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

Internet-based services that span multiple sites will likely benefit from a wide-area distributed storage system to help the sites interact and gain fault tolerance. This paper presents such a storage system, WheelFS, in the form of a distributed file system with a familiar POSIX interface. WheelFS operates as a cooperating set of servers spread over a wide-area network, and thus must cope with server and network failures as well as high latency and restricted bandwidth. A primary focus of WheelFS's design is resolving the tension between the need for sites to see each other's updates and the need for sites to be able to operate independently in the face of failures. WheelFS allows applications to chose how to cope with wide-area network behavior using *semantic cues*, which allow application control over consistency, failure handling, and file and replica placement.

An implementation of WheelFS is deployed on PlanetLab, Emulab, and on a 20-node private testbed. An evaluation with three applications (a CDN, an email service and large file distribution) shows that WheelFS provides the failure behavior that the applications need, is easy to use as part of a distributed application, and provides competitive performance.

1 Introduction

There is a growing set of Internet-based services that are too big to run conveniently on a single data center; examples include web sites for e-mail, video and image hosting, and social networking. Splitting such services over multiple data centers offers potential for improvements beyond just partitioning the load: cooperation among sites, fault tolerance via geographically diverse replication, and reduction of bandwidth cost and latency by locating data near likely consumers. Once an organization has distributed data centers available, it is desirable to be able to re-use the centers for new services easily. Single data centers often have an analogous set of problems when clusters of servers read and write data and need to see each other's changes; general-purpose shared storage systems have addressed these needs successfully [1–4, 13, 20, 32]. Could a shared storage system help distribute services over the wide area? This paper explores the feasibility of using a file system designed specifically to support wide-area distributed services.

A wide-area storage system faces a tension between sharing and site independence. The system must support sharing: if one site stores some data, other sites may need to be able to find and read that data. On the other hand, sharing can be dangerous; one site should be able to make progress even if other sites are unavailable, since a primary goal of multi-site operation is fault tolerance. The storage system's consistency model affects the sharing/independence tradeoff: the stronger forms of consistency usually require serialization by a designated server, whose unreliability may force delays at other sites [21]. 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: the application can control file system behavior such as consistency, failure handling, and replica placement. The controls allow application-specific tradeoffs between performance and consistency, in the style of PRACTI [10] and Padre [11], but in the context of a file system with a POSIX interface.

A central challenge in the design of WheelFS is deciding what behavior the file system should provide by default, and how to let developers specify non-default behavior in a simple way. WheelFS resolves these challenges as follows. By default, WheelFS provides close-to-open consistency and implements a *write-locally* data placement policy so that application output can be written at disk data rates. Applications can adjust this default behavior with *semantic cues*, which specify placement and consistency policies, among other categories. Our hypothesis is that the number of sensible policies for placement, consistency, etc. is small and can be expressed with a small number of cues. WheelFS allows the cues to be expressed in the pathname, and thus no changes are necessary to the standard POSIX interface. The advantage of this solution is that developers can re-use existing code with only small modifications, often just to configurable file names.

Much of the justification for WheelFS providing a file system interface is pragmatic: developers will value the convenience and familiarity of a file system interface. WheelFS may not provide the semantics and performance that some applications require; such applications may use more sophisticated storage systems, perhaps after a proto-

typing stage on WheelFS. At the moment, however, there are few options for wide-area storage; one possibility is DHTs [16, 28, 29, 37]. We hope that other storage systems can take advantage of our experience in the choice of cues to offer.

A prototype of WheelFS runs on FreeBSD, Linux, and MacOS. The client implementation runs as a user-level file system, using FUSE [19] to allow applications to use standard system calls. We have deployed WheelFS on PlanetLab, on an emulated wide-area network on Emulab, and on a set of 20 dedicated nodes at three Internet sites.

