Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services

Minwen Ji, Edward W. Felten, Randolph Wang, and Jaswinder Pal Singh
Department of Computer Science, Princeton University
{mji, felten, rywang, jps}@cs.princeton.edu

Abstract

Maintaining availability in the face of failures is a critical requirement for Internet services. Existing approaches in cluster-based data storage rely on redundancy to survive a small number of failures, but the system becomes entirely unavailable if more failures occur. We describe an approach that allows a cluster file server to *isolate* failures so that the system can continue to serve most clients. Our approach is complementary to existing redundancy-based methods: redundancy can mask the first few failures, and failure isolation can take over and maintain availability for the majority of clients if more failures occur.

The building blocks of our design are self-contained and load-balanced 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 provides failure isolation because each island can function independently of other islands' failures. It also helps the file system scale with the system and workload sizes because communication and synchronization across islands are reduced. We implemented a prototype island-based file system called *Archipelago* on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The measurement of micro benchmark shows that Archipelago adds little overhead to NTFS and Win32 RPC performance; while the measurement of operation mixes based on NTFS traces shows a speedup of 15.7 on 16 islands.

1. Introduction

NT clusters are an important tool for large I/O-intensive applications such as file servers, Web servers, and other Internet services. A wide variety of research projects on cluster file systems have explored approaches to building cluster file systems that provide high availability and scalability.

This paper discusses a new approach to maximizing availability on a cluster file server. We use the percentage of requests that succeed despite the failure of one or more servers as the availability metric, our goal in this work is maximize this percentage.

There are two complementary approaches to maximizing availability. First, we can use redundancy to maintain complete availability in the face of a small number of failures; second, we can try to *isolate* failures in order to serve as many requests as possible even though some cannot be served. These approaches are complementary, since we can use redundancy to mask the first few failures, and then use isolation to cope with any additional failures.

This paper describes an approach to cluster file system design that provides failure isolation. We divide the nodes in the system into groups called *islands*. An island might be a single node, or it might be a group of nodes that use redundancy within the island to mask failures. In either case, island-based design strives to serve as many client requests as possible when one or more islands have crashed or are unavailable.

The main idea underlying island-based design is the *one-island principle*: as many file system operations as possible should require the participation of exactly one island. The one-island principle provides good failure isolation because each island can function independently of other islands' failures. In other words, the failure of 1 out of *n* islands in an island-based file system renders only 1/n data inaccessible. The one-island principle allows island-based systems to scale efficiently with the system and workload sizes because communication and synchronization across islands are reduced.

Our motivation of failure isolation is analogous to the motivation of fault containment in Hive [26]. Hive, an operating system for large-scale shared-memory multiprocessors, attempts to "contain" a failed part so that it does not bring down other parts.

The target application of an island-based file

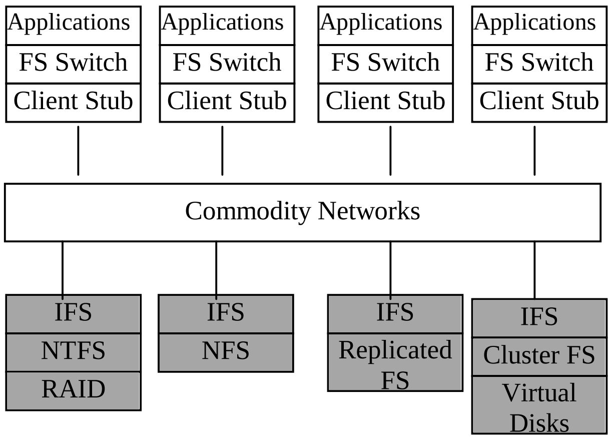

Figure 1. Overview of an island-based file system (IFS). Shaded boxes are islands or servers and non-shaded boxes are clients.

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

We evaluated the island-based design by statistical analysis of the access patterns of existing systems. The results show that the partial availability provided by island-based file system is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients. In one example, if 1 out of 32 islands is down for an hour, we expect that 93.8% clients during that hour will not notice the temporary partial failure. On average 99.8% operations involve a single island and hence do not require communication or synchronization across islands.

We implemented a prototype of island-based file system called *Archipelago* on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The measurement of micro benchmarks shows that Archipelago adds little overhead to NTFS and Win32 RPC performance; the measurement of operation mixes based on NTFS traces shows a speedup of 15.7 on 16 islands.

