Ivy: A Read/Write Peer-to-Peer File System

Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen
{athicha, rtm, thomer, benjie}@lcs.mit.edu
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139.

Abstract

Ivy is a multi-user read/write peer-to-peer file system. Ivy has no centralized or dedicated components, and it provides useful integrity properties without requiring users to fully trust either the underlying peer-to-peer storage system or the other users of the file system.

An Ivy file system consists solely of a set of logs, one log per participant. Ivy stores its logs in the DHash distributed hash table. Each participant finds data by consulting all logs, but performs modifications by appending only to its own log. This arrangement allows Ivy to maintain meta-data consistency without locking. Ivy users can choose which other logs to trust, an appropriate arrangement in a semi-open peer-to-peer system.

Ivy presents applications with a conventional file system interface. When the underlying network is fully connected, Ivy provides NFS-like semantics, such as close-to-open consistency. Ivy detects conflicting modifications made during a partition, and provides relevant version information to application-specific conflict resolvers. Performance measurements on a wide-area network show that Ivy is two to three times slower than NFS.

1 Introduction

This paper describes Ivy, a distributed read/write network file system. Ivy presents a single file system image that appears much like an NFS [33] file system. In contrast to NFS, Ivy does not require a dedicated server; instead, it stores all data and meta-data in the DHash [9] peer-to-peer block storage system. DHash can distribute and replicate blocks, giving Ivy the potential to be highly available. One possible application of Ivy is to support distributed projects with loosely affiliated participants.

Building a shared read-write peer-to-peer file system poses a number of challenges. First, multiple distributed writers make maintenance of consistent file system meta-data difficult. Second, unreliable participants make locking an unattractive approach for achieving meta-data consistency. Third, the participants may not fully trust each other, or may not trust that the other participants' machines have not been compromised by outsiders; thus there should be a way to ignore or un-do some or all modifications by a participant revealed to be untrustworthy. Finally, distributing file-system data over many hosts means that the system may have to cope with operation while partitioned, and may have to help applications repair conflicting updates made during a partition.

Ivy uses logs to solve the problems described above. Each participant with write access to a file system maintains a log of changes they have made to the file system. Participants scan all the logs (most recent record first) to look up file data and meta-data. Each participant maintains a private snapshot to avoid scanning all but the most recent log entries. The use of per-participant logs, instead of shared mutable data structures, allows Ivy to avoid using locks to protect meta-data. Ivy stores its logs in DHash, so a participant's logs are available even when the participant is not.

Ivy resists attacks from non-participants, and from corrupt DHash servers, by cryptographically verifying the data it retrieves from DHash. An Ivy user can cope with attacks from other Ivy users by choosing which other logs to read when looking for data, and thus which other users to trust. Ignoring a log that was once trusted might discard useful information or critical meta-data; Ivy provides tools to selectively ignore logs and to fix broken meta-data.

Ivy provides NFS-like file system semantics when the underlying network is fully connected. For example, Ivy provides close-to-open consistency. In the case of network partition, DHash replication may allow participants to modify files in multiple partitions. Ivy's logs contain version vectors that allow it to detect conflicting updates after partitions merge, and to provide version information to application-specific conflict resolvers.

The Ivy implementation uses a local NFS loop-back server [22] to provide an ordinary file system interface. Performance is within a factor of two to three of NFS. The main performance bottlenecks are network latency and the cost of generating digital signatures on data stored in DHash.

This paper makes three contributions. It describes a read/write peer-to-peer storage system; previous peer-to-peer systems have supported read-only data or data writeable by a single publisher. It describes how to design a distributed file system with useful integrity properties based on a collection of untrusted components. Finally, it explores the use of distributed hash tables as a building-block for more sophisticated systems.

Section 2 describes Ivy's design. Section 3 discusses the consistency semantics that Ivy presents to applications. Section 4 presents tools for dealing with malicious participants. Sections 5 and 6 describe Ivy's implementation and performance. Section 7 discusses related work, and Section 8 concludes.

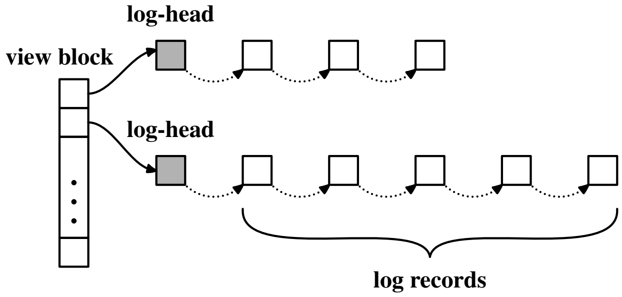

Figure 1: Example Ivy view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

2 Design

An Ivy file system consists of a set of logs, one log per participant. A log contains all of one participant's changes to file system data and meta-data. Each participant appends only to its own log, but reads from all logs. Participants store log records in the DHash distributed hash system, which provides per-record replication and authentication. Each participant maintains a mutable DHash record (called a *log-head*) that points to the participant's most recent log record. Ivy uses version vectors [27] to impose a total order on log records when reading from multiple logs. To avoid the expense of repeatedly reading the whole log, each participant maintains a private snapshot summarizing the file system state as of a recent point in time.

The Ivy implementation acts as a local loop-back NFS v3 [6] server, in cooperation with a host's in-kernel NFS client support. Consequently, Ivy presents file system semantics much like those of an NFS v3 file server.

2.1 DHash

Ivy stores all its data in DHash [9]. DHash is a distributed peer-to-peer hash table mapping keys to arbitrary values. DHash stores each key/value pair on a set of Internet hosts determined by hashing the key. This paper refers to a DHash key/value pair as a DHash block. DHash replicates blocks to avoid losing them if nodes crash.

DHash ensures the integrity of each block with one of two methods. A *content-hash block* requires the block's key to be the SHA-1 [10] cryptographic hash of the block's value; this allows anyone fetching the block to verify the value by ensuring that its SHA-1 hash matches the key. A *public-key block* requires the block's key to be a public key, and the value to be signed using the corresponding private key. DHash refuses to store a value that does not match the key. Ivy checks the authenticity of all data it retrieves from DHash. These checks prevent a malicious or buggy DHash node from forging data, limiting it to denying the existence of a block or producing a stale copy of a public-key block.

Ivy participants communicate only via DHash storage; they don't communicate directly with each other except when setting up a new file system. Ivy uses DHash content-hash blocks to store log records. Ivy stores the DHash key of a participant's most recent log record in a DHash block called the log-head; the log-head is a public-key block, so that the participant can update its value without changing its key. Each Ivy participant caches content-hash blocks locally without fear of using stale data, since content-hash blocks are immutable. An Ivy participant does not cache other participants' log-head blocks, since they may change.