We demonstrate WheelFS's usefulness with several applications: a caching CDN, a distributed email service, and a few smaller applications, including a PlanetLab measurement application. All were easy to build by reusing existing software components, with WheelFS for storage instead of a local file system. The extent to which we could re-use existing non-distributed software in these applications came as a surprise [39], but it illustrates the value of a file system interface. Performance measurements show that these applications behave well with WheelFS, even when there are failures, and that performance is competitive with other approaches. Cues play a large role in allowing the applications to achieve good failure behavior and good performance.

The main contributions of this paper are as follows: a new file system that assists in 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. Section 2 gives the general system model for WheelFS. Section 3 summarizes the challenges that wide-area storage applications face and our general approach to handling them. Section 4 presents WheelFS's core design, while Section 5 presents the cues that allow applications to customize the behavior of WheelFS. Section 6 describes some example applications, while Section 7 outlines the implementation of WheelFS. Section 8 measures the performance of the WheelFS applications. Section 9 discusses related work, and Section 10 concludes.

2 System Model

We consider distributed applications that run on a collection of sites distributed over the wide-area Internet. All nodes running WheelFS are either managed by a single administrative entity or multiple administrative entities that explicitly cooperate with each other. In such a deployment environment, nodes are trustworthy, as opposed to a peer-to-peer environment where nodes could be intentionally malicious. As such, the security requirement of WheelFS is to provide proper access control instead of guarding against data corruption and loss due to malicious servers [9, 24]. Because nodes in our deployment environment are well-managed, we also expect them to have high overall uptimes despite occasional failures. Many existing distributed infrastructures fit our deployment scenario such as wide-area testbeds (*e.g.*, PlanetLab and RON), a collection of data centers spread across the globe (*e.g.*, Amazon's EC2), and federated resources such as Grids.

3 Challenges and Approach

There are many benefits to a wide-area distributed storage system. It can act as a repository of data that multiple sites read. It can serve as a rendezvous point or communication channel among service components running at different sites, as one site writes data that another site later needs to read. It can increase fault tolerance by replicating data at geographically-separated sites. Finally, it can help performance by placing data replicas near the clients most likely to use the data. This section presents a case study of some of the difficulties in providing such wide-area storage, and outlines WheelFS's approach.

3.1 Wide-area challenges

Building a wide-area application requires significant programming effort. What are the sources of this complexity? We use the popular CoralCDN [18] application as a case study. CoralCDN consists of hundreds of Web proxies distributed across the Internet that serve Web pages from a shared cache. The system's goal is to reduce the origin Web server's load as well as browser fetch latency [18, 42, 45]. When a browser sends a request to one of the proxies, the proxy looks for the requested Web page in the shared cache; if it is there and has not expired, the proxy serves the request from the cache. Otherwise the proxy fetches the page from the origin server, stores it in the shared cache, and serves it to the browser.

Roughly half of CoralCDN's 34,000 lines of code implements its distributed sloppy hash table, a storage module that allows nodes to find cached Web pages among all proxies. Most of the remaining code implements a custom web proxy to interface with CoralCDN's distributed hash table. What wide-area challenges does the CoralCDN storage module face?

- *High latency.* Delays to servers that are distant or on heavily-loaded links may be high. CoralCDN reduces delay by reading the cached copy that is nearest in the network. In order to find nearby nodes, CoralCDN groups nodes into clusters according to their pair-wise latencies.

- *Low bandwidth.* Since wide-area network capacity is limited, applications should avoid moving large

amounts of data long distances. Each CoralCDN proxy stores newly-fetched Web pages on its local disk, and moves data to remote proxies only when clients need it.

- *Transient failures.* The wide-area network is prone to transient routing, link, and host failures. Applications can cope by keeping multiple copies of their data. For read/write data this risks inconsistency among the copies, and different applications have different preferences in the tradeoff between availability and consistency. CoralCDN does not require the cache to yield the latest copy of a page because it can use saved HTTP headers to check freshness.