2. Quantified motivation

To quantitatively motivate the potential advantage of island-based design, let us examine the temporary or permanent data loss ratios under partial failures in existing cluster file systems and island-based file systems with the same redundancy schemes.

We modeled the data loss ratio in case of partial failures in cluster file systems (CFS) built on top of virtual storage layers, such as Frangipani [1] and xFS [4], under various redundancy schemes. The results show that CFS loses a significantly larger portion of data than the virtual storage it loses in a partial failure because the data in a surviving server will be inaccessible if any server containing a piece of metadata needed to access the surviving data fails. For example, with the loss of 1 out of 32 non-redundant virtual storage servers or 3.1% non-redundant virtual storage space, CFS is expected to lose 63.8% data in files and directories. The detailed analytic models can be found in our technical report [14].

We suggest that the temporary or permanent data loss of an existing redundant file system can be reduced by a failure isolation scheme without altering the underlying redundancy scheme. We observe that many existing redundant storage systems are divided into groups and that 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 [4]. By configuring each redundant group as an independent file system, we can always achieve better availability than by running a single file system on top of the whole storage system. We change the data loss example above by assuming that each of the 32 virtual storage servers is a RAID-5 single-parity stripe group of 4 physical nodes and that xFS [4] runs on the 128 nodes as a single file system. If we run an independent xFS in each group, we expect to lose only 3.1% vs. 63.8% data when we lose at least 2 nodes in the same group.

The challenge is how to evenly, automatically and dynamically partition a single large file system into a cluster of independent components without causing inconsistency across components in the face of partial failures.

3. System structure

Figure 1 gives an overview of an island-based file system in a typical configuration. An island consists of a server process running on top of a local file system. Client applications view the island-based file system as a single system and access it through local file system switches and stubs. Islands and clients are connected by commodity local area networks such as Ethernet.

Let us examine two important issues in island-based design, data distribution and metadata replication.

3.1 Hash-based data distribution

We designed a new data distribution strategy for island-based file systems: data is distributed to islands at *directory granularity* by *hashing* the *pathnames* of the directories to island indices.

We choose directory granularity rather than block, file or sub tree granularity because most file system operations involve a single directory and hence satisfy the one-island principle, and directories are finer grained than sub trees so as to allow load balance.

We choose hashing instead of recursive name lookup because hash functions can be computed on the client machines without contacting any servers. We choose to hash pathnames instead of low-level integer identifiers such as inode numbers because pathnames are the only information that a client can possibly have without contacting any servers, and they are independent of internal representations of file systems.

Clients determine which island to contact for a directory or a file in that directory by hashing the full pathname of the directory to an island index in two steps: first, hashing the pathname to a *bucket* (an integer) with a universal hash function [7]; second, hashing the bucket to an island index with an extendible hash table [8]. The universal hash function used in an island-based file system is a consistent mapping from a variable-length character string to a 32-bit integer and has good distribution in the output space independently of the input space. A universal hash function can evenly distribute an arbitrary set of directories to buckets; however, it does not have control on the workload distribution across directories; therefore, an additional level of indirection is necessary to handle the hot spots and dynamic load changes. A subset of the 32 bits is used as the index to the extendible hash table and the table entries are island indices. As load imbalance across islands increases or islands are permanently added or removed during system reconfiguration, the table entries are reassigned to islands to rebalance the load using a bin-packing algorithm. The reassignment is made *monotonic*, i.e. each island either loses data or gains data, but not both.

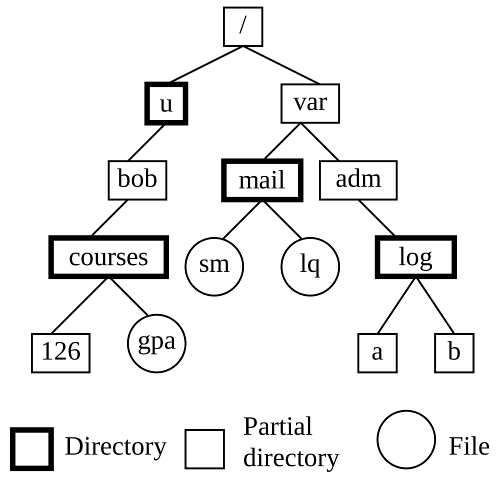

Figure 2. Skeleton hierarchy and directory replication. This is an image of the internal file system in an island that is the directory owner of the highlighted directories. Partial directories are replicas that contain only attributes and partial contents or no contents.

