration. The race conditions were detected and tolerated by the migrators and were transparent to the clients. The race conditions in case 4 occurred when the migrators initiated on-demand migration for directory attributes replication. The numbers of race conditions were relatively small compared to the number of files migrated on demand. With on-demand migration, the system reconfiguration was made transparent to the clients.

Cases	Clients (MB/s)	Migrators (MB/s)	Migrated files	Race conditions
1	3.94	0	0	0
2	4.52	0.36	7191	84
3	5.68	0	7218	42
4	9.05	0.68	0	8
5	9.48	0	0	0

Table 8. Results in the simulated web accesses. The five cases are before the addition of two new islands (1), with both background migrators and on-demand migration (2), with on-demand migration only (3), with background migrators only (4) and after the addition of two new islands (5). The columns "Clients" and "Migrators" show the aggregate bandwidths of clients and migrators, respectively. The clients' bandwidth is the total number of bytes accessed by 16 threads during the simulation divided by the simulation time. The migrators' bandwidth is the total number of bytes read and written by the 2 migrators during the simulation divided by the simulation time. The column "Migrated files" shows the number of files migrated *on demand* during the simulation. The column "Race conditions" shows the number of race conditions during the simulation due to on-demand migration.

The measured impacts of background migrators and on-demand migration in the reconfiguration tests also apply to the cases of renaming directories (Section 10) because these two procedures share most of the code.

15. Related work

In terms of failure isolation, consistency cost, locality and leveraging functions in local file systems, Archipelago is comparable to wide area file systems such as Andrew [8], Sprite [24] [33], JetFS [16], NFS [9] and CIFS [15]. However, in those systems, data is manually partitioned to servers at sub tree granularity. Therefore, those systems do not share load balance and scalability with IFS. Mounted file systems, such as NFS, do not provide location-transparent name spaces. Others do but use a combination of name caching, location hints, replicated name services, recursive lookup and/or multicast for name lookups. In IFS, name lookups are done by hash functions on client machines without contacting any servers. Both wide area systems and IFS can leverage functions in local file systems.

IFS is designed to match the scalability, load balance and easy management of cluster file systems such as Frangipani [1] and xFS [5]. Those systems were designed to take advantage of the aggregate bandwidth of storage servers connected by fast networks and the addition of servers can improve the performance of individual clients. IFS is designed for larger environments where islands communicate through commodity networks. The goal of scaling IFS is to meet the needs of increased number of workloads or clients.

Many systems, such as global memory systems [6], distributed file systems [5] [7] [16] [18] [21], parallel file systems [4] [19], web or proxy servers [2] [3] [13], and database systems [11], use hash-based techniques for distributing or locating data. Global Memory System [6], xFS [5] and Petal [7] use multi-level maps to translate virtual addresses to physical addresses. The two steps of hashing in IFS differ from those multi-level maps in that inputs to the hash functions in IFS are pathnames while the maps in those systems are keyed by integral addresses. Several parallel file systems, such as Vesta [4] and Galley [19], locate the metadata, but not data, of a file by hashing the pathname. The Locality-Aware Request Distribution (LARD) [3] switches between a hash function and a load-based distribution for locality and load balance in cluster-based network servers with shared storage and read-mostly accesses. Archipelago consistently uses hashing and the hashing algorithm itself is reconfigurable based on loads. It can essentially achieve the same locality and load balance as LARD, but for a more generic system structure and access pattern. The work in [18] distributes and migrates files across servers to minimize the cost/performance ratio. Although a similar hash-based distribution is used, the record-structured and key-accessed files in that system are largely different from the files and directories in IFS and other file systems. The idea of monotonic migration in IFS was inspired by the work on consistent hash

functions for web caching [2]. A consistent hash function is one that changes minimally as the range of the function changes. However, we used an extendible hash table instead of a consistent hash function in our system because it was not clear how a consistent hash function can evenly distribute workload, as opposed to number of buckets, across islands.