Ivy uses DHash through a simple interface: `put(key, value)` and `get(key)`. Ivy assumes that, within any given network partition, DHash provides write-read consistency; that is, if `put(k, v)` completes, a subsequent `get(k)` will yield `v`. The current DHash implementation does not guarantee write-read consistency; however, techniques are known which can provide such a guarantee with high probability [19]. These techniques require that DHash replicate data and update it carefully, and might significantly decrease performance. Ivy operates best in a fully connected network, though it has support for conflict detection after operating in a partitioned network (see Section 3.4).

Ivy would in principle work with other distributed hash tables, such as PAST [32], CAN [29], Tapestry [41], or Kademlia [21].

2.2 Log Data Structure

An Ivy log consists of a linked list of immutable log records. Each log record is a DHash content-hash block. Table 1 describes fields common to all log records. The `prev` field contains the previous record's DHash key. A participant stores the DHash key of its most recent log record in its log-head block. The log-head is a public-key block with a fixed DHash key, which makes it easy

Field	Use
`prev`	DHash key of next oldest log record
`head`	DHash key of log-head
`seq`	per-log sequence number
`timestamp`	time at which record was created
`version`	version vector

Table 1: Fields present in all Ivy log records.

for other participants to find.

A log record contains information about a single file system modification, and corresponds roughly to an NFS operation. Table 2 describes the types of log records and the type-specific fields each contains.

Log records contain the minimum possible information to avoid unnecessary conflicts from concurrent updates by different participants. For example, a `Write` log record contains the newly written data, but not the file's new length or modification time. These attributes cannot be computed correctly at the time the `Write` record is created, since the true state of the file will only be known after all concurrent updates are known. Ivy computes that information incrementally when traversing the logs, rather than storing it explicitly as is done in UNIX i-nodes [30].

Ivy records file owners and permission modes, but does not use those attributes to enforce permissions. A user who wishes to make a file unreadable should instead encrypt the file's contents. A user should ignore the logs of people who should not be allowed to write the user's data.

Ivy identifies files and directories using 160-bit i-numbers. Log records contain the i-number(s) of the files or directories they affect. Ivy chooses i-numbers randomly to minimize the probability of multiple participants allocating the same i-number for different files. Ivy uses the 160-bit i-number as the NFS file handle.

Ivy keeps log records indefinitely, because they may be needed to help recover from a malicious participant or from a network partition.

2.3 Using the Log

For the moment, consider an Ivy file system with only one log. Ivy handles non-updating NFS requests with a single pass through the log. Requests that cause modification use one or more passes, and then append one or more records to the log. Ivy scans the log starting at the most recently appended record, pointed to by the log-head. Ivy stops scanning the log once it has gathered enough data to handle the request.

Ivy appends a record to a log as follows. First, it creates a log record containing a description of the update, typically derived from arguments in the NFS request. The new record's `prev` field is the DHash key of the most recent log record. Then, it inserts the new record into DHash, signs a new log-head that points to the new log record, and updates the log-head in DHash.

The following text describes how Ivy uses the log to perform selected operations.

File system creation. Ivy builds a new file system by creating a new log with an `End` record, an `Inode` record with a random i-number for the root directory, and a log-head. The user then mounts the local Ivy server as an NFS file system, using the root i-number as the NFS root file handle.

File creation. When an application creates a new file, the kernel NFS client code sends the local Ivy server an NFS CREATE request. The request contains the directory i-number and a file name. Ivy appends an `Inode` log record with a new random i-number and a `Link` record that contains the i-number, the file's name, and the directory's i-number. Ivy returns the new file's i-number in a file handle to the NFS client. If the application then writes the file, the NFS client will send a WRITE request containing the file's i-number, the written data, and the file offset; Ivy will append a `Write` log record containing the same information.

File name lookup. System calls such as `open()` that refer to file names typically generate NFS LOOKUP requests. A LOOKUP request contains a file name and a directory i-number. Ivy scans the log to find a `Link` record with the desired directory i-number and file name, and returns the file i-number. However, if Ivy first encounters a `Unlink` record that mentions the same directory i-number and name, it returns an NFS error indicating that the file does not exist.

File read. An NFS READ request contains the file's i-number, an offset within the file, and the number of bytes to read. Ivy scans the log accumulating data from `Write` records whose ranges overlap the range of the data to be read, while ignoring data hidden by `SetAttr` records that indicate file truncation.

File attributes. Some NFS requests, including GETATTR, require Ivy to include file attributes in the reply. Ivy only fully supports the file length, file modification time ("mtime"), attribute modification time ("ctime"), and link count attributes. Ivy computes these attributes incrementally as it scans the log. A file's length is determined by either the write to the highest offset since the last truncation, or by the last truncation. Mtime is determined by the `timestamp` in the most recent relevant log record; Ivy must return correct time attributes because NFS client cache consistency depends on it. Ivy computes the number of links to a file by counting the number of relevant `Link` records not canceled by `Unlink` and `Rename` records.

Type	Fields	Meaning
`Inode`	type (file, directory, or symlink), i-number, mode, owner	create new inode
`Write`	i-number, offset, data	write data to a file
`Link`	i-number, i-number of directory, name	create a directory entry
`Unlink`	i-number of directory, name	remove a file
`Rename`	i-number of directory, name, i-number of new directory, new file name	rename a file
`Prepare`	i-number of directory, file name	for exclusive operations
`Cancel`	i-number of directory, file name	for exclusive operations
`SetAttrs`	i-number, changed attributes	change file attributes
`End`	none	end of log

Table 2: Summary of Ivy log record types.

Directory listings. Ivy handles READDIR requests by accumulating all file names from relevant `Link` log records, taking more recent `Unlink` and `Rename` log records into account.

2.4 User Cooperation: Views

When multiple users write to a single Ivy file system, each source of potentially concurrent updates must have its own log; this paper refers to such sources as participants. A user who uses an Ivy file system from multiple hosts concurrently must have one log per host.

The participants in an Ivy file system agree on a *view*: the set of logs that comprise the file system. Ivy makes management of shared views convenient by providing a *view block*, a DHash content-hash block containing pointers to all log-heads in the view. A view block also contains the i-number of the root directory. A view block is immutable; if a set of users wants to form a file system with a different set of logs, they create a new view block.

A user names an Ivy file system with the content-hash key of the view block; this is essentially a self-certifying pathname [23]. Users creating a new file system must exchange public keys in advance by some out-of-band means. Once they know each other's public keys, one of them creates a view block and tells the other users the view block's DHash key.

