Flexible, Wide-Area Storage for Distributed Systems with WheelFS

Jeremy Stribling, Yair Sovran[†] Irene Zhang, Xavid Pretzer,
Jinyang Li[†] M. Frans Kaashoek, and Robert Morris
MIT CSAIL [†]*New York University*

Abstract

WheelFS is a wide-area distributed storage system intended to help multi-site applications share data and gain fault tolerance. WheelFS takes the form of a distributed file system with a familiar POSIX interface. Its design allows applications to adjust the tradeoff between prompt visibility of updates from other sites and the ability for sites to operate independently despite failures and long delays. WheelFS allows these adjustments via *semantic cues*, which provide application control over consistency, failure handling, and file and replica placement.

WheelFS is implemented as a user-level file system and is deployed on PlanetLab and Emulab. Three applications (a distributed Web cache, an email service and large file distribution) demonstrate that WheelFS's file system interface simplifies construction of distributed applications by allowing reuse of existing software. These applications would perform poorly with the strict semantics implied by a traditional file system interface, but by providing cues to WheelFS they are able to achieve good performance. Measurements show that applications built on WheelFS deliver comparable performance to services such as CoralCDN and BitTorrent that use specialized wide-area storage systems.

1 Introduction

There is a growing set of Internet-based services that are too big, or too important, to run at a single site. Examples include Web services for e-mail, video and image hosting, and social networking. Splitting such services over multiple sites can increase capacity, improve fault tolerance, and reduce network delays to clients. These services often need storage infrastructure to share data among the sites. This paper explores the use of a new file system specifically designed to be the storage infrastructure for wide-area distributed services.

A wide-area storage system faces a tension between sharing and site independence. The system must support sharing, so that data stored by one site may be retrieved by others. On the other hand, sharing can be dangerous if it leads to the unreachability of one site causing blocking at other sites, since a primary goal of multi-site operation is fault tolerance. The storage system's consistency model affects the sharing/independence tradeoff: stronger forms of consistency usually involve servers or quorums of servers that serialize all storage operations, whose unreliability may force delays at other sites [23]. The storage system's data and meta-data placement decisions also affect site independence, since data placed at a distant site may be slow to fetch or unavailable.

The wide-area file system introduced in this paper, WheelFS, allows application control over the sharing/independence tradeoff, including consistency, failure handling, and replica placement. Each application can choose a tradeoff between performance and consistency, in the style of PRACTI [8] and PADS [9], but in the context of a file system with a POSIX interface.

Central decisions in the design of WheelFS including defining the default behavior, choosing which behaviors applications can control, and finding a simple way for applications to specify those behaviors. By default, WheelFS provides standard file system semantics (close-to-open consistency) and is implemented similarly to previous wide-area file systems (*e.g.*, every file or directory has a primary storage node). Applications can adjust the default semantics and policies with *semantic cues*. The set of cues is small (around 10) and directly addresses the main challenges of wide-area networks (orders of magnitude differences in latency, lower bandwidth between sites than within a site, and transient failures). WheelFS allows the cues to be expressed in the pathname, avoiding any change to the standard POSIX interface. The benefits of WheelFS providing a file system interface are compatibility with existing software and programmer ease-of-use.

A prototype of WheelFS runs on FreeBSD, Linux, and MacOS. The client exports a file system to local applications using FUSE [21]. WheelFS runs on PlanetLab and an emulated wide-area Emulab network.

Several distributed applications run on WheelFS and demonstrate its usefulness, including a distributed Web cache and a multi-site email service. The applications use different cues, showing that the control that cues provide is valuable. All were easy to build by reusing existing software components, with WheelFS for storage instead of a local file system. For example, the Apache caching web proxy can be turned into a distributed, cooperative Web cache by modifying one pathname in a configuration file, specifying that Apache should store cached data in WheelFS with cues to relax consistency. Although the other applications require more changes, the ease of adapting Apache illustrates the value of a file system interface; the extent to which we could reuse non-distributed software in distributed applications came as a surprise [38].

Measurements show that WheelFS offers more scalable performance on PlanetLab than an implementation of NFSv4, and that for applications that use cues to indicate they can tolerate relaxed consistency, WheelFS continues to provide high performance in the face of network and server failures. For example, by using the cues **.EventualConsistency**, **.MaxTime**, and **.Hotspot**, the distributed Web cache quickly reduces the load on the origin Web server, and the system hardly pauses serving pages when WheelFS nodes fail; experiments on PlanetLab show that the WheelFS-based distributed Web cache reduces origin Web server load to zero. Further experiments on Emulab show that WheelFS can offer better file downloads times than BitTorrent [14] by using network coordinates to download from the caches of nearby clients.

The main contributions of this paper are a new file system that assists in the construction of wide-area distributed applications, a set of cues that allows applications to control the file system's consistency and availability tradeoffs, and a demonstration that wide-area applications can achieve good performance and failure behavior by using WheelFS.

The rest of the paper is organized as follows. Sections 2 and 3 outline the goals of WheelFS and its overall design. Section 4 describes WheelFS's cues, and Section 5 presents WheelFS's detailed design. Section 6 illustrates some example applications, Section 7 describes the implementation of WheelFS, and Section 8 measures the performance of WheelFS and the applications. Section 9 discusses related work, and Section 10 concludes.

2 Goals

A wide-area storage system must have a few key properties in order to be practical. It must be a useful building block for larger applications, presenting an easy-to-use interface and shouldering a large fraction of the overall storage management burden. It must allow inter-site access to data when needed, as long as the health of the wide-area network allows. When the site storing some data is not reachable, the storage system must indicate a failure (or find another copy) with relatively low delay, so that a failure at one site does not prevent progress at other sites. Finally, applications may need to control the site(s) at which data are stored in order to achieve fault-tolerance and performance goals.

As an example, consider a distributed Web cache whose primary goal is to reduce the load on the origin servers of popular pages. Each participating site runs a Web proxy and a part of a distributed storage system. When a Web proxy receives a request from a browser, it first checks to see if the storage system has a copy of the requested page. If it does, the proxy reads the page from the storage system (perhaps from another site) and serves it to the browser. If not, the proxy fetches the page from the origin Web server, inserts a copy of it into the storage system (so other proxies can find it), and sends it to the browser.

The Web cache requires some specific properties from the distributed storage system in addition to the general ability to store and retrieve data. A proxy must serve data with low delay, and can consult the origin Web server if it cannot find a cached copy; thus it is preferable for the storage system to indicate "not found" quickly if finding the data would take a long time (due to timeouts). The storage need not be durable or highly fault tolerant, again because proxies can fall back on the origin Web server. The storage system need not be consistent in the sense of guaranteeing to find the latest stored version of document, since HTTP headers allow a proxy to evaluate whether a cached copy is still valid.

Other distributed applications might need different properties in a storage system: they might need to see the latest copy of some data, and be willing to pay a price in high delay, or they may want data to be stored durably, or have specific preferences for which site stores a document. Thus, in order to be a usable component in many different systems, a distributed storage system needs to expose a level of control to the surrounding application.

3 WheelFS Overview

This section gives a brief overview of WheelFS to help the reader follow the design proposed in subsequent sections.

3.1 System Model

WheelFS is intended to be used by distributed applications that run on a collection of sites distributed over the wide-area Internet. All nodes in a WheelFS deployment are either managed by a single administrative entity or multiple cooperating administrative entities. WheelFS's security goals are limited to controlling the set of participating servers and imposing UNIX-like access controls on clients; it does not guard against Byzantine failures in participating servers [6, 26]. We expect servers to be live and reachable most of the time, with occasional failures. Many existing distributed infrastructures fit these assumptions, such as wide-area testbeds (*e.g.*, PlanetLab and RON), collections of data centers spread across the globe (*e.g.*, Amazon's EC2), and federated resources such as Grids.

3.2 System Overview

WheelFS provides a location-independent hierarchy of directories and files with a POSIX file system interface. At any given time, every file or directory object has a single "primary" WheelFS storage server that is responsible for maintaining the latest contents of that object. WheelFS clients, acting on behalf of applications, use the storage servers to retrieve and store data. By default, clients consult the primary whenever they modify an object or need to find the latest version of an object. Accessing a single file could result in communication with several servers, since each subdirectory in the path could be served by a different primary. WheelFS replicates an object's data using primary/backup replication, and a background maintenance process running on each server ensures that data are replicated correctly. Each update to an object increments a version number kept in a separate meta-data structure, co-located with the data.

When a WheelFS client needs to use an object, it must first determine which server is currently the primary for that object. All nodes agree on the assignment of objects to primaries to help implement the default strong consistency. Nodes learn the assignment from a *configuration service*—a replicated state machine running at multiple sites. This service maintains a table that maps each object to one primary and zero or more backup servers. WheelFS nodes cache a copy of this table. Section 5 presents the design of the configuration service.