3.2 Our approach

WheelFS faces the challenges outlined above while pursuing its main goals. These goals are: 1) to provide distributed storage to multi-site applications, 2) to allow applications to choose a tradeoff between consistency and failure-independence, and 3) to provide a programmer-friendly traditional POSIX file system interface. The hope is that a file system will simplify programming because programmers are used to its API; furthermore, it may allow for more code re-use, enabling application writers to focus on the core of their application while they adopt existing components designed for local file systems (Web proxy, IMAP and SMTP servers, etc.).

WheelFS provides a location-independent hierarchy of directories and files with a POSIX file system interface. WheelFS provides knobs for an application to alter WheelFS's data placement behavior and its consistency and availability tradeoff. These knobs take the form of *semantic cues* embedded in pathnames. For example, reading a file named **/wfs/cache/.EventualConsistency/**foo allows WheelFS to produce out-of-date data if that would avoid a long delay while trying to talk to an unresponsive server. Use of WheelFS's cues often requires only small changes to software that already uses files for storage.

Using WheelFS, we can build a distributed Web cache similar to CoralCDN quickly by re-using existing software. The idea is to run an existing single-server caching proxy (such as Apache or Squid [36]) on each of a collection of hosts, changing the proxy configurations to store cached pages in a common directory on WheelFS instead of on the local disk. When the proxy fetches a page from the origin server, it stores the content in a file whose name is derived from the URL. The existing proxy code handles HTTP, while WheelFS takes care of discovering whether the distributed file system already contains a copy of the desired page. The cache directory includes cues to express two important wide-area optimizations for a CDN (cues are discussed in more detail in Section 5). **Site=local** instructs WheelFS to store all newly-created files on a nearby WheelFS server (though this is the default behavior). **EventualConsistency** enables the WheelFS client to read from a locally-cached copy of a file without checking with the WheelFS server system to see whether the file has been subsequently modified. This design simplifies the construction of the CDN by allowing re-use of existing proxy software, though it does not provide all of the optimizations in CoralCDN. Section 8.2 demonstrates that this design achieves the main properties that one expects from a CDN.

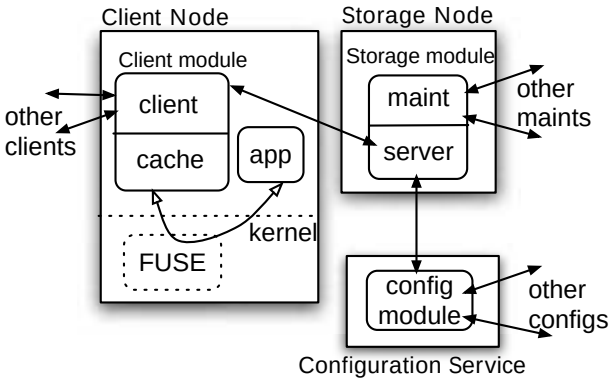

Figure 1: Placement and interaction of WheelFS components.

4 WheelFS Design

A key design challenge is to provide reasonable defaults that reduce application complexity while identifying a small set of cues that can allow applications to tailor storage behavior for high performance on the wide area. This section explains the default behavior—called the "core design"—and Section 5 describes how cues modify the defaults so that applications can tailor WheelFS to their needs. By default, WheelFS provides close-to-open semantics so that if an application works correctly on a POSIX file system, it will also work correctly on WheelFS.

4.1 Overview

A WheelFS deployment (see Figure 1) consists of client nodes and storage nodes; a single host often plays both roles. The WheelFS client software uses FUSE [19] to present the distributed file system to local applications, typically in `/wfs`. All client nodes in a given deployment present the same file system tree in `/wfs`. Thus, when an application opens a file, independent of on which client the application runs, the `open` call returns a file descriptor for the same file. The WheelFS client module communicates with WheelFS storage modules in order to look up file names, create files, get directory listing, and read and write files. Each client keeps a local cache of file and directory contents.