Therefore, only a minimal amount of data needs to be migrated between islands.

Inside each island, we store directories in a *skeleton hierarchy*. We call the file system running inside each island the *internal file system*. An internal file system can be an instance of any existing file system such as a local file system, a replicated file system or a cluster file system. 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 certain metadata or directory attributes.

3.2 Usage-based metadata replication

Although it might not take much space to replicate metadata across islands if it accounts for a small portion of the entire system, updates to replicated metadata will have to be done in all replicas and hence violate one-island principle. Therefore, we use a *usage-based* adaptive replication scheme in the island-based design, i.e. we replicate metadata that is more frequently used to a higher degree.

To help us explain the usage-based metadata replication, we introduce two terms, *directory* *owner* and *parent owner*. The *directory owner* of a 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 will be replicated in its parent owner, in its directory owner and in all the parent owners of its descendent directories. Therefore, the replication scheme can automatically adapt to the usage of the metadata. In particular, the root directory is replicated in all islands; files are not replicated across islands; intermediate directories are replicated to various degrees.

However, only some directory *attributes*, not the directory *contents*, need to be replicated. Directory contents are the lists of names and addresses of sub directories and files. Only the directory owner keeps a complete copy of the directory contents; other replicas have partial contents or no contents. Changes to directory contents, e.g. adding or removing files, need to be done in the directory owner only. Directory attributes include name, size, security, time stamps, read-only tag, compressed tag, etc.. Changes to directory attributes will, however, affect multiple replicas.

We want to replicate only those attributes that are needed when a descendent of the directory is 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. Attributes such as name, security, read-only tag and compressed tag are static. Attributes such as size and time stamps are dynamic. We replicate the static attributes and do not replicate the dynamic attributes. We use a *read-one-write-all* policy to maintain consistency of the static attributes; the overhead of updates is acceptable since static attributes rarely change. We read and write dynamic attributes in a single island, the directory owner.

Figure 2 gives an illustration of the skeleton hierarchy and metadata replication.

3.3 Evaluation

We evaluated the load balance and storage overhead in island-based file systems by statistical analysis of the contents of existing systems. Detailed measurements and analysis can be found in our technical report [14]. We summarize the results as follows:

- Only a small portion of storage is needed for replicating directory attributes (0.1% to 0.5% per island or 0.3% to 7.7% in total in our experiments).
- Load imbalance (average number of bytes per island dividing its standard deviation) resulted from the hashing algorithm in island-based file systems is low (0.0001 to 0.0279 in our experiments) in spite of the unbalanced load across directories or hot spots.

4. Protocols and other design issues

To make the island-based design a viable solution, we need to address the issues of rebalance, consistency, recovery, etc. in addition to data distribution and metadata replication. We use standard approaches that are tailored to island-based file systems, as we will briefly describe below.

4.1 Rebalance protocol

As discussed in the previous section, the hash function in data distribution can be changed to rebalance the load across islands when load imbalance exceeds a threshold or when islands are permanently added to or removed during system reconfiguration.

We use a two-phase commit protocol [16] in the rebalance procedure so that the hash table is updated in all islands atomically in the face of partial failures. In the first phase, load information (number of bytes) is collected from all islands and all islands are prepared for the rebalance. In the second phase, a new hash table is computed according to the load information, and is either updated in all islands or aborted if any island is inaccessible.

The hash table is replicated in all clients of the file system as well as in all islands. The table has an entry per directory bucket and the number of buckets is a constant factor of the number of islands; therefore, the table size is proportional to the number of islands. (The universal hash function can map multiple directories to the same bucket. See Section 3.1.) A client is asked to update its hash table when any server detects its out-of-date copy using piggy-back information in regular operations.

How often the rebalance procedure needs to be invoked depends on the load imbalance that can be tolerated. We expect that a reasonable threshold can be set so that the rebalance procedure is invoked at a non-disruptive frequency, e.g. once every weekend.

A trace-driven study of the online reconfiguration of a web server running on an island-based file system shows that data migration in the rebalance procedure is made transparent to the web server in terms of both functionality and performance [14]. Therefore, we do not expect the rebalance procedure to have a noticeable impact on client operations.