A WheelFS client reads a file's data in blocks from the file's primary server. The client caches the file's data once read, obtaining a lease on its meta-data (including the version number) from the primary. Clients have the option of reading from other clients' caches, which can be helpful for large and popular files that are rarely updated. WheelFS provides close-to-open consistency by default for files, so that if an application works correctly on a POSIX file system, it will also work correctly on WheelFS.

4 Semantic cues

WheelFS provides semantic cues within the standard POSIX file system API. We believe cues would also be useful in the context of other wide-area storage layers with alternate designs, such as Shark [6] or a wide-area version of BigTable [13]. This section describes how applications specify cues and what effect they have on file system operations.

4.1 Specifying cues

Applications specify cues to WheelFS in pathnames; for example, /wfs/**.Cue**/data refers to /wfs/data with the cue **.Cue**. The main advantage of embedding cues in pathnames is that it keeps the POSIX interface unchanged. This choice allows developers to program using an interface with which they are familiar and to reuse software easily.

One disadvantage of cues is that they may break software that parses pathnames and assumes that a cue is a directory. Another is that links to pathnames that contain cues may trigger unintuitive behavior. We have not encountered examples of these problems.

WheelFS clients process the cue path components locally. A pathname might contain several cues, separated by slashes. WheelFS uses the following rules to combine cues: (1) a cue applies to all files and directories in the pathname appearing after the cue; and (2) cues that are specified later in a pathname may override cues in the same category appearing earlier.

As a preview, a distributed Web cache could be built by running a caching Web proxy at each of a number of sites, sharing cached pages via WheelFS. The proxies could store pages in pathnames such as /wfs/**.MaxTime**=200/url, causing `open()` to fail after 200 ms rather than waiting for an unreachable WheelFS server, indicating to the proxy that it should fetch from the original Web server. See Section 6 for a more sophisticated version of this application.

4.2 Categories

Table 1 lists WheelFS's cues and the categories into which they are grouped. There are four categories: placement, durability, consistency, and large reads. These categories reflect the goals discussed in Section 2. The placement cues allow an application to reduce latency by placing data near where it will be needed. The durability and consistency cues help applications avoid data unavailability and timeout delays caused by transient failures. The large read cues increase throughput when reading large and/or popular files. Table 2 shows which POSIX file system API calls are affected by which of these cues.

Each cue is either *persistent* or *transient*. A persistent cue is permanently associated with the object, and may affect all uses of the object, including references that do not specify the cue. An application associates a persistent cue with an object by specifying the cue when first creating the object. Persistent cues are immutable after object creation. If an application specifies a transient cue in a file system operation, the cue only applies to that operation.

Because these cues correspond to the challenges faced by wide-area applications, we consider this set of cues to be relatively complete. These cues work well for the applications we have considered.

4.3 Placement

Applications can reduce latency by storing data near the clients who are likely to use that data. For example, a wide-area email system may wish to store all of a user's message files at a site near that user.

The **.Site=X** cue indicates the desired site for a newly-created file's primary. The site name can be a simple string, *e.g.* **.Site=westcoast**, or a domain name such as

Cue Category	Cue Name	Type	Meaning (and Tradeoffs)
Placement	**.Site=X**	P	Store files and directories on a server at the site named X.
	.KeepTogether	P	Store all files in a directory subtree on the same set of servers.
	.RepSites=N$_{RS}$	P	Store replicas across N$_{RS}$ different sites.
Durability	**.RepLevel=N$_{RL}$**	P	Keep N$_{RL}$ replicas for a data object.
	.SyncLevel=N$_{SL}$	T	Wait for only N$_{SL}$ replicas to accept a new file or directory version, reducing both durability and delay.
Consistency	**.EventualConsistency**	T*	Use potentially stale cached data, or data from a backup, if the primary does not respond quickly.
	.MaxTime=T	T	Limit any WheelFS remote communication done on behalf of a file system operation to no more than T ms.
Large reads	**.WholeFile**	T	Enable pre-fetching of an entire file upon the first read request.
	.Hotspot	T	Fetch file data from other clients' caches to reduce server load. Fetches multiple blocks in parallel if used with **.WholeFile**.

Table 1: Semantic cues. A cue can be either **P**ersistent or **T**ransient (*Section 4.5 discusses a caveat for **.EventualConsistency**).

Cue	open	close	read	write	stat	mkdir	rmdir	link	unlink	readdir	chmod
.S	X					X					
.KT	X					X					
.RS	X	X		X		X	X	X	X		X
.RL	X	X		X		X	X	X	X		X
.SL	X	X		X		X	X	X	X		X
.EC	X	X	X	X	X	X	X	X	X	X	X
.MT	X	X	X	X	X	X	X	X	X	X	X
.WF			X							X	
.H			X								

Table 2: The POSIX file system API calls affected by each cue.

.Site=rice.edu. An administrator configures the correspondence between site names and servers. If the path contains no **.Site** cue, WheelFS uses the local node's site as the file's primary. Use of **random** as the site name will spread newly created files over all sites. If the site indicated by **.Site** is unreachable, or cannot store the file due to storage limitations, WheelFS stores the newly created file at another site, chosen at random. The WheelFS background maintenance process will eventually transfer the misplaced file to the desired site.

The **.KeepTogether** cue indicates that an entire subtree should reside on as few WheelFS nodes as possible. Clustering a set of files can reduce the delay for operations that access multiple files. For example, an email system can store a user's message files on a few nodes to reduce the time required to list all messages.

The **.RepSites=N$_{RS}$** cue indicates how many different sites should have copies of the data. **N$_{RS}$** only has an effect when it is less than the replication level (see Section 4.4), in which case it causes one or more sites to store the data on more than one local server. When possible, WheelFS ensures that the primary's site is one of the sites chosen to have an extra copy. For example, specifying **.RepSites=2** with a replication level of three causes the primary and one backup to be at one site, and another backup to be at a different site. By using **.Site** and **.RepSites**, an application can ensure that a permanently failed primary can be reconstructed at the desired site with only local communication.

4.4 Durability

WheelFS allows applications to express durability preferences with two cues: **.RepLevel=N$_{RL}$** and **.SyncLevel=N$_{SL}$**.

The **.RepLevel=N$_{RL}$** cue causes the primary to store the object on N$_{RL}-1$ backups; by default, N$_{RL}=3$. The WheelFS prototype imposes a maximum of four replicas (see Section 5.2 for the reason for this limit; in a future prototype it will most likely be higher).

The **.SyncLevel=N$_{SL}$** cue causes the primary to wait for acknowledgments of writes from only N$_{SL}$ of the object's replicas before acknowledging the client's request, reducing durability but also reducing delays if some backups are slow or unreachable. By default, N$_{SL}$ = N$_{RL}$.

4.5 Consistency

The **.EventualConsistency** cue allows a client to use an object despite unreachability of the object's primary node, and in some cases the backups as well. For reads and pathname lookups, the cue allows a client to read from a backup if the primary is unavailable, and from the client's local cache if the primary and backups are both unavailable. For writes and filename creation, the cue allows a client to write to a backup if the primary is not available. A consequence of **.EventualConsistency** is that clients may not see each other's updates if they cannot all reliably contact the primary. Many applications such as Web caches and email systems can tolerate eventual consistency without significantly compromising their users' experience, and in return can decrease delays and reduce service unavailability when a primary or its network link are unreliable.

The cue provides eventual consistency in the sense that, in the absence of updates, all replicas of an object will eventually converge to be identical. However, WheelFS does not provide eventual consistency in the rigorous form (*e.g.*, [18]) used by systems like Bayou [39], where all updates, across all objects in the system, are committed in a total order at all replicas. In particular, updates in WheelFS are only eventually consistent with respect to the object they affect, and updates may potentially be lost. For example, if an entry is deleted from a directory under the **.EventualConsistency** cue, it could reappear in the directory later.

When reading files or using directory contents with eventual consistency, a client may have a choice between the contents of its cache, replies from queries to one or more backup servers, and a reply from the primary. A client uses the data with the highest version number that it finds within a time limit. The default time limit is one second, but can be changed with the **.MaxTime=T** cue (in units of milliseconds). If **.MaxTime** is used without eventual consistency, the WheelFS client yields an error if it cannot contact the primary after the indicated time.

The background maintenance process periodically reconciles a primary and its backups so that they eventually contain the same data for each file and directory. The process may need to resolve conflicting versions of objects. For a file, the process chooses arbitrarily among the replicas that have the highest version number; this may cause writes to be lost. For an eventually-consistent directory, it puts the union of files present in the directory's replicas into the reconciled version. If a single filename maps to multiple IDs, the process chooses the one with the smallest ID and renames the other files. Enabling directory merging is the only sense in which the **.EventualConsistency** cue is persistent: if specified at directory creation time, it guides the conflict resolution process. Otherwise its effect is specific to particular references.