Ivy uses the view block key to verify the view block's contents; the contents are the public keys that name and verify the participants' log-heads. A log-head contains a content-hash key that names and verifies the most recent log record. It is this reasoning that allows Ivy to verify it has retrieved correct log records from the untrusted DHash storage system. This approach requires that users exercise care when initially using a file system name; the name should come from a trusted source, or the user should inspect the view block and verify that the public keys are those of trusted users. Similarly, when a file systems' users decide to accept a new participant, they must all make a conscious decision to trust the new user and to adopt the new view block (and newly named file system). Ivy's lack of support for automatically adding new users to a view is intentional.

2.5 Combining Logs

In an Ivy file system with multiple logs, a participant's Ivy server consults all the logs to find relevant information. This means that Ivy must decide how to order the records from different logs. The order should obey causality, and all participants with the same view should choose the same order. Ivy orders the records using a version vector [27] contained in each log record.

When an Ivy participant generates a new log record, it includes two pieces of information that are later used to order the record. The `seq` field contains a numerically increasing sequence number; each log separately numbers its records from zero. The version vector field contains a tuple *U:V* for each log in the view (including the participant's own log), summarizing the participant's most recent knowledge of that log. *U* is the DHash key of the log-head of the log being described, and *V* is the DHash key of that log's most recent record. In the following discussion, a numeric *V* value refers to the sequence number contained in the record pointed to by a tuple.

Ivy orders log records by comparing the records' version vectors. For example, Ivy considers a log record with version vector *(A:5 B:7)* to be earlier in time than a record with version vector *(A:6 B:7)*: the latter vector implies that its creator had seen the record with *(A:5 B:7)*. Two version vectors *u* and *v* are *comparable* if and only if $u < v$ or $v < u$ or $u = v$. Otherwise, *u* and *v* are *concurrent*. For example, *(A:5 B:7)* and *(A:6 B:6)* are concurrent.

Simultaneous operations by different participants will result in equal or concurrent version vectors. Ivy orders equal and concurrent vectors by comparing the public keys of the two logs. If the updates affect the same file, perhaps due to a partition, the application may need to take special action to restore consistency; Section 3 ex-

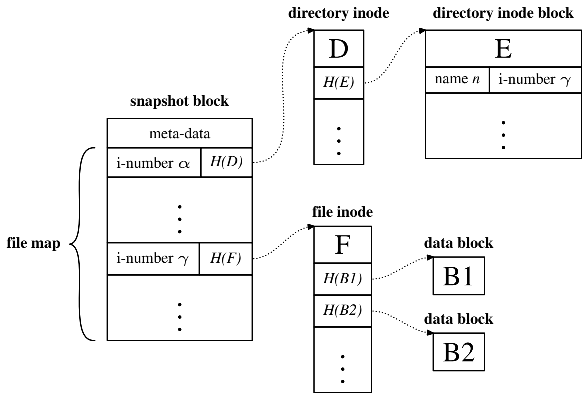

Figure 2: Snapshot data structure. $H(A)$ is the DHash content-hash of A.

plores Ivy's support for application-specific conflict resolution.

Ivy could have used a simpler method of ordering log records, such as a Lamport clock [17]. Version vectors contain more precise information than Lamport clocks about causality; Ivy uses that information to help fix conflicting updates after a partition. Version vectors help prevent a malicious participant from retroactively changing its log by pointing its log-head at a newly-constructed log; other participants' version vectors will still point to the old log's records. Finally, version vectors from one log could be used to help repair another log that has been damaged.

2.6 Snapshots

Each Ivy participant periodically constructs a private snapshot of the file system in order to avoid traversing the entire log. A snapshot contains the entire state of the file system. Participants store their snapshots in DHash to make them persistent. Each participant has its own logically private snapshot, but the fact that the different snapshots have largely identical contents means that DHash automatically shares their storage.

2.6.1 Snapshot Format

A snapshot consists of a *file map*, a set of i-nodes, and some data blocks. Each i-node is stored in its own DHash block. An i-node contains file attributes as well as a list of DHash keys of blocks holding the file's contents; in the case of a directory, the content blocks hold a list of name/i-number pairs. The file map records the DHash key of the i-node associated with each i-number. All of the blocks that make up a snapshot are content-hash blocks. Figure 2 illustrates the snapshot data structure.

2.6.2 Building Snapshots

In ordinary operation Ivy builds each new snapshot incrementally. It starts by fetching all log records (from all logs in the view) newer than the previous snapshot. It traverses these new records in temporal order. For each i-number that occurs in the new log records, Ivy maintains an i-node and a copy of the file contents. Ivy reads the initial copy of the i-node and file contents from the previous snapshot, and performs the operation indicated by each log record on this data.

After processing the new log records, Ivy writes the accumulated i-nodes and file contents to DHash. Then it computes a new file map by changing the entries corresponding to changed i-nodes and appending new entries. Ivy creates a *snapshot block* that contains the file map and the following meta-data: a pointer to the view upon which the snapshot is based, a pointer to the previous snapshot, and a version vector referring to the most recent record from each log that the snapshot incorporates. Ivy stores the snapshot block in DHash under its content-hash, and updates the participant's log-head to refer to the new snapshot.

A new user must either build a snapshot from scratch, starting from the earliest record in each log, or copy another (trusted) user's snapshot.

2.6.3 Using Snapshots

When handling an NFS request, Ivy first traverses log records newer than the snapshot; if it cannot accumulate enough information to fulfill the request, Ivy finds the missing information in the participant's latest snapshot. Ivy finds information in a snapshot based on i-number.

3 Application Semantics

This section describes the file system semantics that Ivy provides to applications, focusing primarily on the ways in which Ivy's semantics differ from those of an ordinary NFS server. Sections 3.1, 3.2, and 3.3 describe Ivy's semantics when the network provides full connectivity. Sections 3.4 and 3.5 describe what happens when the network partitions and then merges.

3.1 Cache Consistency

In general, an update operation that one Ivy participant has completed is immediately visible to operations that other participants subsequently start. The exceptions are that Ivy can't enforce this notion of consistency during network partitions (see Section 3.4), and that Ivy provides close-to-open consistency for file data (see below). Most Ivy updates are immediately visible because 1) an Ivy server performing an update waits until DHash has acknowledged receipt of the new log records and the new log-head before replying to an NFS request, and 2) Ivy asks DHash for the latest log-heads at the start of every NFS operation. Ivy caches log records, but this cache never needs to be invalidated because the records are immutable.

For file reads and writes, Ivy provides a modified form of close-to-open consistency [13]: if application A_1 writes data to a file, then closes the file, and after the close has completed another application A_2 opens the file and reads it, A_2 will see the data written by A_1. Ivy may also make written data visible before the close. Most NFS clients and servers provide this form of consistency.