4.2 Consistency protocol

Since certain states, e.g. static directory attributes, are replicated across islands, a cross-island protocol is necessary to keep the replicas consistent in the face of island failures and network partitions. Cross-island operations in island-based file systems include *CreateDir* and *RemoveDir*, which involve two islands, *SetDirAttr*, *SymLinkDir* and *DeleteLinkDir*, which involve all islands, and *RenameDir*, which involves a variable number of islands depending on the directory to be renamed.

The island-based design eases the consistency maintenance in two ways. First, the majority of operations involve a single island, hence do not require a cross-island protocol for consistency. Second, all cross-island operations on the same object are coordinated by a single island, i.e. the directory or parent owner; 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, hence largely relaxes the synchronization semantics. We designed and implemented a protocol that uses logical clock synchronization [15], logging [10] and two-phase commit [16] for atomicity and serialization of cross-island operations. In particular, we choose to maintain the following invariants in the face of island failures and network partitions:

1. All operations on the same object are serialized, i.e. 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.
4. The ordering relations 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.

4.3 Recovery protocol

We designed and implemented a fairly standard recovery protocol for islands to recover from various combinations of failures back to consistent state.

Cross-island operations are logged on disk if they cannot be committed in all involved islands due to island failures or network partitions. A failed or disconnected island will exchange logs with other islands upon reconnection to those islands. In particular, we choose to maintain the following invariants in the state transitions of a recovering island *r*:

1. All logged operations from other islands will be committed in *r* in the ascending order of their time stamps. That is, operations serialized in real time will be committed 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 logged operations have been committed in *r*. That is, the inconsistent state of *r*, if there is any, is made invisible to clients.

4.4 Other design issues

Island-based file systems inherit most functions from their internal file systems, such as metadata structures, disk space allocation, I/O scheduling, server-side caching, locking, local security, recovery, etc.; therefore, we are not concerned about all the low-level details in file system design and implementation. We extended certain functions, such as symbolic links and renaming directories, to adapt to the island-based environment. Interested readers should refer to our technical report for more information about the design, implementation, and evaluation of our prototype [14].

5. Implementation

We have implemented a prototype of island-based file system called *Archipelago* on a cluster of Pentium II PCs running Windows NT 4.0. NTFS [13] is used as the internal file system. NTFS uses extensive caching and name indexing for better performance and logs metadata changes for local recoverability. NTFS can be configured to run on a group of disks with parity striping for data redundancy.

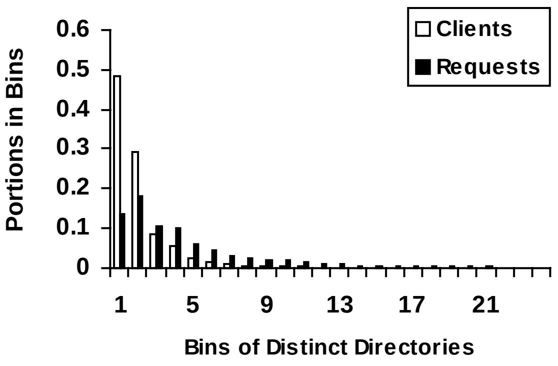

Figure 3. Histograms of clients and requests by bins of distinct directories in the web traces. The numbers read as "48.3% clients accessed 1 distinct directory during every hour" or "17.9% requests were issued by clients who accessed 2 distinct directories during every hour". Accesses to more than 24 directories account for 0.4% clients and 19.3% requests in total, and are omitted in the graph for readability.

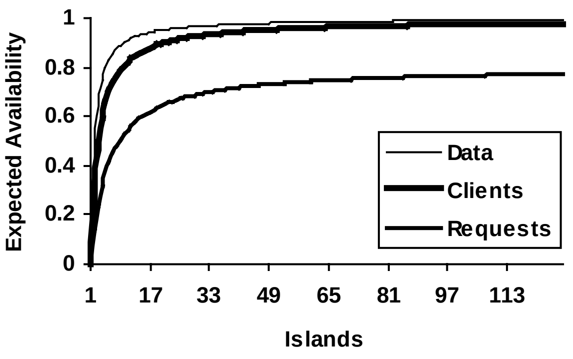

Figure 4. Expected availability for data, clients and requests in the web traces with the failure of 1 out of *n* islands. The x axis is the number *n* of islands. The y axis is the expected availability, i.e. (1-1/*n*) for data and $\sum p(i) \cdot (1-1/n)^i$ for clients and requests, where *i* is the bin of distinct directories and *p(i)* is the portion of clients or requests in the bin *i*.