4.6 Large reads

WheelFS provides two cues that enable large-file read optimizations: **.WholeFile** and **.Hotspot**. The **.WholeFile** cue instructs WheelFS to pre-fetch the entire file into the client cache. The **.Hotspot** cue instructs the WheelFS client to read the file from other clients' caches, consulting the file's primary for a list of clients that likely have the data cached. If the application specifies both cues, the client will read data in parallel from other clients' caches.

Unlike the cues described earlier, **.WholeFile** and **.Hotspot** are not strictly necessary: a file system could potentially learn to adopt the right cue by observing application access patterns. We leave such adaptive behavior to future work.

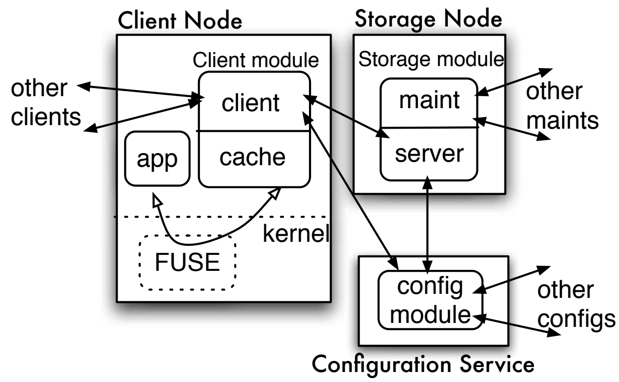

Figure 1: Placement and interaction of WheelFS components.

5 WheelFS Design

WheelFS requires a design flexible enough to follow the various cues applications can supply. This section presents that design, answering the following questions:

- How does WheelFS assign storage responsibility for data objects among participating servers? (Section 5.2)
- How does WheelFS ensure an application's desired level of durability for its data? (Section 5.3)
- How does WheelFS provide close-to-open consistency in the face of concurrent file access and failures, and how does it relax consistency to improve availability? (Section 5.4)
- How does WheelFS permit peer-to-peer communication to take advantage of nearby cached data? (Section 5.5)
- How does WheelFS authenticate users and perform access control? (Section 5.6)

5.1 Components

A WheelFS deployment (see Figure 1) consists of clients and servers; a single host often plays both roles. The WheelFS client software uses FUSE [21] to present the distributed file system to local applications, typically in `/wfs`. All clients in a given deployment present the same file system tree in `/wfs`. A WheelFS client communicates with WheelFS servers in order to look up file names, create files, get directory listings, and read and write files. Each client keeps a local cache of file and directory contents.

The configuration service runs independently on a small set of wide-area nodes. Clients and servers communicate with the service to learn the set of servers and which files and directories are assigned to which servers, as explained in the next section.

5.2 Data storage assignment

WheelFS servers store file and directory objects. Each object is internally named using a unique numeric ID. A file object contains opaque file data and a directory object contains a list of name-to-object-ID mappings for the directory contents. WheelFS partitions the object ID space into 2^S slices using the first S bits of the object ID.

The configuration service maintains a *slice table* that lists, for each slice currently in use, a *replication policy* governing the slice's data placement, and a *replica list* of servers currently responsible for storing the objects in that slice. A replication policy for a slice indicates from which site it must choose the slice's primary (**.Site**), and from how many distinct sites (**.RepSites**) it must choose how many backups (**.RepLevel**). The replica list contains the current primary for a slice, and $N_{RL}-1$ backups.

Because each unique replication policy requires a unique slice identifier, the choice of S limits the maximum allowable number of replicas in a policy. In our current implementation S is fairly small (12 bits), and so to conserve slice identifiers it limits the maximum number of replicas to four.

5.2.1 Configuration service

The configuration service is a replicated state machine, and uses Paxos [25] to elect a new master whenever its membership changes. Only the master can update the slice table; it forwards updates to the other members. A WheelFS node is initially configured to know of at least one configuration service member, and contacts it to learn the full list of members and which is the master.

The configuration service exports a lock interface to WheelFS servers, inspired by Chubby [11]. Through this interface, servers can `acquire`, `renew`, and `release` locks on particular slices, or `fetch` a copy of the current slice table. A slice's lock grants the exclusive right to be a primary for that slice, and the right to specify the slice's backups and (for a new slice) its replication policy. A lock automatically expires after L seconds unless renewed. The configuration service makes no decisions about slice policy or replicas. Section 5.3 explains how WheelFS servers use the configuration service to recover after the failure of a slice's primary or backups.

Clients and servers periodically fetch and cache the slice table from the configuration service master. A client uses the slice table to identify which servers should be contacted for an object in a given slice. If a client encounters an object ID for which its cached slice table does not list a corresponding slice, the client fetches a new table. A server uses the the slice table to find other servers that store the same slice so that it can synchronize with them.

Servers try to always have at least one slice locked, to guarantee they appear in the table of currently locked slices; if the maintenance process notices that the server holds no locks, it will acquire the lock for a new slice. This allows any connected node to determine the current membership of the system by taking the union of the replica lists of all slices.

5.2.2 Placing a new file or directory

When a client creates a new file or directory, it uses the placement and durability cues specified by the application to construct an appropriate replication policy. If **.KeepTogether** is present, it sets the primary site of the policy to be the primary site of the object's parent directory's slice. Next the client checks the slice table to see if an existing slice matches the policy; if so, the client contacts the primary replica for that slice. If not, it forwards the request to a random server at the site specified by the **.Site** cue.

When a server receives a request asking it to create a new file or directory, it constructs a replication policy as above, and sets its own site to be the primary site for the policy. If it does not yet have a lock on a slice matching the policy, it generates a new, randomly-generated slice identifier and constructs a replica list for that slice, choosing from the servers listed in the slice table. The server then acquires a lock on this new slice from the configuration service, sending along the replication policy and the replica list. Once it has a lock on an appropriate slice, it generates an object ID for the new object, setting the first S bits to be the slice ID and all other bits to random values. The server returns the new ID to the client, and the client then instructs the object's parent directory's primary to add a new entry for the object. Other clients that learn about this new object ID from its entry in the parent directory can use the first S bits of the ID to find the primary for the slice and access the object.

5.2.3 Write-local policy

The default data placement policy in WheelFS is to *write locally*, *i.e.*, use a local server as the primary of a newly created file (and thus also store one copy of the contents locally). This policy works best if each client also runs a WheelFS server. The policy allows writes of large non-replicated files at the speed of the local disk, and allows such files to be written at one site and read at another with just one trip across the wide-area network.

Modifying an existing file is not always fast, because the file's primary might be far away. Applications desiring fast writes should store output in unique new files, so that the local server will be able to create a new object ID in a slice for which it is the primary. Existing software often works this way; for example, the Apache caching proxy stores a cached Web page in a unique file named after the page's URL.

An ideal default placement policy would make decisions based on server loads across the entire system; for example, if the local server is nearing its storage capacity but a neighbor server at the same site is underloaded, WheelFS might prefer writing the file to the neighbor rather than the local disk (*e.g.*, as in Porcupine [31]). Developing such a strategy is future work; for now, applications can use cues to control where data are stored.

5.3 Primary/backup replication

WheelFS uses primary/backup replication to manage replicated objects. The slice assignment designates, for each ID slice, a primary and a number of backup servers. When a client needs to read or modify an object, by default it communicates with the primary. For a file, a modification is logically an entire new version of the file contents; for a directory, a modification affects just one entry. The primary forwards each update to the backups, after which it writes the update to its disk and waits for the write to complete. The primary then waits for replies from $\mathrm{N}_{SL}-1$ backups, indicating that those backups have also written the update to their disks. Finally, the primary replies to the client. For each object, the primary executes operations one at a time.

After being granted the lock on a slice initially, the WheelFS server must renew it periodically; if the lock expires, another server may acquire it to become the primary for the slice. Since the configuration service only grants the lock on a slice to one server at a time, WheelFS ensures that only one server will act as a primary for a slice at any given time. The slice lock time L is a compromise: short lock times lead to fast reconfiguration, while long lock times allow servers to operate despite the temporary unreachability of the configuration service.

In order to detect failure of a primary or backup, a server pings all other replicas of its slices every five minutes. If a primary decides that one of its backups is unreachable, it chooses a new replica from the same site as the old replica if possible, otherwise from a random site. The primary will transfer the slice's data to this new replica (blocking new updates), and then renew its lock on that slice along with a request to add the new replica to the replica list in place of the old one.

If a backup decides the primary is unreachable, it will attempt to acquire the lock on the slice from the configuration service; one of the backups will get the lock once the original primary's lock expires. The new primary checks with the backups to make sure that it didn't miss any object updates (*e.g.*, because $\mathrm{N}_{SL}<\mathrm{N}_{RL}$ during a recent update, and thus not all backups are guaranteed to have committed that update).

