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. It is suitable for small cooperative groups spread over large geographic areas. Ivy allows such groups to avoid the reliability and trust problems inherent in use of a central file server.

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 traditional 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 about a factor of XXX slower than NFS.

1 Introduction

This paper describes Ivy, a distributed read/write network file system. Ivy is distributed in the sense that it has no centralized components or data structures; it stores and replicates all data and meta-data in an underlying peer-to-peer block storage system.

One target use of Ivy is to support distributed projects with loosely affiliated participants. Ivy presents a single file system image, much like an NFS [33] server. In contrast to NFS, Ivy does not require a dedicated server, but replicates all data over many machines. Consequently, an Ivy file system is highly available despite the failure of any one computer or (assuming enough geographically separated participants) any one site.

This mode of use 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 undo some or all modifications by a participant revealed to be untrustworthy. Finally, distributed applications using the file system may not have a reliable locking mechanism available; this suggests that the file system should provide tools to detect and help repair application-level consistency problems.

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. To avoid scanning through the whole log, each participant maintains a private snapshot which allows Ivy to find data quickly; Ivy typically only scans the last few records of each log. Ivy's use of per-participant logs allows it to avoid locking to protect meta-data, since Ivy has no shared mutable data structures.

Ivy stores its log records in the DHash [9] distributed hash system; interaction among Ivy participants proceeds indirectly via DHash storage. DHash replicates each record on multiple Internet hosts to increase availability. Ivy cryptographically verifies all data it retrieves from DHash.

An Ivy participant can choose 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 traditional file system semantics when the underlying network is fully connected. For example, Ivy provides close-to-open consistency. If the network is partitioned, 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 partition merges, 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 to each participant's applications. Performance is good enough for applications such as shared CVS [4] software repositories. 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 security 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. Section 5 and 6 describes Ivy's implementation and performance. Section 7 suggests future work. Section 8 discusses related work. Section 9 concludes.

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 reading entire logs on each read operation, 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 cryptographic hash of the block's value, using SHA-1 [10]; this allows anyone fetching the block to verify the value by ensuring its SHA-1 hash matches the

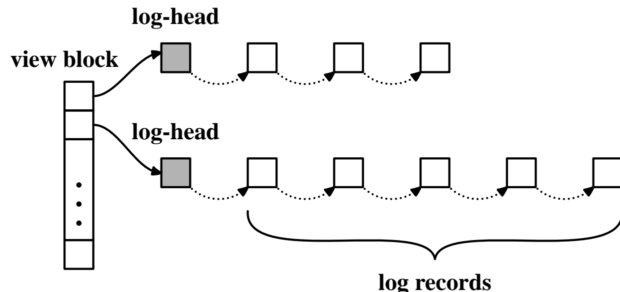

Figure 1: Example Ivy view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

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 never communicate directly with each other. 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. Ivy does not cache public-key blocks, since they may be changed.

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]. Ivy operates best in a fully connected network, though it has support for conflict detection after operation 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 [39], 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 block. Table 1 describes fields common to all log records. The `prev` field contains a pointer to the previous record, in the form of

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.

the previous record's DHash key. A participant stores the content hash of its most recent log record in its log-head block. The log-head is a public-key block with a fixed address, which makes it easy for other participants to find it.

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 reconstructs that information when traversing the logs. Another way of looking at this is that Ivy logs do not include explicit data structures corresponding to UNIX i-nodes [30, 25].

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 using the same i-number. Ivy uses the 160-bit i-number as the NFS file handle.

The only NFS file attributes that Ivy fully supports are file size, file modification time, file attribute modification time, and link count. 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.

2.3 Using the Log

For the moment, consider an Ivy file system with only one log. Ivy handles NFS read 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 performs selected operations.

File system creation. Ivy builds a new file system by creating a new log with an `End` record and an `Inode` record with a random i-number for the root directory. Then it creates a log-head that points to the `Inode` record. The user then instructs the in-kernel NFS client code to mount the Ivy file system as a local NFS file system, giving the NFS client the root i-number as the 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 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 (in the file handle), the written data, and the file offset; Ivy will append a `Write` log record containing the same information.