Hive [37] is an operating system for large-scale shared-memory multiprocessors with independent kernels called cells. The multi-cellular structure in Hive was designed to improve reliability and scalability. We used the same principle in designing our distributed file system. However, our methods are distinct from those in Hive because the problem areas are largely different.

Storage systems such as Petal [7] use background processes during reconfiguration to migrate data blocks from old storage servers to new ones. We expect the bandwidth in their migration to be higher than the bandwidth of the migrators in Archipelago because there is little overhead associated with metadata operations at the disk block level. On the other hand, it is easier to control the interaction between the migrators and clients in Archipelago given the information available at file system level.

Some replicated file or storage systems, such as Locus [31] and Bayou [32], use a version vector per replica or a logic clock per operation to detect conflicting updates in case of network partitions, and reconcile the conflicts when detected. Archipelago uses a version vector per island to detect and prevent inconsistency in case of network partitions, and no reconciliation is needed because no conflicting updates will possibly occur in the island-based design.

Recovery protocols were addressed in distributed file systems that use extensive client caching. Calypso [34] and Sprite [33] use distributed state among the clients to reconstruct the before-crash state of a recovering server, and guarantee data consistency and congestion control during the recovery. Archipelago resembles those systems in that it also uses distributed state (logs in surviving servers), guarantees consistency and handles message queue overflow during recovery; it differs in that the protocol is designed to update the recovering server with the state that the rest of the system reached *after* it crashed; therefore, it needs to address the additional issue of operation serialization.

16. Future work and conclusion

We designed an island-based file system for improved failure isolation and consistency cost by enforcing a one-island principle in the data distribution. We evaluated the design by analytic modeling and statistical analysis on the access patterns and contents in various existing systems in use. We implemented Archipelago, a prototype of the island-based file system, with a consistency protocol and a recovery for high availability and reliability, and a rebalance protocol for dynamic load balance and low-cost reconfiguration. We built a randomized test engine to test the correctness of the protocols, and studied the performance of Archipelago in micro benchmarks, operation mixes and trace-driven simulations.

We are considering extensions to the hashing of directories. Ideally, we would like to have an adaptive hashing algorithm that determines the height of a sub tree or the granularity of a file to hash based on the current state of load balance and access patterns. We are also going to improve the performance of all-island operations like SetDirAttr by replacing the *2*n* unicast messages and *2*n* replies with 2 broadcast or multicast messages and *2*n* replies, where *n* is the number of islands.

We draw the following conclusions:

- Data loss in case of partial failures can be reduced by replicating a small amount of metadata across islands.
- The majority of web clients are likely to survive a partial failure in IFS.
- Data distribution at directory granularity eases the consistency maintenance across islands.
- A universal hash function can evenly distribute directories to buckets; however, an extendible hashing algorithm is necessary to dynamically balance the actual workload across islands.
- The island-based design makes it easy to maintain the consistency of the distributed file system, in terms of protocol design as well as amortized cost.
- Performance can scale efficiently with the system size, if and only if cross-island communications are minimized; the overall performance scales efficiently as estimated.
- Background migration of data during an online reconfiguration has a minor impact on client performance, and the reconfiguration can be made transparent to clients by on-demand migration.

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] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", in Proceedings of the 29th ACM Symposium on Theory of Computing, May 1997.

[3] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers", in Proceedings of the 8th International Conference on Architectural Support for Programming Languages and