A primary's maintenance process periodically checks that the replicas associated with each slice match the slice's policy; if not, it will attempt to recruit new replicas at the appropriate sites. If the current primary wishes to recruit a new primary at the slice's correct primary site (*e.g.*, a server that had originally been the slice's primary but crashed and rejoined), it will release its lock on the slice, and directly contact the chosen server, instructing it to acquire the lock for the slice.

5.4 Consistency

By default, WheelFS provides close-to-open consistency: if one application instance writes a file and waits for `close()` to return, and then a second application instance `open()`s and reads the file, the second application will see the effects of the first application's writes. The reason WheelFS provides close-to-open consistency by default is that many applications expect it.

The WheelFS client has a write-through cache for file blocks, for positive and negative directory entries (enabling faster pathname lookups), and for directory and file meta-data. A client must acquire an *object lease* from an object's primary before it uses cached meta-data. Before the primary executes any update to an object, it must invalidate all leases or wait for them to expire. This step may be time-consuming if many clients hold leases on an object.

Clients buffer file writes locally to improve performance. When an application calls `close()`, the client sends all outstanding writes to the primary, and waits for the primary to acknowledge them before allowing `close()` to return. Servers maintain a version number for each file object, which they increment after each `close()` and after each change to the object's meta-data.

When an application `open()`s a file and then reads it, the WheelFS client must decide whether the cached copy of the file (if any) is still valid. The client uses cached file data if the object version number of the cached data is the same as the object's current version number. If the client has an unexpired object lease for the object's meta-data, it can use its cached meta-data for the object to find the current version number. Otherwise it must contact the primary to ask for a new lease, and for current meta-data. If the version number of the cached data is not current, the client fetches new file data from the primary.

By default, WheelFS provides similar consistency for directory operations: after the return of an application system call that modifies a directory (links or unlinks a file or subdirectory), applications on other clients are guaranteed to see the modification. WheelFS clients implement this consistency by sending directory updates to the directory object's primary, and by ensuring via lease or explicit check with the primary that cached directory contents are up to date. Cross-directory rename operations in WheelFS are not atomic with respect to failures. If a crash occurs at the wrong moment, the result may be a link to the moved file in both the source and destination directories.

The downside to close-to-open consistency is that if a primary is not reachable, all operations that consult the primary will delay until it revives or a new primary takes over. The **.EventualConsistency** cue allows WheelFS to avoid these delays by using potentially stale data from backups or local caches when the primary does not respond, and by sending updates to backups. This can result in inconsistent replicas, which the maintenance process resolves in the manner described in Section 4.5, leading eventually to identical images at all replicas. Without the **.EventualConsistency** cue, a server will reject operations on objects for which it is not the primary.

Applications can specify timeouts on a per-object basis using the **.MaxTime=T** cue. This adds a timeout of T ms to every operation performed at a server. Without **.EventualConsistency**, a client will return a failure to the application if the primary does not respond within T ms; with **.EventualConsistency**, clients contact backup servers once the timeout occurs. In future work we hope to explore how to best divide this timeout when a single file system operation might involve contacting several servers (*e.g.*, a create requires talking to the parent directory's primary and the new object's primary, which could differ).

5.5 Large reads

If the application specifies **.WholeFile** when reading a file, the client will pre-fetch the entire file into its cache. If the application uses **.WholeFile** when reading directory contents, WheelFS will pre-fetch the meta-data for all of the directory's entries, so that subsequent lookups can be serviced from the cache.

To implement the **.Hotspot** cue, a file's primary maintains a soft-state list of clients that have recently cached blocks of the file, including which blocks they have cached. A client that reads a file with **.Hotspot** asks the server for entries from the list that are near the client; the server chooses the entries using Vivaldi coordinates [15]. The client uses the list to fetch each block from a nearby cached copy, and informs the primary of successfully fetched blocks.

If the application reads a file with both **.WholeFile** and **.Hotspot**, the client will issue block fetches in parallel to multiple other clients. It pre-fetches blocks in a random order so that clients can use each others' caches even if they start reading at the same time [6].

5.6 Security

WheelFS enforces three main security properties. First, a given WheelFS deployment ensures that only authorized hosts participate as servers. Second, WheelFS ensures that requests come only from users authorized to use the deployment. Third, WheelFS enforces user-based permissions on requests from clients. WheelFS assumes that authorized servers behave correctly. A misbehaving client can act as any user that has authenticated themselves to WheelFS from that client, but can only do things for which those users have permission.

All communication takes place through authenticated SSH channels. Each authorized server has a public/private key pair which it uses to prove its identity. A central administrator maintains a list of all legitimate server public keys in a deployment, and distributes that list to every server and client. Servers only exchange inter-server traffic with hosts authenticated with a key on the list, and clients only send requests to (and use responses from) authentic servers.

Each authorized user has a public/private key pair; WheelFS uses SSH's existing key management support. Before a user can use WheelFS on a particular client, the user must reveal his or her private key to the client. The list of authorized user public keys is distributed to all servers and clients as a file in WheelFS. A server accepts only client connections signed by an authorized user key. A server checks that the authenticated user for a request has appropriate permissions for the file or directory being manipulated—each object has an associated access control list in its meta-data. A client dedicated to a particular distributed application stores its "user" private key on its local disk.

Clients check data received from other clients against server-supplied SHA-256 checksums to prevent clients from tricking each other into accepting unauthorized modifications. A client will not supply data from its cache to another client whose authorized user does not have read permissions.

There are several planned improvements to this security setup. One is an automated mechanism for propagating changes to the set of server public keys, which currently need to be distributed manually. Another is to allow the use of SSH Agent forwarding to allow users to connect securely without storing private keys on client hosts, which would increase the security of highly privileged keys in the case where a client is compromised.

6 Applications

WheelFS is designed to help the construction of wide-area distributed applications, by shouldering a significant part of the burden of managing fault tolerance, consistency, and sharing of data among sites. This section evaluates how well WheelFS fulfills that goal by describing four applications that have been built using it.

All-Pairs-Pings. All-Pairs-Pings [37] monitors the network delays among a set of hosts. Figure 2 shows a simple version of All-Pairs-Pings built from a shell script and WheelFS, to be invoked by each host's `cron` every few minutes. The script pings the other hosts and puts the results in a file whose name contains the local host name

```
FILE=`date +%s`.`hostname`.dat
D=/wfs/ping
BIN=$D/bin/.EventualConsistency/
.MaxTime=5000/.HotSpot/.WholeFile
DATA=$D/.EventualConsistency/dat
mkdir -p $DATA/`hostname`
cd $DATA/`hostname`
xargs -n1 $BIN/ping -c 10 <
 $D/nodes > /tmp/$FILE
cp /tmp/$FILE $FILE
rm /tmp/$FILE
if [ `hostname` = "node1" ]; then
 mkdir -p $D/res
 $BIN/process * > $D/res/`date +%s`.o
fi
```

Figure 2: A shell script implementation of All-Pairs-Pings using WheelFS.

and the current time. After each set of pings, a coordinator host ("node1") reads all the files, creates a summary using the program `process` (not shown), and writes the output to a results directory.

This example shows that WheelFS can help keep simple distributed tasks easy to write, while protecting the tasks from failures of remote nodes. WheelFS stores each host's output on the host's own WheelFS server, so that hosts can record ping output even when the network is broken. WheelFS automatically collects data files from hosts that reappear after a period of separation. Finally, WheelFS provides each host with the required binaries and scripts and the latest host list file. Use of WheelFS in this script eliminates much of the complexity of a previous All-Pairs-Pings program, which explicitly dealt with moving files among nodes and coping with timeouts.

Distributed Web cache. This application consists of hosts running Apache 2.2.4 caching proxies (`mod_disk_cache`). The Apache configuration file places the cache file directory on WheelFS:

`/wfs/`**.EventualConsistency/.MaxTime=1000/**
.Hotspot`/cache/`

When the Apache proxy can't find a page in the cache directory on WheelFS, it fetches the page from the origin Web server and writes a copy in the WheelFS directory, as well as serving it to the requesting browser. Other cache nodes will then be able to read the page from WheelFS, reducing the load on the origin Web server. The **.Hotspot** cue copes with popular files, directing the WheelFS clients to fetch from each others' caches to increase total throughput. The **.EventualConsistency** cue allows clients to create and read files even if they cannot contact the primary server. The **.MaxTime** cue instructs WheelFS to return an error if it cannot find a file quickly, causing Apache to fetch the page from the origin Web server. If WheelFS returns an expired version of the file, Apache will notice by checking the HTTP header in the cache file, and it will contact the origin Web server for a fresh copy.