File name lookup. System calls that refer to file names, such as `open()`, 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 to handle the NFS request. A file's length is determined by either the write to the highest offset since the last truncation, or by

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.

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 implementations depend 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. Ivy retrieves the owner and mode from the `Inode` record or from the most recent relevant `SetAttrs` record.

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 others 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 contains a tuple *U:V* for each other log in the view, summarizing the participant's most recent knowledge of that log. *U* is the DHash key of the log's log-head, and *V* is the DHash key of the most recent record in the log. 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 sequence numbers in the records' version vectors. For example, *(A:5 B:7)* $<$ *(A:6 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.

Suppose an Ivy file system has two logs, A and B. The sequence number in A's most recent record is 7, and that in B's most recent record is 3. If the participant that

owns log A appends a record, the new record will have sequence number 8, and version vector *(A:7 B:3)*. When either participant reads the two logs, it will consider A's new record to be more recent than B's record number 3.

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

Ivy could have used a simpler method of ordering log records, such as a Lamport clock. The additional information in the version vectors has two advantages. First, 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. Second, version vectors from one log could be used to help repair another log that has been damaged.

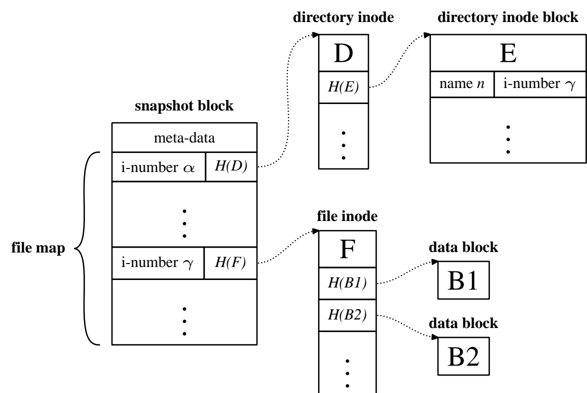

Figure 2: Snapshot data structure. $H(A)$ is the DHash content-hash of A.

2.6 Snapshots

In order to avoid the expense of traversing the entire log, each Ivy participant periodically constructs a private snapshot of the file system. A snapshot contains the entire current 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

An Ivy participant can build a snapshot incrementally from an older snapshot or completely from scratch. In ordinary operation Ivy builds snapshots incrementally; it only builds one from scratch when first joining an Ivy file system.

Ivy starts to build an incremental snapshot 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. File i-node data are kept in memory, while file contents are kept on-disk. It 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 build a snapshot by copying another (trusted) user's snapshot. A read-only user can use any trusted participant's snapshot.

2.6.3 Using Snapshots

When handling an NFS request, Ivy first traverses log records newer than the latest 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.

Ivy keeps log records even after it has built a snapshot: other participants may need those records to build their own snapshots. Another reason is that a snapshot may need to be re-built entirely from the logs if a user is revealed to be malicious. Finally, the conflict resolution

algorithms described in Section 3 require the log records.

3 Application Semantics

This section describes the file system semantics that Ivy provides to applications. Sections 3.1, 3.2, and 3.3 describe application semantics when the network provides full connectivity. Section 3.4 describes what happens when the network partitions and then merges. Section 3.5 deals with application level conflict resolution.

3.1 Write-Read Consistency

When the network provides full connectivity, Ivy provides nearly the same semantics as an ordinary single-server NFS file system. A file read reflects the most recently completed write. Ivy differs from single-server semantics only in its treatment of concurrent directory modifications. Sections 3.2 and 3.3 details the issues in concurrent directory operations and how Ivy resolves them.

To improve performance, each Ivy participant maintains a cache of log records. Because log records are immutable once written, no cache consistency protocol is needed. When a participant appends to the log, it waits until the new log records and the new log-head are stored in DHash before replying to the NFS request. A participant does not cache other participants' log-head blocks; instead, it re-reads the other log-heads at the start of every NFS operation. These rules mean that by the time one participant sends an NFS reply for an updating operation, the update will be visible to all other participants.