Close-to-open consistency allows Ivy to avoid fetching every log-head for each NFS READ operation. Ivy caches file blocks along with the version vector at the time each block was cached. When the application opens a file and causes NFS to send an ACCESS request, Ivy fetches all the log-heads from DHash. If no other log-heads have changed since Ivy cached blocks for the file, Ivy will satisfy subsequent READ requests from cached blocks without re-fetching log-heads. While the NFS client's file data cache often satisfies READs before Ivy sees them, Ivy's cache helps when an application has written a file and then re-reads it; the NFS client can't decide whether to satisfy the reads from the cached writes since it doesn't know whether some other client has concurrently written the file, whereas Ivy can decide if that is the case by checking the other log-heads.

Ivy defers writing file data to DHash until NFS tells it that the application is closing the file. Before allowing the `close()` system call to complete, Ivy appends the written data to the log and then updates the log-head. Ivy writes the data log records to DHash in parallel to reduce latency. This arrangement allows Ivy to sign and insert a new log-head once per file close, rather than once per file write. We added a new CLOSE RPC to the NFS client to make this work. Ivy also flushes cached writes if it receives a synchronous WRITE or a COMMIT.

3.2 Concurrent Updates

Ordinary file systems have simple semantics with respect to concurrent updates: the results are as if the updates occurred one at a time in some order. These semantics are natural and relatively easy to implement in a single file server, but they are more difficult for a decentralized file system. As a result, Ivy's semantics differ slightly from those of an ordinary file server.

The simplest case is that of updates that don't affect the same data or meta-data. For example, two participants may have created new files with different names in the same directory, or might have written different bytes in the same file. In such cases Ivy ensures that both updates take effect.

If different participants simultaneously write the same bytes in the same file, the writes will likely have equal or concurrent version vectors. Recall that Ivy orders incomparable version vector by comparing the participants' public keys. When the concurrent writes have completed, all the participants will agree on their order; in this case Ivy provides the same semantics as an ordinary file system. It may be the case that the applications did not intend to generate conflicting writes; Ivy provides both tools to help applications avoid conflicts (Section 3.3) and tools to help them detect and resolve unavoidable conflicts (Section 3.4).

Serial semantics for operations that affect directory entries are harder to implement. We believe that applications rely on the file system to provide serial semantics on directory operations in order to implement locking. Ivy supports one type of locking through the use of exclusive creation of directory entries with the same name (Section 3.3). Applications that use exclusive directory creation for locking will work on Ivy.

In the following paragraphs, we discuss specific cases that Ivy differs from a centralized file system due to the lack of serialization of directory operations.

Ivy does not serialize combinations of creation and deletion of a directory entry. For example, suppose one participant calls `unlink("a")`, and a second participant calls `rename("a", "b")`. Only one of these operations can succeed. On one hand, Ivy provides the expected semantics in the sense that participants who subsequently look at the file system will agree on the order of the concurrent log records, and will thus agree on which operation succeeded. On the other hand, Ivy will return a success status to both of the two systems calls, even though only one takes effect, which would not happen in an ordinary file system.

There are cases in which an Ivy participant may read logs that are actively being updated and initially see only a subset of a set of concurrent updates. A short time later the remaining concurrent updates might appear, but be ordered before the first subset. If the updates affect the same meta-data, observers could see the file system in states that could not have occured in a serial execution. For example, suppose application A_1 executes `create("x")` and `link("x","y")`, and application A_2 on a different Ivy host concurrently executes `remove("x")`. A third application A_3 might first see just the log records from A_1, and thus see files `x` and `y`; if Ivy orders the concurrent `remove()` between the `create()` and `link()`, then A_3 might later observe that both `x` and `y` had disappeared. If the three applications compare notes they will realize that the system did not behave like a serial server.

ExclusiveLink(*dir-inum*, *file*, *file-inum*)
  append a `Prepare`(*dir-inum*, *file*) log record
  if *file* exists
    append a `Cancel`(*dir-inum*, *file*) record
    return EXISTS
  if another un-canceled `Prepare`(*dir-inum*, *file*) exists
    append a `Cancel`(*dir-inum*, *file*) record
    backoff()
    return ExclusiveLink(*dir-inum*, *file*, *file-inum*)
  append `Link`(*dir-inum*, *file*, *file-inum*) log record
  return OK

Figure 3: Ivy's exclusive directory entry creation algorithm.

3.3 Exclusive Create

Ordinary file system semantics require that most operations that create directory entries be exclusive. For example, trying to create a directory that already exists should fail, and creating a file that already exists should return a reference to the existing file. Ivy implements exclusive creation of directory entries because some applications use those semantics to implement locks. However, Ivy only guarantees exclusion when the network provides full connectivity.

Whenever Ivy is about to append a `Link` log record, it first ensures exclusion with a variant of two-phase commit shown in Figure 3. Ivy first appends a `Prepare` record announcing the intention to create the directory entry. This intention can be canceled by a `Cancel` record, an eventual `Link` record, or a timeout. Then, Ivy checks to see whether any other participant has appended a `Prepare` that mentions the same directory i-number and file name. If not, Ivy appends the `Link` record. If Ivy sees a different participant's `Prepare`, it appends a `Cancel` record, waits a random amount of time, and retries. If Ivy sees a different participant's `Link` record, it appends a `Cancel` record and indicates a failure.

3.4 Partitioned Updates

Ivy cannot provide the semantics outlined above if the network has partitioned. In the case of partition, Ivy's design maximizes availability at the expense of consistency, by letting updates proceed in all partitions. This approach is similar to that of Ficus [26].

Ivy is not directly aware of partitions, nor does it directly ensure that every partition has a complete copy of all the logs. Instead, Ivy depends on DHash to replicate data enough times, and in enough distinct locations, that each partition is likely to have a complete set of data. Whether this succeeds in practice depends on the sizes of the partitions, the degree of DHash replication, and the total number of DHash blocks involved in the file system. The particular case of a user intentionally disconnecting a laptop from the network could be handled by instructing the laptop's DHash server to keep replicas of all the log-heads and the user's current snapshot; there is not currently a way to ask DHash to do this.

After a partition heals, the fact that each log-head was updated from just one host prevents conflicts within individual logs; it is sufficient for the healed system to use the newest version of each log-head.

Participants in different partitions may have updated the file system in ways that conflict; this will result in concurrent version vectors. Ivy orders such version vectors following the scheme in Section 2.5, so the participants will agree on the file system contents after the partition heals.

The file system's meta-data will be internally correct after the partition heals. What this means is that if a piece of data was accessible before the partition, and neither it nor any directory leading to it was deleted in any partition, then the data will also be accessible after the partition.