Although this distributed Web cache implementation is fully functional, it does lack features present in other similar systems. For example, CoralCDN uses a hierarchy of caches to avoid overloading any single tracker node when a file is popular.

Mail service. The goal of Wheemail, our WheelFS-based mail service, is to provide high throughput by spreading the work over many sites, and high availability by replicating messages on multiple sites. Wheemail provides SMTP and IMAP service from a set of nodes at these sites. Any node at any site can accept a message via SMTP for any user; in most circumstances a user can fetch mail from the IMAP server on any node.

Each node runs an unmodified sendmail process to accept incoming mail. Sendmail stores each user's messages in a WheelFS directory, one message per file. The separate files help avoid conflicts from concurrent message arrivals. A user's directory has this path:

`/wfs/mail/`**.EventualConsistency**`/`**.Site=X**`/`
.KeepTogether`/`**.RepSites=2**`/`*user*`/Mail/`

Each node runs a Dovecot IMAP server [17] to serve users their messages. A user retrieves mail via a nearby node using a locality-preserving DNS service [20].

The **.EventualConsistency** cue allows a user to read mail via backup servers when the primary for the user's directory is unreachable, and allows incoming mail to be stored even if primary and all backups are down. The **.Site=X** cue indicates that a user's messages should be stored at site X, chosen to be close to the user's usual location to reduce network delays. The **.KeepTogether** cue causes all of a user's messages to be stored on a single replica set, reducing latency for listing the user's messages [31]. Wheemail uses the default replication level of three but uses **.RepSites=2** to keep at least one off-site replica of each mail. To avoid unnecessary replication, Dovecot uses **.RepLevel=1** for much of its internal data.

Wheemail has goals similar to those of Porcupine [31], namely, to provide scalable email storage and retrieval with high availability. Unlike Porcupine, Wheemail runs on a set of wide-area data centers. Replicating emails over multiple sites increases the service's availability when a single site goes down. Porcupine consists of custom-built storage and retrieval components. In contrast, the use of a wide-area file system in Wheemail allows it to reuse existing software like sendmail and Dovecot. Both Porcupine and Wheemail use eventual consistency to increase availability, but Porcupine has a better reconciliation policy as its “deletion record” prevents deleted emails from reappearing.

File Distribution. A set of many WheelFS clients can cooperate to fetch a file efficiently using the large read cues:

`/wfs/`**.WholeFile**`/`**.Hotspot**`/largefile`

Efficient file distribution may be particularly useful for binaries in wide-area experiments, in the spirit of Shark [6] and CoBlitz [29]. Like Shark, WheelFS uses cooperative caching to reduce load on the file server. Shark further reduces the load on the file server by using a distributed index to keep track of cached copies, whereas WheelFS relies on the primary server to track copies. Unlike WheelFS or Shark, CoBlitz is a CDN, so files cannot be directly accessed through a mounted file system. CoBlitz caches and shares data between CDN nodes rather than between clients.

7 Implementation

The WheelFS prototype consists of 19,000 lines of C++ code, using pthreads and STL. In addition, the implementation uses a new RPC library (3,800 lines) that implements Vivaldi network coordinates [15].

The WheelFS client uses FUSE’s “low level” interface to get access to FUSE identifiers, which it translates into WheelFS-wide unique object IDs. The WheelFS cache layer in the client buffers writes in memory and caches file blocks in memory and on disk.

Permissions, access control, and secure SSH connections are implemented. Distribution of public keys through WheelFS is not yet implemented.

8 Evaluation

This section demonstrates the following points about the performance and behavior of WheelFS:

- For some storage workloads common in distributed applications, WheelFS offers more scalable performance than an implementation of NFSv4.
- WheelFS achieves reasonable performance under a range of real applications running on a large, wide-area testbed, as well as on a controlled testbed using an emulated network.
- WheelFS provides high performance despite network and server failures for applications that indicate via cues that they can tolerate relaxed consistency.
- WheelFS offers data placement options that allow applications to place data near the users of that data, without the need for special application logic.
- WheelFS offers client-to-client read options that help counteract wide-area bandwidth constraints.
- WheelFS offers an interface on which it is quick and easy to build real distributed applications.

8.1 Experimental setup

All scenarios use WheelFS configured with 64 KB blocks, a 100 MB in-memory client LRU block cache supplemented by an unlimited on-disk cache, one minute object leases, a lock time of $L = 2$ minutes, 12-bit slice IDs, 32-bit object IDs, and a default replication level of three (the responsible server plus two replicas), unless stated otherwise. Communication takes place over plain TCP, not SSH, connections. Each WheelFS node runs both a storage server and a client process. The configuration service runs on five nodes distributed across three wide-area sites.

We evaluate our WheelFS prototype on two testbeds: PlanetLab [7] and Emulab [42]. For PlanetLab experiments, we use up to 250 nodes geographically spread across the world at more than 140 sites (we determine the site of a node based on the domain portion of its hostname). These nodes are shared with other researchers and their disks, CPU, and bandwidth are often heavily loaded, showing how WheelFS performs in the wild. These nodes run a Linux 2.6 kernel and FUSE 2.7.3. We run the configuration service on a private set of nodes running at MIT, NYU, and Stanford, to ensure that the replicated state machine can log operations to disk and respond to requests quickly (`fsync()`s on PlanetLab nodes can sometimes take tens of seconds).

For more control over the network topology and host load, we also run experiments on the Emulab [42] testbed. Each Emulab host runs a standard Fedora Core 6 Linux 2.6.22 kernel and FUSE version 2.6.5, and has a 3 GHz CPU. We use a WAN topology consisting of 5 LAN clusters of 3 nodes each. Each LAN cluster has 100 Mbps, sub-millisecond links between each node. Clusters connect to the wide-area network via a single bottleneck link of 6 Mbps, with 100 ms RTTs between clusters.

8.2 Scalability

We first evaluate the scalability of WheelFS on a microbenchmark representing a workload common to distributed applications: many nodes reading data written by other nodes in the system. For example, nodes running a distributed Web cache over a shared storage layer would be reading and serving pages written by other nodes. In this microbenchmark, N clients mount a shared file system containing N directories, either using NFSv4 or WheelFS. Each directory contains ten 1 MB files. The clients are PlanetLab nodes picked at random from the set of nodes that support both mounting both FUSE and NFS file systems. This set spans a variety of nodes distributed across the world, from nodes at well-connected educational institutions to nodes behind limited-upload DSL lines. Each client reads ten random files from the file

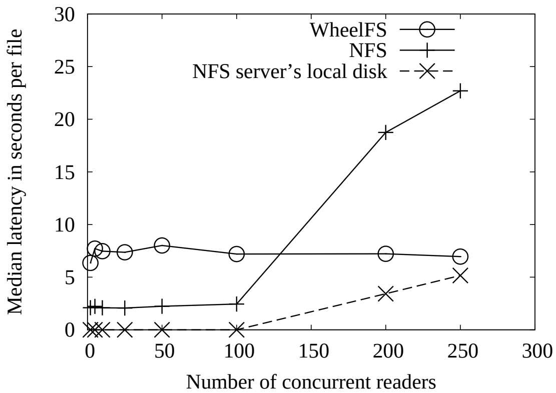

Figure 3: The median time for a set of PlanetLab clients to read a 1 MB file, as a function of the number of concurrently reading nodes. Also plots the median time for a set of local processes to read 1 MB files from the NFS server's local disk through `ext3`.

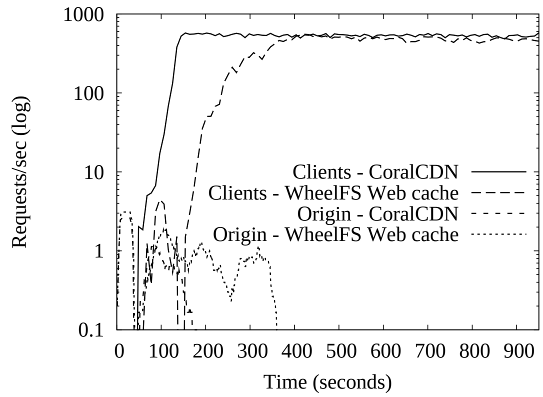

Figure 4: The aggregate client service rate and origin server load for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

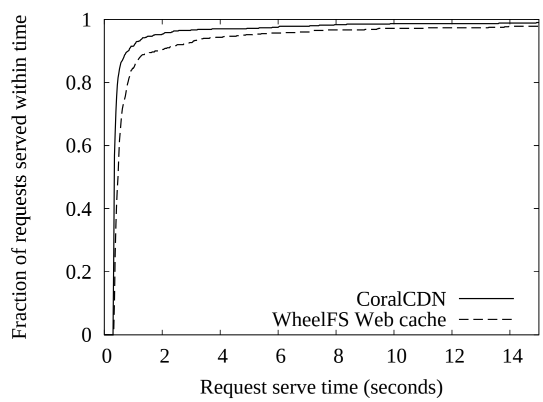

Figure 5: The CDF for the client request latencies of both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

system in sequence, and measures the read latency. The clients all do this at the same time.