The NFS client in the kernel typically caches file data and meta-data. Most NFS clients provide a form of close-to-open consistency [13], typically by issuing an NFS ACCESS request to check the last-modified time of a file when an application opens it; this allows the NFS client to decide if the cached data is stale. NFS clients also ensure that they have sent all modified data to the server when the application closes a file. Together, Ivy and the NFS client provide the following consistency guarantee: if application A_1 writes data to a file, closes the file, and waits for the close to complete, and after that another application A_2 opens the file and reads it, A_2 will see the data written by A_1.

There are cases in which an Ivy participant may read logs that are actively being updated and see only a subset of a set of concurrent updates. A short time later the participant may see all of the updates, but it may turn out that the updates it saw later are earlier in the version vector order. This can cause a brief inconsistency.

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 Ivy. 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 participant's public keys. Thus Ivy serializes the writes, and all participants agree on the 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. Ivy serializes the creation of directory entries with the same name, using techniques described in the next section. However, 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. One the 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 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.

3.3 Exclusive Create

Ordinary file system semantics require that most operations that create directory entries be exclusive. For example, trying to create a subdirectory 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

```
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.

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 returns a failure indication.

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 [14].

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; Ivy does not yet have a way to tell 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. If that happens, the version vectors will reflect the partition. For example, if the system partitions after version vector *(A:4 B:6)*, A's log may contain version vectors such as *(A:9 B:6)*, while B's log might contain *(A:4 B:8)*. Such records are not ordered with respect to each other; Ivy will break the tie by comparing the numeric values of the participants' public keys. This causes the participants to 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 were lost, 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 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 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. In more complex cases, application-specific resolvers such as those used by Coda [15, 17] could be used to merge the changes from each partition.

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.

Because Ivy logs all operations, users have a choice of ways to cope with a malicious participant. Ivy allows the good users to create a new view block that excludes the bad participant's log, or excludes that participant's log records after a certain point; the latter works because log records are immutable and because good users' version vectors fix the history of the bad user's log. The good users could inspect the bad user's log, accepting only a subset of the records; Ivy does not currently support this.

If the good users exclude some or all of the bad user's log, the resulting file system's meta-data may not be consistent. For example, `Write` records that refer to a file created in the excluded log may be uninterpretable. Similarly, ignoring `Link` records in the excluded log can cause files and directories to become inaccessible.

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*. Ivy can do this by inspecting valid logs only or by using the untrusted log(s) to provide hints as to what these `Link` and `Inode` records should look like. For instance, without the original `Link` log record it is hard to guess a file's name or containing directory; in such cases `ivycheck` puts the file in a `lost+found` directory.

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. Every participating host runs an Ivy server which acts as a loop-back NFS v3 server; this arrangement effectively forwards application system calls via NFS to the local Ivy server. An Ivy file system appears under a name that encodes the DHash key of the file system's view block, for example, `/ivy/9RYBbWyeDVEQnxeL95LG5jJjwa4`. The Ivy server does not itself hold local participants' 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 the DHash storage system; this means that the Ivy server sends DHash put/get RPCs to DHash servers scattered around the network.