However, if concurrent applications rely on file system techniques such as atomic directory creation for mutual exclusion, then applications in different partitions might update files in ways that cause the application data to be inconsistent. For example, e-mails might be appended to the same mailbox file in two partitions; after the partitions heal, this will appear as two concurrent writes to the same offset in the mailbox file. Ivy knows that the writes conflict, and automatically orders the log entries so that all participants see the same file contents after the partition heals. However, this masks the fact that some file updates are not visible, and that the user or application may have to take special steps to restore them. Ivy does not currently have an automatic mechanism for signaling such conflicts to the user; instead the user must run the `lc` tool described in the next section to discover conflicts. A better approach might be to borrow Coda's technique of making the file inaccessible until the user fixes the conflict.

3.5 Conflict Resolution

Ivy provides a tool, `lc`, that detects conflicting application updates to files; these may arise from concurrent writes to the same file by applications that are in different partitions or which do not perform appropriate locking. `lc` scans an Ivy file system's log for records with concurrent version vectors that affect the same file or directory entry. `lc` determines the point in the logs at which the partition must have occurred, and determines which participants were in which partition. `lc` then uses Ivy views to construct multiple historic views of the file system: one as of the time of partition, and one for each partition

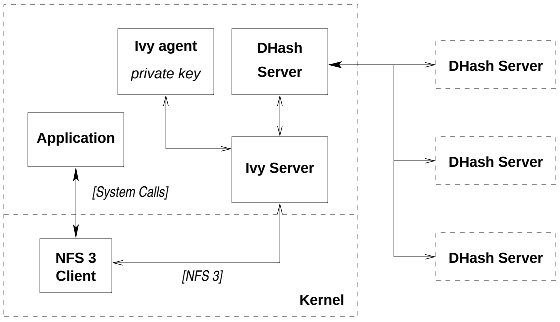

Figure 4: Ivy software structure.

just before the partition healed. For example,

```
% ./lc -v /ivy/BXz4+udjsQm4tX63UR9w71SNP0c
before: +WzW8s7fTEt6pehaB7isSfhkc68
partition1: l3qLDU5icVMRrbLvhxuJ1WkNvWs
partition2: JyCKgcsAjZ4uttbbtIX9or+qEXE
% cat /ivy/+WzW8s7fTEt6pehaB7isSfhkc68/file1
original content of file1
% cat /ivy/l3qLDU5icVMRrbLvhxuJ1WkNvWs/file1
original content of file1, changed
append on first partition
% cat /ivy/JyCKgcsAjZ4uttbbtIX9or+qEXE/file1
original content of file1
append on second partition
```

In simple cases, a user could simply examine the versions of the file and merge them by hand in a text editor. Application-specific resolvers such as those used by Coda [14, 16] could be used for more complex cases.

4 Security and Integrity

Since Ivy is intended to support distributed users with arms-length trust relationships, it must be able to recover from malicious participants. The situation we envision is that a participant's bad behavior is discovered after the fact. Malicious behavior is assumed to consist of the participant using ordinary file system operations to modify or delete data. One form of malice might be that an outsider breaks into a legitimate user's computer and modifies files stored in Ivy.

To cope with a good user turning bad, the other participants can either form a new view that excludes the bad participant's log, or form a view that only includes the log records before a certain point in time. In either case the resulting file system may be missing important meta-data. Upon user request, Ivy's `ivycheck` tool will detect and fix certain meta-data inconsistencies. `ivycheck` inspects an existing file system, finds missing `Link` and `Inode` meta-data, and creates plausible replacements in a new *fix log*. `ivycheck` can optionally look in the excluded log in order to find hints about what the missing meta-data should look like.

5 Implementation

Ivy is written in C++ and runs on FreeBSD. It uses the SFS tool-kit [22] for event-driven programming and NFS loop-back server support.

Ivy is implemented as several cooperating parts, illustrated in Figure 4. Each participating host runs an Ivy server which exposes Ivy file systems as locally-mounted NFS v3 file systems. A file system name encodes the DHash key of the file system's view block, for example, `/ivy/9RYBbWyeDVEQnxeL95LG5jJjwa4`. The Ivy server does not hold private keys; instead, each participant runs an agent to hold its private key, and the Ivy server asks the participant's local agent program to sign log heads. The Ivy server acts as a client of a local DHash server, which consults other DHash servers scattered around the network. The Ivy server also keeps a LRU cache of content-hash blocks (e.g. log records and snapshot blocks) and log-heads that it recently modified.

6 Evaluation

This section evaluates Ivy's performance 1) in a purely local configuration, 2) over a WAN, 3) as a function of the number of participants, 4) as a function of the number of DHash nodes, 5) as a function of the number of concurrent writers, and 6) as a function of the snapshot interval. The main goal of the evaluation is to understand the costs of Ivy's design in terms of network latency and cryptographic operations.

Ivy is configured to construct a snapshot every 20 new log records, or when 60 seconds have elapsed since the construction of the last snapshot. Unless otherwise stated, Ivy's block cache size is 512 blocks. DHash nodes are PlanetLab [1] nodes, running Linux 2.4.18 on 1.2 GHz Pentium III CPUs, and RON [2] nodes, running FreeBSD 4.5 on 733 MHz Pentium III CPUs. DHash was configured with replication turned off, since the replication implementation is not complete; replication would probably decrease performance significantly. Unless otherwise stated, this section reports results averaged over five runs.

The workload used to evaluate Ivy is the Modified Andrew Benchmark (MAB), which consists of five phases: (1) create a directory hierarchy, (2) copy files into these directories, (3) walk the directory hierarchy while reading attributes of each file, (4) read the files, and (5) compile the files into a program. Unless otherwise stated, the MAB and the Ivy server run on a 1.2 GHz AMD Athlon computer running FreeBSD 4.5 at MIT.

6.1 Single User MAB

Table 3 shows Ivy's performance on the phases of the MAB for a file system with just one log. All the soft-

Phase	Ivy (s)	NFS (s)
Mkdir	0.6	0.5
Create/Write	6.6	0.8
Stat	0.6	0.2
Read	1.0	0.8
Compile	10.0	5.3
Total	18.8	7.6

Table 3: Real-time in seconds to run the MAB with a single Ivy log and all software running on a single machine. The NFS column shows MAB run-time for NFS over a LAN.

Phase	Ivy (s)	NFS (s)
Mkdir	11.2	4.8
Create/Write	89.2	42.0
Stat	65.6	47.8
Read	65.8	55.6
Compile	144.2	130.2
Total	376.0	280.4

Table 4: MAB run-time with four DHash servers on a WAN. The file system contains four logs.

ware (the MAB, Ivy, and a single DHash server) ran on the same computer. To put the Ivy performance in perspective, Table 3 also shows MAB performance over NFS; the client and NFS server are connected by a 100 Mbit LAN. Note that this comparison is unfair to NFS, since NFS involved network communication while the Ivy benchmark did not.