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 server is implemented as a user-level process. For expediency, our prototype client is implemented as a stub .dll that redirects requests for Archipelago files directly to servers, bypassing the in-kernel file system drivers. This solution is adequate for experimental purposes, although it does not provide total seamless integration with existing applications. A more complete solution would implement a full installable file system driver [20]. We believe the performance difference in these two solutions to be negligible compared with the time to service file system requests in a distributed file system.

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.

6. Measurements

In this section, we present the selected measurements to answer the following questions.

1. How many clients will likely notice a partial failure in an island-based system? (Section 6.1)
2. What is the overhead of island-based design in simple cases? (Section 6.2)
3. How many operations require cross-island communication and synchronization? (Section 6.3)
4. How do cross-island operations affect the overall scalability of an island-based file system? (Section 6.4)

6.1 Impact of partial availability on web clients

The effective availability of an island-based file system with partial failures depends on the number of distinct directories that clients access because a partial failure in the system causes a random set of directories to be inaccessible.

We compute the histograms of clients and requests by the distinct directories they touched from the access logs of the web server running on our site [23]. We assume that the island-based file system 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 group the HTTP requests into clients by the hostnames or IP addresses in the requests, and within each client, we group requests into directories by the URLs in the requests. We compute 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), using a time window size of an hour. The results, in Figure 3, show that the largest portion (48.3%) of clients accessed only

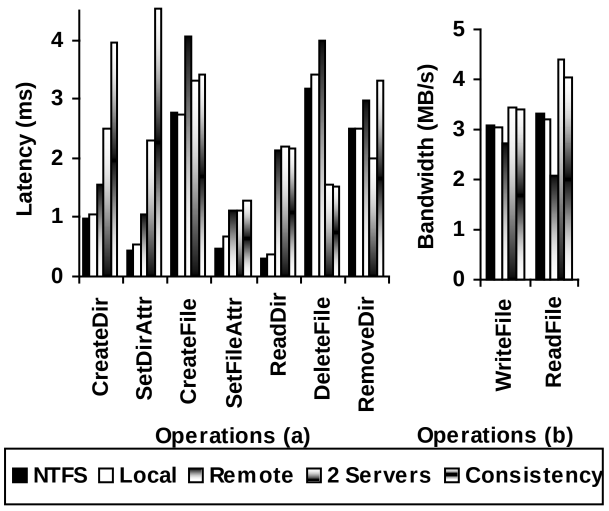

Figure 5. Single client performance. A single client runs the micro benchmarks in five cases: directly on NTFS (NTFS), on the local machine of an Archipelago server (Local), on a remote machine from the server (Remote), with two servers (2 Servers), and with the consistency protocol turned on with two servers (Consistency), respectively. The y-axis in (a) is the latency in milliseconds measured at the client side. Lower columns represent better performance. The y-axis in (b) is the bandwidth in MB/s in the WriteFile and ReadFile operations measured at the client side. Higher columns represent better performance.

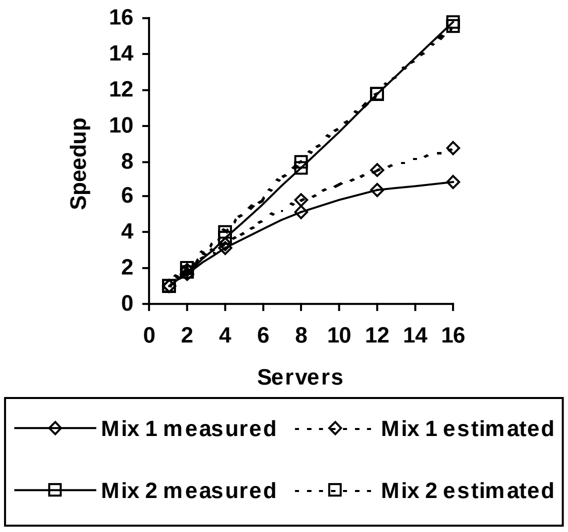

Figure 6. Speedup of throughputs of randomized operation mixes. The four curves are the measured speedup of operation mix 1 (Table 2), estimated speedup of operation mix 1, measured speedup of operation mix 2 (Table 2), and estimated speedup of operation mix 2, respectively. The speedup is calculated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

1 distinct directory during every hour and the largest portion (17.9%) of requests were issued by clients who accessed 2 distinct directories during every hour. Requests are more scattered across bins in the histogram because larger bins have more accesses and hence more weights. We computed the histograms by dividing the traces into other time windows ranging from 30 minutes to 8 hours, but there was no significant difference across time windows.