Ivy provides tools for users to create new public/private key pairs, to create new file systems, and to create and modify views.

5.1 Close-to-Open Consistency Cache

Ivy avoids having to fetch every log-head for each NFS READ operation in the following way. Ivy caches file contents, and remembers for each cached file block the version vector as of the time the block was cached. When an application opens a file (and the NFS client code sends an ACCESS request), Ivy reads all the log-heads to check if the cached contents are still up-to-date. If they are, Ivy will satisfy subsequent READ requests without fetching log-heads. This provides the same cache consistency as ordinary NFS.

When Ivy receives a WRITE request from an application, it does not immediately append a new log record or update the log-head. Instead, Ivy waits until the NFS client code notifies it that the application is closing the file; at that point (before allowing the `close()` system call to complete) Ivy will append the `Write` records and update the log-head. We modified the FreeBSD NFS client implementation to send a new CLOSE RPC in order to make this work. Ivy also flushes the cached writes and updates the log-head if it receives a synchronous WRITE or a COMMIT. Again, this provides the same cache consistency as ordinary NFS.

This caching means that Ivy does not need to retrieve other participants' log-heads on each READ or WRITE, but only when files are opened and closed. When writing files, Ivy only needs to update the log-head (and wait for the update to finish) when the file is closed, rather than during each WRITE. In addition, Ivy sends batches of new `Write` records to DHash in parallel, which reduces the latency of file writes. These optimizations reduce the effect of the latency involved in waiting for DHash to read and write log-heads, and reduce the CPU time spent signing log heads and checking signatures.

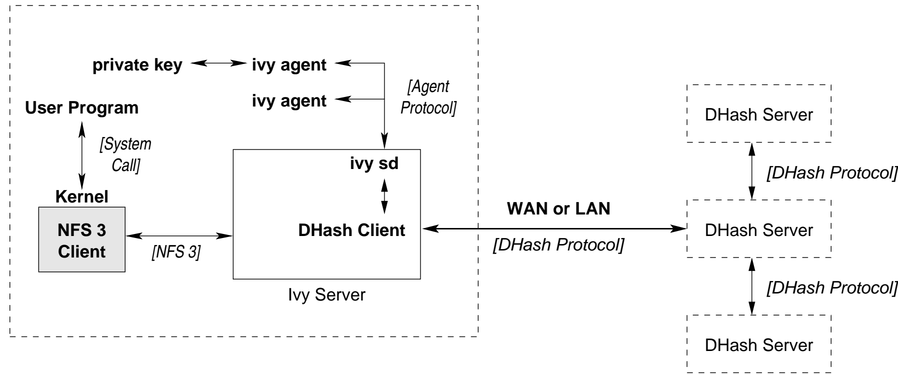

Figure 4: Implementation overview of the Ivy file system.

6 Evaluation

This section evaluates Ivy's performance: the performance of the Modified Andrew Benchmark (MAB) on a single-participant Ivy system with one DHash node, the performance of an Ivy system over a WAN, and the performance as a function of the number of participants, DHash nodes, and concurrent writers.

For all of the experiments, the Ivy servers run on PCs running FreeBSD 4.5 on 1.2 GHz AMD Athlon CPUs. Ivy is configured to construct a snapshot every 10 new log records [1]. 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. Unless otherwise stated, this section reports results averaged over five runs.

The MAB 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 a program.

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 software (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 were 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 the 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 follow WRITE RPCs. Each CLOSE RPC flushes a cached file to the log. Another 133 RPCs are synchronous WRITE RPCs. Overall, Ivy updated the log-head 508 times. Table 4 summarizes micro-benchmark times for various cryptographic operations; 508 public-key signatures takes 7.2 seconds of CPU time. Profiling shows that 7.6 seconds are spent in the Ivy agent, which computes the signatures.

The remaining time can be roughly divided into time spent in the Ivy server (4.9 seconds), time spent in the DHash server (2.9 seconds), and time spent in the processes that MAB invokes (2.6 seconds). Profiling indi-

[1] On machines with sufficient memory and a fast disk, Ivy's performance does not change when snapshot construction frequency varies from 1 log record to 20 log records.

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: Seconds required to run MAB on Ivy for a one-participant log with all software running on a single machine. The NFS column shows the performance of MAB on a client connected by a LAN to an NFS server.

Operation	Cost (μs)
Signing a 1024 byte log-head block	14,192
Verify signature on log-head block	39
Compute SHA-1 hash on 1024 bytes	29

Table 4: Costs of the cryptographic operations that Ivy uses in microseconds of CPU time, measured on a 1.2 GHz AMD Athlon computer.

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 5: Run-time of a single MAB with four DHash servers on a WAN. The file system contains four logs.

cates that the most expensive operations in the Ivy and DHash servers are SHA-1 hashes and memory copies.