The following analysis explains Ivy's 18.8 seconds of run-time. The MAB produces 386 NFS RPCs that modify the Ivy log. 118 of these are either MKDIR or CREATE, which require two log-head writes to achieve atomicity. 119 of the 386 RPCs are COMMITs or CLOSEs that require Ivy to flush written data to the log. Another 133 RPCs are synchronous WRITEs generated by the linker. Overall, the 386 RPCs caused Ivy to update the log-head 508 times. Computing a public-key signature uses about 14.2 milliseconds (ms) of CPU time, for a total of 7.2 seconds of CPU time.

The remaining time is spent in the Ivy server (4.9 seconds), the DHash server (2.9 seconds), and in the processes that MAB invokes (2.6 seconds). Profiling indicates that the most expensive operations in the Ivy and DHash servers are SHA-1 hashes and memory copies.

The MAB creates a total of 1.6 MBytes of file data. Ivy, in response, inserts a total of 8.8 MBytes of log and snapshot data into DHash.

6.2 Performance on a WAN

Table 4 shows the time for a single MAB instance with four DHash servers on a WAN. One DHash server runs on the same computer that is running the MAB. The average network round-trip times to the other three DHash servers are 9, 16, and 82 ms. The file system contains four logs. The benchmark only writes one of the logs, though the other three log-heads are consulted to make sure operations see the most up-to-date data. The four log-heads are stored on three DHash servers. The log-head that is being written to is stored on the DHash server with a round-trip time of 9 ms from the local machine. One log-head is stored on the server with a round-trip time of 82 ms from the local machine. The DHash servers' node IDs are chosen so that each is responsible for roughly the same number of blocks.

A typical NFS request requires Ivy to fetch the three other log-heads from DHash; this involves just one DHash network RPC per log-head. Ivy issues the three RPCs in parallel, so the time for each log-head check is governed by the largest round-trip time of 82 ms. The MAB causes Ivy to retrieve log-heads 3,346 times, for a total of 274 seconds. This latency dominates Ivy's WAN performance.

The remaining 102 seconds of MAB run-time are used in four ways. Running the MAB on a LAN takes 22 seconds, mostly in the form of CPU time. Ivy writes its log-head to DHash 508 times; each write takes 9 ms of network latency, for a total of 5 seconds. Ivy inserts 1,003 log records, some of them concurrently. The average insertion takes 54 ms (27 ms for the Chord [37] lookup, then another 27 ms for the DHash node to acknowledge receipt). This accounts for roughly 54 seconds. Finally, the local computer sends and receives 7.0 MBytes of data during the MAB run. This accounts for the remaining run time. During the experiment Ivy also inserts 358 DHash blocks while updating its snapshot; because Ivy doesn't wait for these inserts, they contribute little to the total run time.

Table 4 also shows MAB performance over wide-area NFS. The round-trip time between the NFS client and server is 79 ms, which is roughly the time it takes Ivy to fetch all the log-heads. We use NFS over UDP because it is faster for this benchmark than FreeBSD's NFS over TCP implementation. Ivy is slower than NFS because Ivy operations often require more network round-trips; for example, some NFS requests require Ivy to both fetch and update log-heads, requiring two round-trips.

6.3 Many Logs, One Writer

Figure 5 shows how Ivy's performance changes as the number of logs increases. Other than the number of logs, this experiment is identical to the one in the previous sec-

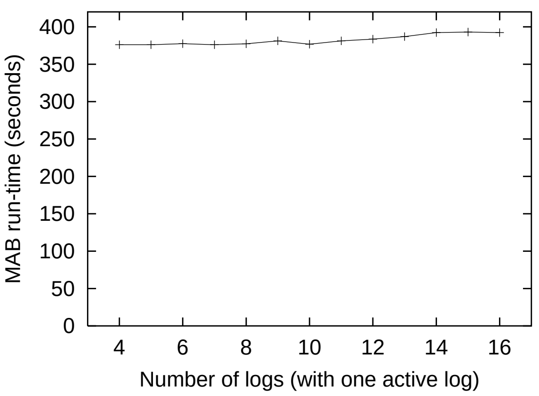

Figure 5: MAB run-time as a function of the number of logs. Only one participant is active.

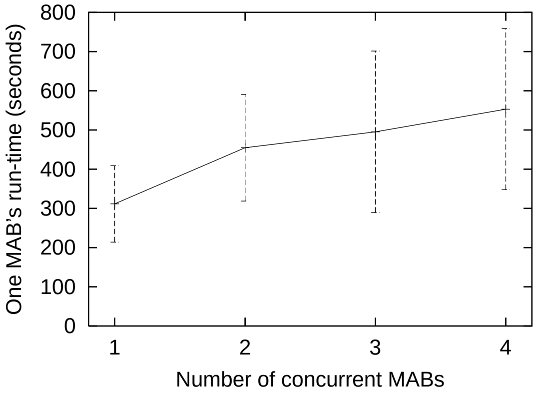

Figure 7: Average run-time of MAB when several MABs are running concurrently on different hosts on the Internet. The error bars indicate standard deviation over all the MAB runs.

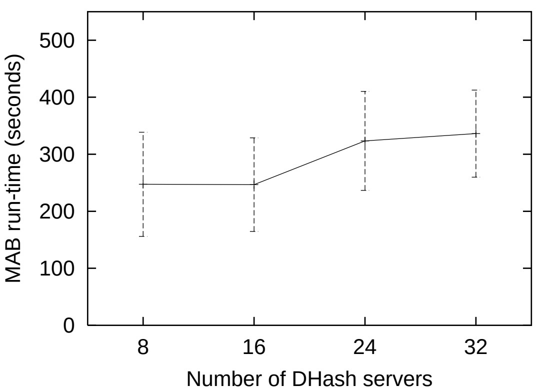

Figure 6: Average MAB run-time as the number of DHash servers increases. The error bars indicate standard deviation over different choices of PlanetLab hosts and different mappings of blocks to DHash servers.

tion. The number of logs ranges from 4 to 16, but only one participant executes the MAB — the other logs never change. Figure 5 reports results averaged over three runs.

The number of logs has relatively little impact on run-time because Ivy fetches the log-heads in parallel. There is a slight increase caused by the fact that the version vector in each log record has one 44-byte entry per participant.

6.4 Many DHash Servers

Figure 6 shows the averages and standard deviations of Ivy's MAB performance as the number of DHash servers increases from 8 to 32. For each number of servers we perform ten experimental runs. For each run, all but one of the DHash servers are placed on randomly chosen PlanetLab hosts (from a pool of 32 hosts); new log-head public keys are also used to ensure the log-heads are placed on random DHash servers. One DHash server, the Ivy server, and the MAB always execute on a host at MIT. The round-trip times from the host at MIT to the PlanetLab hosts average 32 ms, with a minimum of 1 ms, a maximum of 78 ms, and a standard deviation of 27 ms. There are four logs in total; only one of them changes.