Given the statistics of distinct directories, we compute the expected availability of the island-based file system for data, clients and requests, respectively, shown in Figure 4. 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 the system causes a random set of directories to be inaccessible. For example, if 1 out of 32 islands is down for an hour, we expect that 93.8% clients of the web server during that hour will not notice the temporary partial failure.

6.2 Single client performance

In this section, we present the results of running single client micro benchmarks on Archipelago in various configurations. The machines used in our experiments have Pentium II 300 MHz processors, 128 MB main memories and 6.4 GB Quantum Fireball IDE hard disks for use by Archipelago. The PCs are connected by a 3COM SuperStack II 100Mbps Ethernet hub. The PCs run Windows NT Workstation 4.0 and the hard disks for Archipelago are formatted in NTFS.

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

We ran the micro benchmarks with a single client in five cases: directly on NTFS (NTFS), on the local machine of an Archipelago server (Local), on a remote machine from the server (Remote), with two servers (2 Servers), and with the consistency protocol turned on with two servers (Consistency), respectively. Figure 5 shows the bandwidth in WriteFile and ReadFile and the response times in other operations, all measured at the client side.

The difference between the NTFS and Local cases is caused by the overhead of computing hash functions. This overhead is low compared to the operation time itself. The difference between the Local and Remote cases is caused by the communication overhead (Win32 RPC on TCP/IP and 100 Mbps Ethernet) between the client and the server, i.e. 0.48 ms latency and 8.67 MB/s bandwidth in our experiments. There are two causes for the difference between the Remote and 2-Server cases: the cross-island operations such as CreateDir and SetDirAttr involve an additional server in the latter case and there was more total file system buffer cache in the latter case. The difference between the 2-Server and Consistency cases is caused by the overhead of the consistency protocol.

The results show that island-based design adds little overhead to NTFS and Win32 RPC performance and that the consistency protocol slows down the cross-island operations but does not have a noticeable impact on one-island operations.

We ran the same micro benchmarks with 1 to 16 servers and clients. The results, not shown here [14], indicate that the one-island operations scale linearly with the system and workload sizes. Two-island operations scale less efficiently and all-island operations do not scale because the consistency protocol requires 2**k* uni-cast messages per cross-island operation, where *k* is the number of islands involved in the operation. Therefore, the overall scalability depends on the actual operation breakdown.

6.3 Operation breakdown in NTFS traces

Previous studies of file system traces indicated that the cross-island operations are rare [17] [9] [18]. However, it is well known that file access patterns are highly dependent on the operating systems where the traces were taken. Since we implement Archipelago on Windows NT as opposed to UNIX, in which the Sprite and NFS traces were taken, we feel it important to study the file access patterns in NTFS. We choose 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 [19]. We present the events in Table 1 and infer the operation breakdown from them.

No.	Events	Avg.	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 1. Percentages of file system events in NTFS traces. Row 1 (Total events) shows the total number of events in all traces. Rows 2 through 19 show the percentage of each event. Shaded events correspond to cross-island operations in island-based file systems. 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. The names "FileLink" and "DirLink" refer to symbolic links (*shortcuts* in NT) to files and directories, respectively.

Table 1 shows that, on average, one-island operations account for 99.8% of total operations. The slow operations in island-based file systems, e.g. setting directory attributes, renaming directories, creating symbolic links (shortcuts) to directories, are rare.

The section below shows how the cross-island operations affect the overall scalability of the system, given the measured breakdown in this section.

6.4 Scalability of operation mixes

We run a benchmark of randomized operation mixes to measure the overall scalability of Archipelago. The benchmark is extended from the SPEC SFS or LADDIS benchmark [9]. 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 in NTFS, as shown in the previous section. The experiment environment and configuration are the same as in Section 6.2.

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 directories, 2000 files, and 100 symbolic links shared by all clients, and the same numbers of private objects (directories, files and symbolic links) per client. The client repeatedly does an operation that is randomly chosen at specified frequencies. For each operation, the client randomly chooses an object, either from the existing shared or private objects, or by generating a new name in an existing directory, depending on the operation. The WriteFile operation writes a random number (chosen from 0 to 1 MB) of bytes to the file; both WriteFile and ReadFile operations transfer up to 8KB per request so that the operation time is comparable to those of other operations. Each client maintains its own view of the shared objects and its private objects, but does not synchronize with other clients on the creation and deletion of the shared objects. Therefore, an operation on a shared object might fail if it conflicts with a previous operation on the same object from another client [9]. 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. Note that more WriteFile than ReadFile events are recorded in the NTFS traces because reads that hit in cache cannot be captured by ReadDirectoryChangesW.