Each run of MAB produces a directory hierarchy with 1.6 MBytes of data in files. During the MAB Ivy inserts 8.8 MBytes of log records and snapshot data into DHash. The current implementation of Ivy never reclaims this storage; Section 7 discusses the conditions under which this would be desirable, and ways it might be done.

6.2 Performance on a WAN

Table 5 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 milliseconds. 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 milliseconds from the local machine. One log-head is stored on the server with a round-trip time of 82 milliseconds from the local machine. The DHash servers' node IDs were chosen so that each is responsible for roughly the same number of blocks.

Ivy fetches all the log-heads for most NFS operations; however, the log-head that is being modified is cached by the Ivy server. During the experiment, Ivy retrieves each of the other three log-heads 3,346 times. It does so in parallel, but must wait for the slowest reply. The slowest replies come from the DHash server with a round-trip time of 82 milliseconds. Therefore fetching log-heads account for 274 seconds of the benchmark's run time. This cost dominates the performance of Ivy over WAN.

The remaining 102 seconds can be divided four ways. First, a MAB instance running on the same file system on a LAN takes 22 seconds. This accounts for the time spent by the Ivy server, Ivy agent, and the four DHash servers. Second, Ivy writes the log-head 508 times. Each write costs 9 milliseconds. This accounts for 5 seconds. Third,

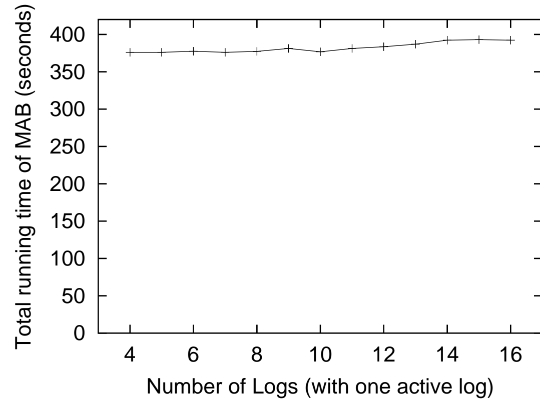

Figure 5: Run-time of a single MAB as a function of the number of Ivy participant logs. Only one participant is actively writing to the file system.

Ivy inserts 1,003 log records, some of them concurrently. Each insertion takes, on average, 54 milliseconds (27 milliseconds, on average, to find the predecessor of the block ID, then another 27 milliseconds, on average, to wait for the node responsible for the block 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 snapshot blocks. Because these insertions are asynchronous, they do not contribute to the total run time.

Table 5 also shows the performance over NFS over UDP; we found that FreeBSD's NFS performed better over UDP than over TCP, even on a WAN. The network round trip time between the NFS client and server was 79 milliseconds, which roughly equals to the time required to fetch all the log-heads. Ivy is slower than NFS because Ivy operations often require more network round-trips; for example, Ivy has to wait for four responses from DHash in order to create a file, one to fetch the log-heads, two to insert the log record, and one to update the log-head.

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 was identical to the one in the previous section. The number of participant logs in the file system ranges from 4 to 16, but only one participant executes the MAB — the other logs are empty.

Figure 5 shows that Ivy scales well as the number of potential writers increases. Because Ivy fetches all log-heads in parallel, time Ivy spends fetching log-heads remains unchanged as the number of participants increases.

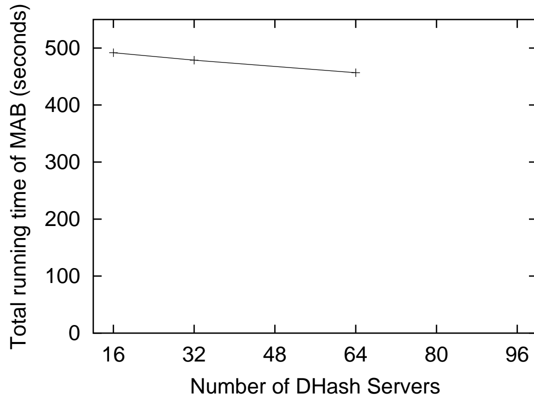

Figure 6: Run-time of one MAB as the number of DHash servers increases.

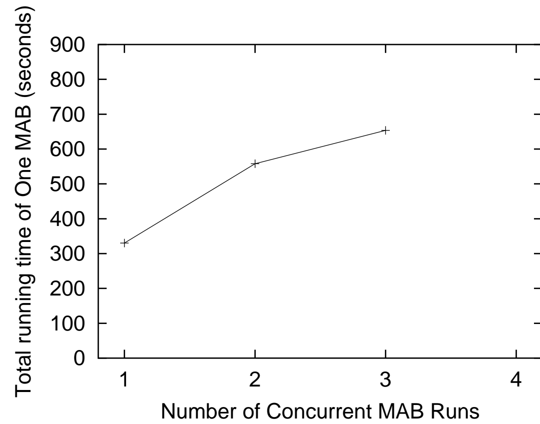

Figure 7: Run-time of one MAB for competing MABs. Each MAB instance reads and writes its own directory.

However, the size of each version vector, and therefore log record, increases. This increases the cost of transferring and storing log records, comparing version vectors, and performing cryptographic operations on log records.

6.4 Many DHash Servers

Figure 6 shows the performance of Ivy as the number of DHash servers increases. XXX number of participants is fixed at 16. XXX describe topology. XXX talk about server selection and virtual DHash nodes. XXX some machines are pegged unfairly, as more blocks may be allocated to them. XXX each experiment uses a different, randomly chosen, set of public keys, so that the allocation of log-heads onto DHash servers changes every experiment. XXX should we use a table to list the number of hops on each insert for each data point?

6.5 Many Writers

Figure 7 shows the performance of Ivy for concurrently running MABs in a setup with a four-participant file system on 32 DHash servers. Each experiment uses a differnt, randomly chosen, set of public keys. The number of concurrent MABs increases from one to four. Each MAB runs in its own directory of the file system, on a separate computer. Figure 7 reports the MAB runtime on one computer.