The run-time in Figure 6 grows because more Chord messages are required to find each log record block in DHash. An average of 2.3, 2.9, 3.3, and 3.8 RPCs are required for 8, 16, 24, and 32 DHash servers, respectively. These numbers include the final DHash RPC as well as Chord lookup RPCs.

The high standard deviation in Figure 6 is due to the fact that the run-time is dominated by the round-trip times to the four particular DHash servers that store the log-heads. This means that adding more DHash servers doesn't reduce the variation.

6.5 Many Writers

Figure 7 shows the effect of multiple active writers. We perform three experiments for each number N of participants; each experiment involves one MAB running concurrently on each of N different Ivy hosts on the Internet, a file system with four logs, new log-head public keys, and 32 DHash servers. Each MAB run uses its own directory in the Ivy file system. Each data point shows the average and standard deviation of MAB run-time over the $3N$ MAB executions.

The run-time increases with the number of active participants because each has to fetch the others' newly appended log records from DHash. The run-time increases relatively slowly because Ivy fetches records from the different logs in parallel. The deviation in run-times is

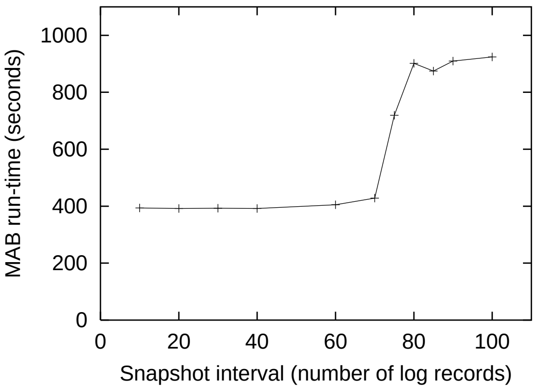

Figure 8: MAB run-time a function of the interval between snapshots. For these experiments, the size of Ivy's block cache is 80 blocks.

Phase	Ivy (s)	NFS (s)	ssh (s)
Commit	420.8	224.6	3.4
Update	284.2	135.2	2.3

Table 5: Run-times for the CVS experiment phases. DHash is running on 32 nodes on a wide-area network.

due to each participant having different network round-trip latencies to the DHash servers.

6.6 Snapshot Interval

Figure 8 shows the effect on MAB run-time of the interval between snapshots. The experiments involve one MAB instance, four logs, and 32 DHash servers. The x-axis represents the number of new log records inserted before Ivy builds each a new snapshot. For these experiments, the size of Ivy's block cache is 80 blocks. The reason the run-time increases when the interval is greater than 80 is that not all the records needed to build each snapshot can fit in the cache.

6.7 Wide-area CVS on Ivy

To evaluate Ivy's performance as a source-code or document repository, we show the run-time of some operations on a CVS [4] repository stored in Ivy. The Ivy file system has four logs stored on 32 wide-area DHash servers. The round-trip times from the Ivy host to the DHash servers storing the log-heads are 17, 36, 70, and 77 ms. The CVS repository contains 251 files and 3.3 MBytes. Before the experiment starts, two Ivy participants, X and Y, check out a copy of the repository to their local disks, and both create an Ivy snapshot of the file system. Each participant then reboots its host to ensure that no data is cached. The experiment consists of two phases. First, X commits changes to 38 files, a total of 4333 lines. Second, Y updates its local copy to reflect X's changes. Table 5 shows the run-times for the two phases. For comparison, Table 5 shows the time to perform the same CVS operations over NFS and `ssh`; in both cases the client to server round-trip latency is 77 ms.

Ivy's performance with CVS is disappointing. During a commit or update, CVS looks at every file in the repository; for each file access, Ivy checks whether some other participant has recently changed the file. CVS has locked the repository, so no such changes are possible; but Ivy doesn't know that. During a CVS commit, Ivy waits for the DHash insert of new log records and an updated log-head for each file modified; again, since CVS has locked the repository, Ivy could have written all the log records in parallel and just a single updated log-head for the whole CVS commit. A transactional interface between application and file system would help performance in this situation.

7 Related Work

Ivy was motivated by recent work on peer-to-peer storage, particularly FreeNet [8], PAST [32], and CFS [9]. The data authentication mechanisms in these systems limit them to read-only or single-publisher data, in the sense that only the original publisher of each piece of data can modify it. CFS builds a file-system on top of peer-to-peer storage, using ideas from SFSRO [11]; however, each file system is read-only. Ivy's primary contribution relative to these systems is that it uses peer-to-peer storage to build a read/write file system that multiple users can share.

7.1 Log-structured File System

Sprite LFS [31] represents a file system as a log of operations, along with a snapshot of i-number to i-node location mappings. LFS uses a single log managed by a single server in order to to speed up small write performance. Ivy uses multiple logs to let multiple participants update the file system without a central file server or lock server; Ivy does not gain any performance by use of logs.

7.2 Distributed Storage Systems

Zebra [12] maintains a per-client log of file contents, striped across multiple network nodes. Zebra serializes meta-data operations through a single meta-data server. Ivy borrows the idea of per-client logs, but extends them to meta-data as well as file contents. This allows Ivy to avoid Zebra's single meta-data server, and thus potentially achieve higher availability.

xFS [3], the Serverless Network File System, distributes both data and meta-data across participating hosts. For every piece of meta-data (e.g. an i-node) there is a host that is responsible for serializing updates to that meta-data to maintain consistency. Ivy avoids any meta-data centralization, and is therefore more suitable for wide-area use in which participants cannot be trusted to run reliable servers. However, Ivy has lower performance than xFS and adheres less strictly to serial semantics.

Frangipani [40] is a distributed file system with two layers: a distributed storage service that acts as a virtual disk and a set of symmetric file servers. Frangipani maintains fairly conventional on-disk file system structures, with small, per-server meta-data logs to improve performance and recoverability. Frangipani servers use locks to serialize updates to meta-data. This approach requires reliable and trustworthy servers.

Harp [18] uses a primary copy scheme to maintain identical replicas of the entire file system. Clients send all NFS requests to the current primary server, which serializes them. A Harp system consists of a small cluster of well managed servers, probably physically colocated. Ivy does without any central cluster of dedicated servers—at the expense of strict serial consistency.

7.3 Reclaiming Storage

The Elephant file system [34] allows all file system operations to be undone for a period defined by the user, after which the change becomes permanent. While Ivy does not currently reclaim log storage, perhaps it could adopt Elephant's version retention policies; the main obstacle is that discarding log entries would hurt Ivy's ability to recover from malicious participants. Experience with Venti [28] suggests that retaining old versions of files indefinitely may not be too expensive.