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 2 shows the recorded operation mixes and Figure 6 shows both the measured speedups and estimated speedups. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup with *n* servers is *n/(1+overhead_per_operation)*, where the *overhead per operation* is the total number of server-to-server RPCs divided by the total number of successful client operations.

	Mix 1 (%)	Mix 2 (%)
CreateDir	0.9297	0.0522
CreateFile	4.0314	3.5661
DeleteFile	2.7731	2.4353
DeleteLinkDir	0.9850	0.0128
ReadDir	14.4505	15.6528
ReadFile	14.1343	15.2778
RemoveDir	0.7543	0.0162
ResolveLinkDir	1.7205	0.1014
SetDirAttr	1.0383	0.0713
SetFileAttr	26.6085	29.2835
SymLinkDir	1.0089	0.0109
WriteFile	31.5656	33.5194
Successful	45360 to 309960	48042 to 756120
Total	48042 to 325534	48043 to 780260
Throughput (requests/sec)	75.6 to 516.6	80.07 to 1260.2

Table 2. Operation mixes. Each percentage in this table is the number of successful requests on each operation divided by the total number of successful requests, averaged over 1 to 16 clients and servers. The total numbers of requests and throughputs grow with the numbers of clients and servers for the fixed 10 minutes period; the ranges are shown in the last three rows in the table.

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

6.5 Implications for larger scale systems

Given the percentages of one-island (*P*1), two-island (*P*2) and all-island (*Pa*) operations, where *P*1+*P*2+*Pa*=1, we can predict the speedup efficiency at large scale with an analytic 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 the overhead per operation is the average number of server-to-server RPCs per operation and equals (2-1)*2**P*2+(n-1)*2**Pa*. (The factor 2 results from the two-phase commit protocol.) 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 [18]. The resulting percentages are *P*1=99.768%, *P*2=0.161% and *Pa*=0.071%. From the speedup efficiency model above, we predict that the system can scale up to 702 islands while maintaining the efficiency higher than 50%; that is, an island-based file system can achieve a higher speedup than 351 with 702 islands.

While such a large cluster is not currently available to us for experiments, our measurement results on the small cluster are encouraging and we are seeking external resources for further scalability tests.

6.6 Discussion

Although the target applications of an island-based file system are Internet services, we use a more generic benchmark in the scalability measurements. Our purpose of those measurements is to learn the impact of cross-island operations on the overall scalability of an island-based file system, but web access logs only give file-reading operations. We do not model in our benchmark the self similarity or hot spots in web accesses because it is not clear whether the same patterns will necessarily show up in disk accesses if web requests can be processed with data in the main memory cache of web servers or file system clients.

7. Related work

Existing file systems designed for high availability, such as Coda [28] and Ficus [29], replicate data across servers. Our approach in island-based file system, i.e. failure isolation, is complementary to the data redundancy approach for high availability. Client caching is extensively used in distributed file systems like Coda [28], Andrew [5] and Sprite [17] to support disconnected operations and to reduce traffics to servers. Similar to server replication, client caching improves availability by data redundancy, i.e. by replicating data in clients. It also improves scalability by reducing server load so that the same number of servers can serve a larger number of clients gracefully. Our scalability goal in island-based file system is to achieve efficient speedup when servers are added to the cluster, which is orthogonal to the goal of client caching. We have not implemented client caching in Archipelago, but we do not expect the island-based design to add any difficulty to such implementation.

State-of-the-art cluster file systems like Frangipani [1] and xFS [4] achieve high reliability and scalability by data redundancy. A fast system area network such as ATM is typically used in those cluster file systems for aggressive communications across data replicas. The majority of operations in island-based file systems do not require communication or synchronization across islands; therefore, an island-based file system can scale efficiently with commodity networks such as Ethernet. The ideal configuration for maximal reliability, availability and scalability is to run an island-based file system with a file system like Frangipani or xFS inside each island.

In terms of failure isolation, cross-node communications, locality and leveraging functions in local file systems, island-based file systems are comparable to distributed file systems like NFS [6], JetFS [12] and CIFS [11]. However, those systems do not share with island-based file systems scalability, load balance, and/or automatic data partitioning and reconfiguration.