With one active participant, Ivy fetches all log records from its cache. With more active Ivy participants, MAB runtime increases because most operations require Ivy to fetch other participants' new log records from DHash. Each fetch requires at least two network round-trips: one to search for the predecessor of the block's ID, and one to fetch the block. Furthermore, Ivy spends more time verifying the content hashes of other participants' log records, and comparing their version vectors.

Phase	Ivy (s)	NFS (s)
Commit	51.2	13.0
Update	9.7	3.7

Table 6: Run-time of the commit and update phases of the CVS experiment. DHash is running on four nodes on a wide-area network. Each number is an average over 3 runs.

With more active participants, the amount of time Ivy spends fetching other participants' log records grows slowly because new log records from different participants are sometimes fetched in parallel.

6.6 Frequency of Snapshots

6.7 Wide-area CVS on Ivy

MAB experiments don't reflect Ivy's intended mode of use. In particular, compiling and linking in an Ivy file system is impractical. Ivy's intended use is as a shared repository for source code or documents; users periodically reconcile local copies against the shared repository.

To estimate Ivy's performance as a repository, we measured the run-time of some CVS [4] operations on a CVS repository stored in Ivy. The Ivy file system has four participants over a wide-area DHash system comprised of four nodes—similar to previous experiment. We set up a CVS repository in Ivy containing 40 files or 534 KBytes. Two Ivy participants, X and Y, check out a working version of the repository onto their local disks. We ensure that both participants create a snapshot of the repository. Ivy is re-started to ensure that neither it, nor local NFS clients have any cached data.

The experiment consists of two phases. First, participant X checks in changes that affects 20 files, a total of 2634 lines. The line marked "commit" in Table 6 refers

to this phase. Next, participant Y checks out the changes using a different Ivy node; the table line marked "update" refers to this phase.

For comparison, Table 6 shows measurements for a CVS repository on an NFS server with an 18 millisecond round trip time. As in previous experiments, NFS is faster than Ivy, but Ivy is fast enough to be useful.

7 Future Work

Ivy could be extended and improved in a number of ways; among the most pressing areas are reclamation of log space, tools to help select benign records from the logs of malicious users, more sophisticated conflict resolution tools, and more sophisticated tools to check and restore the integrity of file systems.

The current Ivy implementation keeps all log records indefinitely. This policy is reasonable for its intended use as a repository, since a repository typically does not see a high modification rate. Even file systems containing users' home directories change at relatively slow rates; for example, one deployment of the Venti file system, which takes daily snapshots of all files, consumed 650 Gbytes to store the complete daily record for 10 years for 50 to 100 active users [28]. Venti uses storage space much more efficiently than Ivy; however, Venti demonstrates that maintaining old versions for long periods is not out of question.

In Ivy systems whose logs grow too long to be stored, older log records could be discarded. Participants would then depend on snapshots for old data. This poses two problems of trust. First, a new participant would need to build an initial snapshot based on another participant's snapshot, rather than from the logs. Second, it would not be possible to retroactively ignore changes made by a malicious participant before the date of the earliest preserved log records. Keeping log records for a reasonably long time, such as six months, might be a sound alternative.

As described in Section 4, Ivy has some preliminary tools to deal with file system inconsistencies that result from unavailable log records. Certain inconsistencies cannot be resolved automatically. For example, a user may wish to ignore a subset of the modifications to a file made by a malicious user. To that end, we intend to extend `ivycheck`'s functionality to allow manual repair. This would allow a user to select certain records from the untrusted log to use in the repair. A user interface would provide the user sufficient information to understand the contents of log records.

8 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.

8.1 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.