For WheelFS, each client also acts as a server, and is the primary for one directory and all files within that directory. WheelFS clients do not read files for which they are the primary, and no file is ever read twice by the same node. The NFS server is a machine at MIT running Debian's nfs-kernel-server version 1.0.10-6 using the default configuration, with a 2.8 GHz CPU and a SCSI hard drive.

Figure 3 shows the median time to read a file as N varies. For WheelFS, a very small fraction of reads fail because not all pairs of PlanetLab nodes can communicate; these reads are not included in the graph. Each point on the graph is the median of the results of at least one hundred nodes (*e.g.*, a point showing the latency for five concurrent nodes represents the median reported by all nodes across twenty different trials).

Though the NFS server achieves lower latencies when there are few concurrent clients, its latency rises sharply as the number of clients grows. This rise occurs when there are enough clients, and thus files, that the files do not fit in the server's 1GB file cache. Figure 3 also shows results for N concurrent processes on the NFS server, accessing the `ext3` file system directly, showing a similar latency increase after 100 clients. WheelFS latencies are not affected by the number of concurrent clients, since WheelFS spreads files and thus the load across many servers.

8.3 Distributed Web Cache

Performance under normal conditions. These experiments compare the performance of CoralCDN and the WheelFS distributed Web cache (as described in Section 6, except with **.MaxTime=2000** to adapt to PlanetLab's characteristics). The main goal of the cache is to reduce load on target Web servers via caching, and secondarily to provide client browsers with reduced latency and increased availability.

These experiments use forty nodes from PlanetLab hosted at `.edu` domains, spread across the continental United States. A Web server, located at NYU behind an emulated slow link (shaped using Click [24] to be 400 Kbps and have a 100 ms delay), serves 100 unique 41KB Web pages. Each of the 40 nodes runs a Web proxy. For each proxy node there is another node less than 10 ms away that runs a simulated browser as a Web client. Each Web client requests a sequence of randomly selected pages from the NYU Web server. This experiment, inspired by one in the CoralCDN paper [19], models a flash crowd where a set of files on an under-provisioned server become popular very quickly.

Figures 4 and 5 show the results of these experiments. Figure 4 plots both the total rate at which the proxies send requests to the origin server and the total rate at which the proxies serve Web client requests (the y-axis is a log scale). WheelFS takes about twice as much time as Coral-

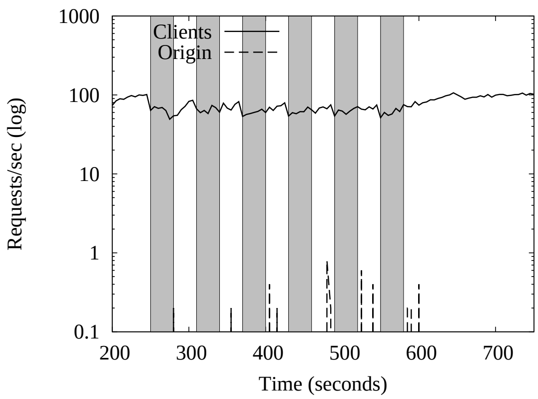

Figure 6: The WheelFS-based Web cache running on Emulab with failures, using the **.EventualConsistency** cue. Gray regions indicate the duration of a failure.

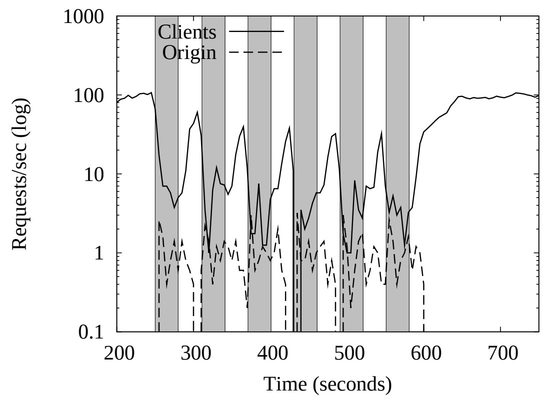

Figure 7: The WheelFS-based Web cache running on Emulab with failures, with close-to-open consistency. Gray regions indicate the duration of a failure.

CDN to reduce the origin load to zero; both reach similar sustained aggregate Web client service rates. Figure 5 plots the cumulative distribution function (CDF) of the request latencies seen by the Web clients. WheelFS has somewhat higher latencies than CoralCDN.

CoralCDN has higher performance because it incorporates many application-specific optimizations, whereas the WheelFS-based cache is built from more general-purpose components. For instance, a CoralCDN proxy pre-declares its intent to download a page, preventing other nodes from downloading the same page; Apache, running on WheelFS, has no such mechanism, so several nodes may download the same page before Apache caches the data in WheelFS. Similar optimizations could be implemented in Apache.

Performance under failures. Wide-area network problems that prevent WheelFS from contacting storage nodes should not translate into long delays; if a proxy cannot quickly fetch a cached page from WheelFS, it should ask the origin Web server. As discussed in Section 6, the cues **.EventualConsistency** and **.MaxTime=1000** yield this behavior, causing `open()` to either find a copy of the desired file or fail in one second. Apache fetches from the origin Web server if the `open()` fails.

To test how failures affect WheelFS application performance, we ran a distributed Web cache experiment on the Emulab topology in Section 8.1, where we could control the network's failure behavior. At each of the five sites there are three WheelFS Web proxies. Each site also has a Web client, which connects to the Web proxies at the same site using a 10 Mbps, 20 ms link, issuing five requests at a time. The origin Web server runs behind a 400 Kbps link, with 150 ms RTTs to the Web proxies.

Figures 6 and 7 compare failure performance of WheelFS with the above cues to failure performance of close-to-open consistency with 1-second timeouts (**.MaxTime=1000**). The y-axes of these graphs are log-scale. Each minute one wide-area link connecting an entire site to the rest of the network fails for thirty seconds and then revives. This failure period is not long enough to cause servers at the failed site to lose their slice locks. Web clients maintain connectivity to the proxies at their local site during failures. For comparison, Figure 8 shows WheelFS's performance on this topology when there are no failures.

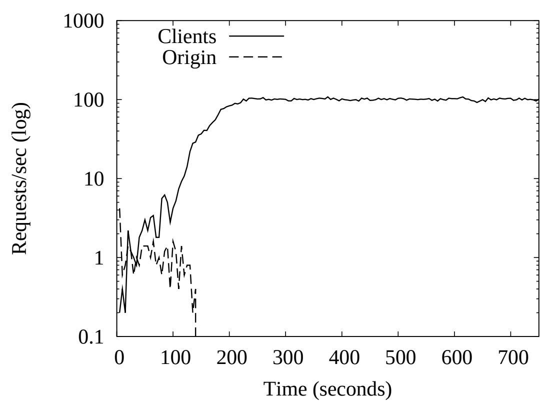

Figure 8: The aggregate client service rate and origin server load for the WheelFS-based Web cache, running on Emulab, without failures.

When a Web client requests a page from a proxy, the proxy must find two pieces of information in order to find a copy of the page (if any) in WheelFS: the object ID to which the page's file name resolves, and the file content for that object ID. The directory information and the file content can be on different WheelFS servers. For each kind of information, if the proxy's WheelFS client has cached the information and has a valid lease, the WheelFS

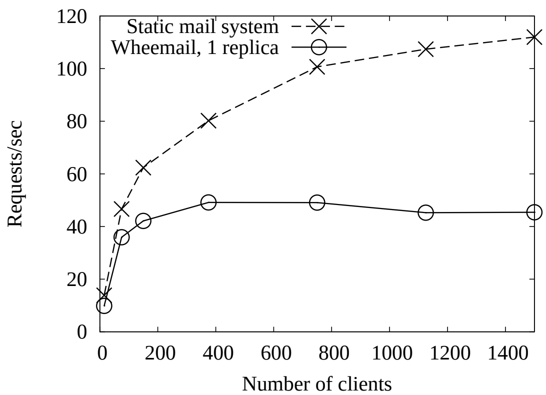

Figure 9: The throughput of Wheemail compared with the static system, on the Emulab testbed.