In Teradata [27], two orthogonal hash functions are used to map data items to two nodes. In an island-based file system, each data item is mapped to a single island but redundancy might be used inside the island. The Teradata approach offers better load balance when a single node fails, but the failures of two nodes always render a portion of data inaccessible. Our approach makes most operations involve a single island, isolates failures across islands, and does not lose data unless all replicas in the same island fail.

A large scale Internet service typically consists of three logical tiers: request distribution tier, service-specific processing tier and data storage tier. The Locality-Aware Request Distribution (LARD) [3] is a solution to locality and load balance in the distribution tier. The Cluster-Based Scalable Network Services (SNS) [21] [22] provides a programming model for the processing tier. In particular, the authors proposed application decomposition and orthogonal mechanism for graceful degradation during partial failures. Island-based design addresses failure isolation, locality and load balance in the storage tier. While the distribution-tier and processing-tier approaches suffice for read-mostly access patterns and weak consistency requirements, a robust and scalable storage tier is necessary for services with read-write access patterns and strong consistency requirements, such as shared calendar services and online shopping sites. The combination of the approaches in all three tiers can potentially achieve high availability and scalability for Internet services with a wide range of access patterns and consistency requirements.

Commercial web content distributors such as Akamai [24] [2] and Sandpiper [25] provide geographically distributed replication services to read-mostly web contents so that the latency in delivering contents to clients can be reduced. We are focused on improving the availability and scalability of local sites with read-write patterns. Their approach and ours are complementary to each other in improving the overall availability and scalability of Internet services.

Our main contributions are:

1. We address the availability and scalability issues for Internet services in the data storage tier.
2. Our approach to availability and scalability is isolating failures and reducing communication.
3. We achieve failure isolation and reduced communication by enforcing a one-island principle in hash-based data distribution and usage-based metadata replication.

8. Future work and conclusion

NT farms are a fact of life -- people are already using them to provide scalable services. An important question for people who are running all those NT farms to understand is how to structure the cluster in a way that can both balance loads and isolate failures without having to reinvent a distributed file system from scratch, which is a very difficult endeavor, by leveraging as much as possible from the existing NT infrastructure. Our experience suggests that this is indeed possible.

We designed an island-based file system as the data storage for highly available and scalable Internet services. We evaluated the design by statistical analysis of the access patterns in existing systems. We implemented Archipelago, a prototype of the island-based file system, and studied the performance of Archipelago in micro benchmarks and operation mixes.

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:

- The failure isolation provided by island-based file systems is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients.
- An island-based file system can scale well with the system and workload sizes because the majority of operations do not require communication or synchronization across islands.

9. Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, David C. Steere, for their valuable comments and suggestions. A number of theoreticians including Sanjeev Arora, Yaoyun Shi and Amit Chakrabarti helped us with the

mathematic modeling parts in the project.

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] 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.

[5] 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.

[6] 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.

[7] J. L. Carter, and M. N. Wegman, "Universal Classes of Hash Functions", in Journal of Computer and System Sciences 18, 1979.

[8] 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.

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

[10] 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.

[11] Microsoft, the Common Internet File System (CIFS) specification reference, 1996.

[12] 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.

[13] H. Custer, "Inside the Windows NT File System", Microsoft Press, 1994.

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

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

[16] J. Gray, "Notes on Database Operating Systems", in Operating Systems: An Advanced Course, 1978.

[17] 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.

[18] D. Roselli, and T. E. Anderson, "Characteristics of File System Workloads", Technical Report UCB//CSD-98-1029, 1998, and personal communications, April 1999.

[19] Microsoft Corporation, "Platform SDK: Windows Base Services: Files and I/O", in MSDN Library Visual Studio 6.0, 1998.

[20] Microsoft Corporation, "Windows NT IFS Kit", Early Release, March 1997.

[21] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services", in Proceedings of the 16th ACM Symposium on Operating Systems Principles, Octobor 1997.

[22] A. Fox, and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems", in Proceedings of HotOS-VII, March 1999.

[23] http://www.cs.princeton.edu

[24] http://www.akamai.com

[25] http://www.sandpiper.com

[26] 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.

[27] DBC/1012 database computer system manual release 2.0. Technical Report Document No. C10-0001-02, Teradata Corporation, Nov 1985.

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

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