7.4 Consistency and Conflict Resolution

Coda [14, 16] allows a disconnected client to modify its own local copy of a file system, which is merged into the main replica when the client re-connects. A Coda client keeps a replay log that records modifications to the client's local copies while the client is in disconnected mode. When the client reconnects with the server, Coda propagates client's changes to the server by replaying the log on the server. Coda detects changes that conflict with changes made by other users, and presents the details of the changes to application-specific conflict resolvers. Ivy's behavior after a partition heals is similar to Coda's conflict resolution: Ivy automatically merges non-conflicting updates in the logs and lets application-specific tools handle conflicts.

Ficus [26] is a distributed file system in which any replica can be updated. Ficus automatically merges non-conflicting updates from different replicas, and uses version vectors to detect conflicting updates and to signal them to the user. Ivy also faces the problem of conflicting updates performed in different network partitions, and uses similar techniques to handle them. However, Ivy's main focus is connected operation; in this mode it provides close-to-open consistency, which Ficus does not, and (in cooperation with DHash) does a better job of automatically distributing storage over a wide-area system.

Bayou [39] represents changes to a database as a log of updates. Each update includes an application-specific *merge procedure* to resolve conflicts. Each node maintains a local log of all the updates it knows about, both its own and those by other nodes. Nodes operate primarily in a disconnected mode, and merge logs pairwise when they talk to each other. The log and the merge procedures allow a Bayou node to re-build its database after adding updates made in the past by other nodes. As updates reach a special primary node, the primary node decides the final and permanent order of log entries. Ivy differs from Bayou in a number of ways. Ivy's per-client logs allow nodes to trust each other less than they have to in Bayou. Ivy uses a distributed algorithm to order the logs, which avoids Bayou's potentially unreliable primary node. Ivy implements a single coherent data structure (the file system), rather than a database of independent entries; Ivy must ensure that updates leave the file system consistent, while Bayou shifts much of this burden to application-supplied merge procedures. Ivy's design focuses on providing serial semantics to connected clients, while Bayou focuses on managing conflicts caused by updates from disconnected clients.

7.5 Storing Data on Untrusted Servers

BFS [7], OceanStore [15], and Farsite [5] all store data on untrusted servers using Castro and Liskov's practical Byzantine agreement algorithm [7]. Multiple clients are allowed to modify a given data item; they do this by sending update operations to a small group of servers holding replicas of the data. These servers agree on which operations to apply, and in what order, using Byzantine agreement. The reason Byzantine agreement is needed is that clients cannot directly validate the data they fetch from the servers, since the data may be the result of incremental operations that no one client is aware of. In contrast, Ivy exposes the whole operation history to every client. Each Ivy client signs the head of a Merkle hash-tree [25] of its log. This allows other clients to verify that the log is correct when they retrieve it from DHash; thus Ivy clients do not need to trust the DHash servers to maintain the correctness or order of the logs. Ivy is vulnerable to DHash returning stale copies of signed log-heads; Ivy could detect stale data using techniques introduced by SUNDR [24]. Ivy's use of logs makes it slow, although this inefficiency is partially offset by its snapshot mechanism.

TDB [20], S4 [38], and PFS [36] use logging and (for TDB and PFS) collision-resistant hashes to allow modifications by malicious users or corrupted storage devices to be detected and (with S4) undone; Ivy uses similar techniques in a distributed file system context.

Spreitzer et al. [35] suggest ways to use cryptographically signed log entries to prevent servers from tampering with client updates or producing inconsistent log orderings; this is in the context of Bayou-like systems. Ivy's logs are simpler than Bayou's, since only one client writes any given log. This allows Ivy to protect log integrity, despite untrusted DHash servers, by relatively simple per-client use of cryptographic hashes and public key signatures.

8 Conclusion

This paper presents Ivy, a multi-user read/write peer-to-peer file system. Ivy is suitable for small groups of cooperating participants who do not have (or do not want) a single central server. Ivy can operate in a relatively open peer-to-peer environment because it does not require participants to trust each other.

An Ivy file system consists solely of a set of logs, one log per participant. This arrangement avoids the need for locking to maintain integrity of Ivy meta-data. Participants periodically take snapshots of the file system to minimize time spent reading the logs. Use of per-participant logs allows Ivy users to choose which other participants to trust.

Due to its decentralized design, Ivy provides slightly non-traditional file system semantics; concurrent updates can generate conflicting log records. Ivy provides several tools to automate conflict resolution. More work is under way to improve them.

Experimental results show that the Ivy prototype is two to three times slower than NFS. Ivy is available from `http://www.pdos.lcs.mit.edu/ivy/`.

Acknowledgments

We thank M. Satyanarayanan and Carnegie-Mellon University for making the Modified Andrew Benchmark available. We are grateful to the PlanetLab and RON testbeds for letting us run wide-area experiments. Sameer Ajmani, Trevor Blackwell, Miguel Castro, Josh Cates, Russ Cox, Peter Druschel, Frans Kaashoek, Alex Lewin, David Mazières, and Rodrigo Rodrigues gave us helpful feedback about the design and description of Ivy. This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933, and by grants from NTT Corporation under the NTT-MIT collaboration.

References

[1] Planet Lab. `http://www.planet-lab.org/`.

[2] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of the ACM Symposium on Operating System Principles*, pages 109–126, December 1995.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proc. Winter 1990 USENIX Technical Conference*, 1990.

[5] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *ACM SIGMETRICS Conference*, June 2000.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[10] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[11] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[12] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 213–225, 1991.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, pages 190–201, November 2000.

[16] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proc. of the Second International Conference on Parallel and Distributed Information Systems*, pages 202–213, January 1993.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 226–38, 1991.

[19] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[20] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[21] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[22] D. Mazières. A toolkit for user-level file systems. In *Proc. of the Usenix Technical Conference*, pages 261–274, June 2001.

[23] D. Mazières, M. Kaminsky, M. Frans Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of the ACM Symposium on Operating System Principles*, December 1999.

[24] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[25] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, 1987.

[26] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.

[27] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, volume 9(3), pages 240–247, 1983.

[28] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. of the Conference on File and Storage Technologies (FAST)*, January 2002.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, August 2001.

[30] D. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[31] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[32] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[33] R. Sandberg, D. Goldberg, D. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Usenix Summer Conference*, pages 119–130, June 1985.

[34] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 110–123, 1999.

[35] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of the ACM/IEEE MobiCom Conference*, September 1997.

[36] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proc. of the USENIX Technical Conference*, pages 79–90, 2001.

[37] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[38] J. Strunk, G. Goodson, M. Scheinholtz, and C. Soules. Self-securing storage: Protecting data in compromised systems. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–179, October 2000.

[39] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 172–183, December 1995.

[40] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 224–237, 1997.

[41] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.