8.2 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 [38] 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 co-located. Ivy does without any central cluster of dedicated servers—at the expense of strict serial consistency.

8.3 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. Elephant enforces its versioning policy on a per-file granularity. In contrast, Ivy logs operations rather than whole files. Ivy does not implement any storage reclamation policies on log records. Ivy could adopt Elephant's file retention policies to decide when to discard log records.

8.4 Data Consistency and Conflict Resolution

Coda [15, 17] 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 [14] 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 [37] 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.

8.5 Storing Data on Untrusted Servers

BFS [7], OceanStore [16], 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 [26] 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] and S4 [36] use logging to allow modifications by malicious users 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.

9 Conclusion

This paper presented 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.

The key feature of Ivy is complete decentralization. The 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 on read operations. Use of per-participant logs allows Ivy users to choose which other participants to trust.

Due to the 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.

Replication in the DHash peer-to-peer storage system results in high availability of file system data in the face of unreliable servers or networks. Experimental results show that the Ivy prototype is fast enough to use for applications such as the back-end storage for CVS.

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. Trevor Blackwell, Miguel Castro, Josh Cates, Russ Cox, Peter Druschel, Frans Kaashoek, Alex Lewin, and David Mazieres gave us helpful feedback about the design and description of Ivy.

References

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

[2] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[3] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[4] Brian Berliner. CVS II: Parellelizing software development. In *Proceedings of the Winter 1990 USENIX Technical Conference*, Colorado Springs, CO, 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] Migel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[8] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

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

[11] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[12] John H. Hartman and John K. 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 System*, 6(1), February 1988.

[14] Thomas W. Page Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, Los Angeles, CA (USA), 1991.

[15] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[16] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, Boston, MA, November 2000.

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

[18] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

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

[20] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 135–150, San Diego, California, October 2000.

[21] Petar Maymounkov and David Mazirèes. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

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

[23] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999. `http://www.fs.net`.

[24] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, Monterey, California, 2002.

[25] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

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

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

[29] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, San Diego, CA, August 2001.

[30] Dennis M. Ritchie and Ken Thompson. The UNIX timesharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

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

[32] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 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] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Symposium on Operating Systems 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] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, and Craig A.N. Soules. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 165–179, San Diego, California, October 2000.

[37] 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 *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, (Cooper Mountain, Colorado)*, pages 172–183, December 1995.

[38] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[39] Ben Zhao, John Kubiatowicz, and Anthony 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.