Operating Systems, October 1998.
[4] P. F. Corbett, and D. G. Feitelson, "The Vesta Parallel File System", in ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.
[5] 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.
[6] 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.
[7] 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.
[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in A Distributed File System", in ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.
[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System", in Proceedings of USENIX Summer Technical Conference, Summer 1985.
[10] J. L. Carter, and M. N. Wegman, "Universal Classes of Hash Functions", in Journal of Computer and System Sciences 18, 1979.
[11] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", in ACM Transactions on Database Systems, Vol. 4 No. 3, 1979.
[12] B. E. Keith, and M. Wittle, "LADDIS: the Next Generation in NFS File Server Benchmarking", in Proceedings of USENIX Summer Technical Conference, June 1993.
[13] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", in Proceedings of ACM SIGCOMM, February 1998.
[14] 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.
[15] Microsoft, the Common Internet File System (CArchipelago) specification reference, 1996.
[16] 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.
[17] H. Custer, “Inside the Windows NT File System”, Microsoft Press, 1994.
[18] R. Vingralek, Y. Breitbart, and G. Weikum, "Distributed File Organization with Scalable Cost/Performance", in Proceedings of ACM SIGMOD, May 1994.
[19] N. Nienwejaar, and D. Kotz, "The Galley Parallel File System", in Proceedings of the International Conference on Supercomputing, July 1996.
[20] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. H. Tetzlaff, “Evaluation of Design Alternatives for a Cluster File System”, in Proceedings of USENIX Winter Technical Conference, Winter 1995.
[21] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg and J. Zelenka, “File Server Scaling with Network-Attached Secure Disks”, in Proceedings of ACM SIGMETRICS, June 1997.
[22] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, in Communications of the ACM, July 1978.
[23] J. Gray, “Notes on Database Operating Systems”, in Operating Systems: An Advanced Course, 1978.
[24] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System", in ACM Transactions on Computer Systems, February 1988.
[25] A. Watson, and P. Benn, "Multiprotocol Data Access: NFS, CIFS, and HTTP", Technical Report 3014, Network Appliance, May 1999.
[26] H. C. Rao, and L. L. Peterson, "Accessing Files in an Internet: the Jade File System", in IEEE Transactions on Software Engineering, IEEE, Vol. 19 No. 6, June 1993.
[27] http://www.viarch.org
[28] 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.
[29] N. Lynch, and M. Tuttle, “An Introduction to Input/Output Automata”, CWI-Quarterly, 2(3), September 1989.
[30] 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.
[31] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, “Detection of Mutual Inconsistency in Distributed Systems”, in IEEE Transactions on Software Engineering 9(3), May 1983.
[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”, in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.
[33] M. Baker, and J. K. Outsterhout, “Availability in the Sprite Distributed File System”, in ACM Operating Systems Review, 25, 2, April 1991.
[34] M. Devarakonda, B. Kish, and A. Mohindra, “Recovery in the Calypso File System”, in ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.
[35] R. Golding, J. Wilkes, and A. Veitch, private communications, August 1999.
[36] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction”, in Proceedings ACM Symposium on Principles of Programming Languages, January 1992.
[37] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors", in Proceedings of the

15th ACM Symposium on Operating Systems and Principles, December 1995.
[38] D. Roselli, and T. E. Anderson, "Characteristics of File System Workloads", Technical Report UCB//CSD-98-1029, 1998, and personal communications, April 1999.
[39] K. W. Shirriff, and J. K. Ousterhout, "A Trace-Driven Analysis of Name and Attribute Caching in A Distributed System", in Proceedings of USENIX Technical Conference, 1992.
[40] Microsoft Corporation, "Platform SDK: Windows Base Services: Files and I/O", in MSDN Library Visual Studio 6.0, 1998.
[41] B. 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.
[42] J. R. Douceur and W. J. Bolosky, "A Large-scale Study of File-system Contents", in Proceedings of ACM SIGMETRICS, May 1999.
[43] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-Performance, Reliable Secondary Storage", ACM Computing Surveys, June 1994.
[44] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, P. M. Friedenbach, "The Implications of Distributed Data in a Commercial Environment on the Design of Hierarchical Storage Management", Performance Evaluation, 1994.