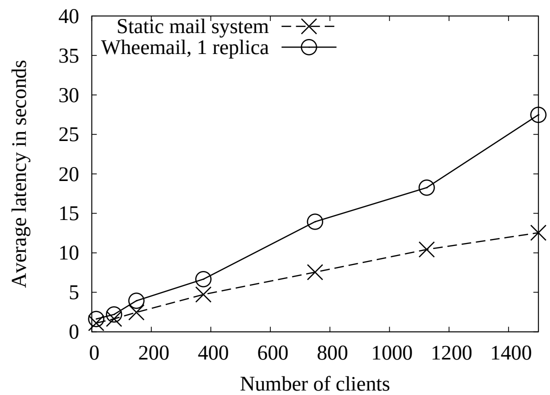

Figure 10: The average latencies of individual SMTP requests, for both Wheemail and the static system, on Emulab.

client need not contact a server. If the WheelFS client doesn't have information with a valid lease, and is using eventual consistency, it tries to fetch the information from the primary; if that fails (after a one-second timeout), the WheelFS client will try fetch from a backup; if that fails, the client will use locally cached information (if any) despite an expired lease; otherwise the `open()` fails and the proxy fetches the page from the origin server. If a WheelFS client using close-to-open consistency does not have cached data with a valid lease, it first tries to contact the primary; if that fails (after timeout), the proxy must fetch the page from the origin Web server.

Figure 6 shows the performance of the WheelFS Web cache with eventual consistency. The graph shows a period of time after the initial cache population. The gray regions indicate when a failure is present. Throughput falls as WheelFS clients encounter timeouts to servers at the failed site, though the service rate remains near 100 requests/sec. The small load spikes at the origin server after a failure reflect requests queued up in the network by the failed site while it is partitioned. Figure 7 shows that with close-to-open consistency, throughput falls significantly during failures, and hits to the origin server increase greatly. This shows that a cooperative Web cache, which does not require strong consistency, can use WheelFS's semantic cues to perform well under wide-area conditions.

8.4 Mail

The Wheemail system described in Section 6 has a number of valuable properties such as the ability to serve and accept a user's mail from any of multiple sites. This section explores the performance cost of those properties by comparing to a traditional mail system that lacks those properties.

IMAP and SMTP are stressful file system benchmarks. For example, an IMAP server reading a Maildir-formatted inbox and finding no new messages generates over 600 FUSE operations. These primarily consist of lookups on directory and file names, but also include more than 30 directory operations (creates/links/unlinks/renames), more than 30 small writes, and a few small reads. A single SMTP mail delivery generates over 60 FUSE operations, again consisting mostly of lookups.

In this experiment we use the Emulab network topology described in Section 8.1 with 5 sites. Each site has a 1 Mbps link to a wide-area network that connects all the sites. Each site has three server nodes that each run a WheelFS server, a WheelFS client, an SMTP server, and an IMAP server. Each site also has three client nodes, each of which runs multiple load-generation threads. A load-generation thread produces a sequence of SMTP and IMAP requests as fast as it can. 90% of requests are SMTP and 10% are IMAP. User mailbox directories are randomly and evenly distributed across sites. The load-generation threads pick users and message sizes with probabilities from distributions derived from SMTP and IMAP logs of servers at NYU; there are 47699 users, and the average message size is 6.9 KB. We measure throughput in requests/second, with an increasing number of concurrent client threads.

When measuring WheelFS, a load-generating thread at a given site only generates requests from users whose mail is stored at that site (the user's "home" site), and connects only to IMAP and SMTP servers at the local site. Thus an IMAP request can be handled entirely within a home site, and does not generate any wide-area traffic (during this experiment, each node has cached directory lookup information for the mailboxes of all users at its site). A load-generating thread generates mail to random users, connecting to a SMTP server at the same site; that server writes the messages to the user's directory in WheelFS, which is likely to reside at a different site. In this experiment, user mailbox directories are not replicated.

We compare against a "static" mail system in which users are partitioned over the 15 server nodes, with the

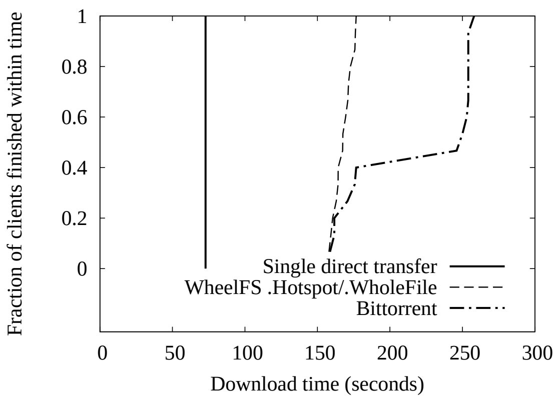

Figure 11: CDF of client download times of a 50 MB file using BitTorrent and WheelFS with the **.Hotspot** and **.WholeFile** cues, running on Emulab. Also shown is the time for a single client to download 50 MB directly using `ttcp`.

SMTP and IMAP servers on each server node storing mail on a local disk file system. The load-generator threads at each site only generate IMAP requests for users at the same site, so IMAP traffic never crosses the wide area network. When sending mail, a load-generating client picks a random recipient, looks up that user's home server, and makes an SMTP connection to that server, often across the wide-area network.

Figure 9 shows the aggregate number of requests served by the entire system per second. The static system can sustain 112 requests per second. Each site's 1 Mbps wide-area link is the bottleneck: since 90% of the requests are SMTP (message with an average size 6.85 KB), and 80% of those go over the wide area, the system as a whole is sending 4.3 Mbps across a total link capacity of 5 Mbps, with the remaining wide-area bandwidth being used by the SMTP and TCP protocols.

Wheemail achieves up to 50 requests per second, 45% of the static system's performance. Again the 1 Mbps WAN links are the bottleneck: for each SMTP request, WheelFS must send 11 wide-area RPCs to the target user's mailbox site, adding an overhead of about 40% to the size of the mail message, in addition to the continuous background traffic generated by the maintenance process, slice lock renewal, Vivaldi coordinate measurement, and occasional lease invalidations.

Figure 10 shows the average latencies of individual SMTP requests for Wheemail and the static system, as the number of clients varies. Wheemail's latencies are higher than those of the static system by nearly 60%, attributable to traffic overhead generated by WheelFS.

Though the static system outperforms Wheemail for this benchmark, Wheemail provides many desirable properties that the static system lacks. Wheemail transparently redirects a receiver's mail to its home site, regardless of where the SMTP connection occurred; additional storage can be added to the system without major manual reconfiguration; and Wheemail can be configured to offer tolerance to site failures, all without any special logic having to be built into the mail system itself.

Application	LoC	Reuses
CDN	1	Apache+mod_disk_cache
Mail service	4	Sendmail+Procmail+Dovecot
File distribution	N/A	Built-in to WheelFS
All-Pairs-Pings	13	N/A

Table 3: Number of lines of changes to adapt applications to use WheelFS.

8.5 File distribution

Our file distribution experiments use a WheelFS network consisting of 15 nodes, spread over five LAN clusters connected by the emulated wide-area network described in Section 8.1. Nodes attempt to read a 50 MB file simultaneously (initially located at an originating, 16^{th} WheelFS node that is in its own cluster) using the **.Hotspot** and **.WholeFile** cues. For comparison, we also fetch the file using BitTorrent [14] (the Fedora Core distribution of version 4.4.0-5). We configured BitTorrent to allow unlimited uploads and to use 64 KB blocks like WheelFS (in this test, BitTorrent performs strictly worse with its usual default of 256 KB blocks).

Figure 11 shows the CDF of the download times, under WheelFS and BitTorrent, as well as the time for a single direct transfer of 50 MB between two wide-area nodes (73 seconds). WheelFS's median download time is 168 seconds, showing that WheelFS's implementation of cooperative reading is better than BitTorrent's: BitTorrent clients have a median download time of 249 seconds. The improvement is due to WheelFS clients fetching from nearby nodes according to Vivaldi coordinates; BitTorrent does not use a locality mechanism. Of course, both solutions offer far better download times than 15 simultaneous direct transfers from a single node, which in this setup has a median download time of 892 seconds.

8.6 Implementation ease

Table 3 shows the number of new or modified lines of code (LoC) we had to write for each application (excluding WheelFS itself). Table 3 demonstrates that developers can benefit from a POSIX file system interface and cues to build wide-area applications with ease.

9 Related Work

There is a humbling amount of past work on distributed file systems, wide-area storage in general and the tradeoffs of availability and consistency. PRACTI [8] is a recently-proposed framework for building storage systems with arbitrary consistency guarantees (as in TACT [43]). Like PRACTI, WheelFS maintains flexibility by separating policies from mechanisms, but it has a different goal. While PRACTI and its recent extension PADS [9] are designed to simplify the development of new storage or file systems, WheelFS itself is a flexible file system designed to simplify the construction of distributed applications. As a result, WheelFS's cues are motivated by the specific needs of applications (such as the **.Site** cue) while PRACTI's primitives aim at covering the entire spectrum of design tradeoffs (*e.g.*, strong consistency for operations spanning multiple data objects, which WheelFS does not support).