Appendix A: Data loss in non-redundant CFS model

Given the parameters *s* (number of storage servers), *h* (directory tree height), *d* (number of sub directories per directory), *bs* (block size), *f* (number of files per directory), and *fs* (file size), the data loss in a single directory tree with the loss of 1 out of *s* servers is

$$L(h) = \frac{d^h \cdot (d \cdot bs + f \cdot fs) \cdot Q(1-\frac{1}{s}, h+1)}{s \cdot (d-1)}$$

$$- \frac{(d + f \cdot fs) \cdot Q(d \cdot (1-\frac{1}{s}), h+1)}{s \cdot (d-1)}$$

$$+ Q(d \cdot (1-\frac{1}{s}), h) \cdot (1-\frac{1}{s}) \cdot f \cdot fs \cdot (1-(1-\frac{1}{s})^{\lceil \frac{fs}{bs} \rceil}),$$

where $Q(x, y) = \frac{x^y - 1}{x - 1}$.

Appendix B: Expected load imbalances

Assuming that objects *O* are to be distributed to units *U*, we define the *imbalance* I_{OU} as the standard deviation of objects *O* in units *U* divided by the average objects in each unit. I_{OU} is zero if the distribution is perfectly even. Let *B* be the number of buckets, *D* be the number of directories, *W* be the workload, and *S* be the number of islands. We define the variables x_{ij} as

$$\forall i = 1..D, j = 1..B, x_{ij} = \begin{cases} 1, \text{ if directory } i \text{ is hashed to bucket } j; \\ 0, \text{ otherwise.} \end{cases}$$

The bucket size is $Y_j = \sum_{i=1}^{D} x_{ij}$, the expected bucket size is $E[Y_j] = \frac{D}{B}$, and the variance (square of standard deviation) of bucket size is

$$Var[Y_j] = Var[\sum_{i=1}^{D} x_{ij}] = \sum_{i=1}^{D} Var[x_{ij}]$$

$$= \sum_{i=1}^{D} E[(x_{ij} - E[x_{ij}])^2] = \sum_{i=1}^{D} \frac{(1-\frac{1}{B})^2 + (\frac{1}{B})^2 \cdot (B-1)}{B}$$

$$= \frac{D}{B}(1-\frac{1}{B}).$$

The second step of derivation above is based on the property of universal hash functions that x_{ij}'s are pairwise independent. The others are by definitions. The imbalance in directory distribution across buckets is

$$I_{DB} = \frac{\sqrt{Var[Y_j]}}{E[Y_j]} = \sqrt{\frac{B-1}{D}}.$$

We define w_i as the workload in directory *i* and $W = \sum_{i=1}^{D} w_i$. The workload in a directory does not include the loads in its sub directories, hence we can assume that w_i's are pairwise independent. The expected directory workload is $E[w_i] = \frac{W}{D}$, the variance of directory workload is $Var[w_i] = \frac{\sum_{i=1}^{D} w_i^2}{D} - E^2[w_i]$, and the imbalance in workload distribution across directories is $I_{WD} = \sqrt{D \cdot \frac{\sum_{i=1}^{D} w_i^2}{W^2} - 1}$. The workload in bucket *j* is $Z_j = \sum_{i=1}^{D} x_{ij} \cdot w_i$, the expected bucket workload is $E[Z_j] = \frac{W}{B}$, the variance of bucket workload is

$$Var[Z_j] = Var[\sum_{i=1}^{D} x_{ij} \cdot w_i] = \sum_{i=1}^{D} Var[x_{ij} \cdot w_i]$$

$$= \frac{\sum_{i=1}^{D} w_i^2}{B} - \frac{\sum_{i=1}^{D} w_i^2}{B^2},$$

and the load imbalance across buckets is $I_{WB} = \sqrt{(B-1) \cdot \frac{\sum_{i=1}^{D} {w_i}^2}{W^2}}$. We substitute $\frac{\sum_{i=1}^{D} {w_i}^2}{W^2}$ with $\frac{{I_{WD}}^2 + 1}{D}$ and get $I_{WB} = I_{DB} \cdot \sqrt{{I_{WD}}^2 + 1}$.