Most distributed file systems are designed to support a workload generated by desktop users (*e.g.*, NFS [33], AFS [34], Farsite [2], xFS [5], Frangipani [12], Ivy [27]). They usually provide a consistent view of data, while sometimes allowing for disconnected operation (*e.g.*, Coda [35] and BlueFS [28]). Cluster file systems such as GFS [22] and Ceph [41] have demonstrated that a distributed file system can dramatically simplify the construction of distributed applications within a large cluster with good performance. Extending the success of cluster file systems to the wide-area environment continues to be difficult due to the tradeoffs necessary to combat wide-area network challenges. Similarly, Sinfonia [3] offers highly-scalable cluster storage for infrastructure applications, and allows some degree of inter-object consistency via lightweight transactions. However, it targets storage at the level of individual pieces of data, rather than files and directories like WheelFS, and uses protocols like two-phase commit that are costly in the wide area. Shark [6] shares with WheelFS the goal of allowing client-to-client data sharing, though its use of a centralized server limits its scalability for applications in which nodes often operate on independent data.

Successful wide-area storage systems generally exploit application-specific knowledge to make decisions about tradeoffs in the wide-area environment. As a result, many wide-area applications include their own storage layers [4, 14, 19, 31] or adapt an existing system [29, 40]. Unfortunately, most existing storage systems, even more general ones like OceanStore/Pond [30] or S3 [1], are only suitable for a limited range of applications and still require a large amount of code to use. DHTs are a popular form of general wide-area storage, but, while DHTs all offer a similar interface, they differ widely in implementation. For example, UsenetDHT [36] and CoralCDN [19] both use a DHT, but their DHTs differ in many details and are not interchangeable.

Some wide-area storage systems offer configuration options in order to make them suitable for a larger range of applications. Amazon's Dynamo [16] works across multiple data centers and provides developers with two knobs: the number of replicas to read or to write, in order to control durability, availability and consistency tradeoffs. By contrast, WheelFS's cues are at a higher level (*e.g.*, eventual consistency versus close-to-open consistency). Total Recall [10] offers a per-object flexible storage API and uses a primary/backup architecture like WheelFS, but assumes no network partitions, focuses mostly on availability controls, and targets a more dynamic environment. Bayou [39] and Pangaea [32] provide eventual consistency by default while the latter also allows the use of a "red button" to wait for the acknowledgment of updates from all replicas explicitly. Like Pangaea and Dynamo, WheelFS provides flexible consistency tradeoffs. Additionally, WheelFS also provides controls in other categories (such as data placement, large reads) to suit the needs of a variety of applications.

10 Conclusion

Applications that distribute data across multiple sites have varied consistency, durability, and availability needs. A shared storage system able to meet this diverse set of needs would ideally provide applications a flexible and practical interface, and handle applications' storage needs without sacrificing much performance when compared to a specialized solution. This paper describes WheelFS, a wide-area storage system with a traditional POSIX interface augmented by cues that allow distributed applications to control consistency and fault-tolerance tradeoffs.

WheelFS offers a small set of cues in four categories (placement, durability, consistency, and large reads), which we have found to work well for many common distributed workloads. We have used a WheelFS prototype as a building block in a variety of distributed applications, and evaluation results show that it meets the needs of these applications while permitting significant code reuse of their existing, non-distributed counterparts. We hope to make an implementation of WheelFS available to developers in the near future.

Acknowledgments

This research was supported by NSF grant CNS-0720644 and by Microsoft Research Asia and Tsinghua University. We thank the many people who have improved this work through discussion, testbed support, reviews, and skepticism: our shepherd Jeff Dean, the anonymous reviewers, the members of the PDOS research group at MIT CSAIL, Sapan Bhatia, Russ Cox, Frank Dabek, Marc Fiuczynski, Michael Freedman, Jeremy Kepner, Jay Lepreau, Jinyuan Li, David Mazières, Michael Puskar and NYU ITS, Emil Sit, Michael Walfish, Daniel Hokka Zakrisson, and Nickolai Zeldovich.

References

[1] Amazon Simple Storage System. `http://aws.amazon.com/s3/`.

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th OSDI* (Dec. 2002).

[3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st SOSP* (Oct. 2007).

[4] ALLCOCK, W., BRESNAHAN, J., KETTIMUTHU, R., LINK, M., DUMITRESCU, C., RAICU, I., AND FOSTER, I. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 Super Computing* (Nov. 2005).

[5] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *Proceedings of the 15th SOSP* (Dec. 1995).

[6] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIÈRES, D. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd NSDI* (May 2005).

[7] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating systems support for planetary-scale network services. In *Proceedings of the 1st NSDI* (Mar. 2004).

[8] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI replication. In *Proceedings of the 3rd NSDI* (2006).

[9] BELARAMANI, N., ZHENG, J., NAYATE, A., SOULÉ, R., DAHLIN, M., AND GRIMM, R. PADS: A policy architecture for building distributed storage systems. In *Proceedings of the 6th NSDI* (Apr. 2009).

[10] BHAGWAN, R., TATI, K., CHENG, Y.-C., SAVAGE, S., AND VOELKER, G. M. Total Recall: System support for automated availability management. In *Proceedings of the 1st NSDI* (Mar. 2004).

[11] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th OSDI* (Nov. 2006).

[12] C. THEKKATH, T. MANN, E. L. Frangipani: A scalable distributed file system. In *Proceedings of the 16th SOSP*.

[13] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th OSDI* (Nov. 2006).

[14] COHEN, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[15] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. A decentralized network coordinate system. In *Proceedings of the 2004 SIGCOMM* (2004).

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st SOSP* (Oct. 2007).

[17] Dovecot IMAP server. `http://www.dovecot.org/`.

[18] FEKETE, A., GUPTA, D., LUCHANGCO, V., LYNCH, N., AND SCHVARTSMAN, A. Eventually-serializable data services. *Theoretical Computer Science* (June 1999).

[19] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with Coral. In *Proceedings of the 1st NSDI* (Mar. 2004).

[20] FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIÈRES, D. OASIS: Anycast for any service. In *Proceedings of the 3rd NSDI* (May 2006).

[21] Filesystem in user space. `http://fuse.sourceforge.net/`.

[22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th SOSP* (Dec. 2003).

[23] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. In *ACM SIGACT News* (June 2002), vol. 33.

[24] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Trans. on Computer Systems* (Aug. 2000).

[25] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[26] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure Untrusted data Repository (SUNDR). In *Proceedings of the 6th OSDI* (Dec. 2004).

[27] MUTHITACHAROEN, A., MORRIS, R., GIL, T., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th OSDI* (2002).

[28] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th OSDI* (Dec. 2004).

[29] PARK, K., AND PAI, V. S. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd NSDI* (May 2006).

[30] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: The OceanStore prototype. In *Proceedings of the 2nd FAST* (Mar. 2003).

[31] SAITO, Y., BERSHAD, B., AND LEVY, H. Manageability, availability and performance in Porcupine: A highly scalable internet mail service. *ACM Transactions of Computer Systems* (2000).

[32] SAITO, Y., KARAMONOLIS, C., KARLSSON, M., AND MAHALINGAM, M. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th OSDI* (2002).

[33] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX* (June 1985).

[34] SATYANARAYANAN, M., HOWARD, J., NICHOLS, D., SIDEBOTHAM, R., SPECTOR, A., AND WEST, M. The ITC distributed file system: Principles and design. In *Proceedings of the 10th SOSP* (1985).

[35] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., OKASAKI, M., SIEGEL, E., AND STEERE, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Comp. 4*, 39 (Apr 1990), 447–459.

[36] SIT, E., MORRIS, R., AND KAASHOEK, M. F. UsenetDHT: A low-overhead design for Usenet. In *Usenix NSDI* (2008).

[37] STRIBLING, J. PlanetLab All-Pairs-Pings. `http://pdos.csail.mit.edu/~strib/pl_app/`.

[38] STRIBLING, J., SIT, E., KAASHOEK, M. F., LI, J., AND MORRIS, R. Don't give up on distributed file systems. In *Proceedings of the 6th IPTPS* (2007).

[39] TERRY, D., THEIMER, M., PETERSEN, K., DEMERS, A., SPREITZER, M., AND HAUSER, C. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th SOSP* (1995).

[40] VON BEHREN, J. R., CZERWINSKI, S., JOSEPH, A. D., BREWER, E. A., AND KUBIATOWICZ, J. Ninjamail: the design of a high-performance clustered, distributed e-mail system. In *Proceedings of the ICPP '00* (2000).

[41] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th OSDI* (Nov. 2006).

[42] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th OSDI* (Dec. 2002).

[43] YU, H., AND VAHDAT, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS 20*, 3 (Aug. 2002), 239–282.