Each file and directory object has a unique ID. At any given time, every ID has a single "primary" stor-

age node that is responsible for maintaining the latest contents of that object. WheelFS replicates objects using primary/backup replication. A configuration service is responsible for ensuring agreement (using Paxos [23]) on the assignment of IDs to primary and backup storage nodes. A client contacts an object's primary storage node when it needs to use the object; for example, looking up a path name with multiple components typically involves contacting a different storage node for each component, which yields the ID of the next component in the path.

The core design of WheelFS addresses the following questions:

- How does WheelFS partition the storage responsibility for data objects among participating storage nodes? (Section 4.2)
- How does WheelFS provide close-to-open consistency in the face of concurrent file access and failures? (Section 4.4)
- How does WheelFS provide reasonable default read and write performance for typical wide-area applications? (Section 4.5)
- How does WheelFS authenticate users and perform access control? (Section 4.6)

4.2 Data storage assignment

WheelFS storage nodes store file and directory objects. Each object is internally named using a unique 64-bit 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 spreads the load of storing and serving files and directories across the participating storage nodes.

WheelFS partitions the object ID space into 2^{16} slices using the first 16 bits of the object ID. The configuration service decides, for each slice, which storage nodes are responsible for storing the objects in that slice. Each slice has an associated replication level r, and the configuration service ensures that there is one "primary" storage node and $r - 1$ "backup" storage nodes for the slice. The number of ID slices is larger than the number of storage nodes to help balance load. The configuration service reacts to storage node failures and additions by creating new slice assignments. The configuration service's use of Paxos ensures that at any time there is at most one slice assignment.

A WheelFS client must know the slice assignment in order to know which storage node to contact for each object ID. To reduce the load on the configuration service, the clients gossip with each other to learn the contents of new assignments.

When an application creates a new file or directory, the WheelFS client uses its knowledge of the slice assignment to assign the new object an ID that makes the storage node on the same host (if there is one) responsible for the new object. Storing data locally allows applications to achieve high write throughput, since they can write at the rate of the local disk instead of the rate of the wide-area network.

The WheelFS `maint` component of the storage module runs every ten minutes to ensure that each object is stored on the servers in its slice assignment. `maint` transfers those local objects that the storage node is not in charge of to their responsible storage nodes. The primary storage node in the replica set also synchronizes with its backups to ensure that the desired number of replicas is stored, similar to the PassingTone algorithm [35].

4.3 Primary/backup replication

WheelFS uses primary/backup replication to manage replicated objects. The slice assignment designates, for each ID slice, a primary storage node and a number of backup storage nodes. When a client needs to read or modify an object, it communicates only with the primary. The primary forwards updates to the backups. For each object, the primary executes operations one at a time.

If the configuration service decides a storage node is not reachable, it will create a new slice assignment that omits that server. For slices for which the dead node was the primary, the configuration service selects a new primary, usually by promoting a backup. For slices for which the dead node was a backup, the configuration service selects a new backup. As a result, storage nodes may be responsible for slices for which they do not store the corresponding objects; `maint` copies the required objects.

There must be only one primary server for a slice, to avoid inconsistency. If the configuration service decides that a storage node is no longer the primary, that node must stop serving requests for that slice. This can be difficult if the underlying problem is a network partition: the configuration service might think the node is dead when it is actually alive, and the node did not hear about the new slice assignment. WheelFS handles this with *slice leases*—the primary for a slice must check with the configuration service every ten minutes to ensure it is still the primary. The configuration service does not announce a new slice assignment until all relevant storage nodes have acknowledged slice lease revocation or their slice leases have expired. The slice lease time is a compromise: short lease times lead to fast reconfiguration, while long lease times allow storage nodes to operate despite the temporary unreachability of the configuration service.

4.4 Close-to-open consistency

WheelFS provides close-to-open consistency: if one application writes a file and waits for `close()` to return, and then a second application `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. The primary server grants *object leases* for cached meta-data and directory contents; the server invalidates object leases for a given piece of meta-data before it executes any update requests.

Client nodes send file writes to WheelFS storage nodes without waiting for the replies, to improve performance. When an application calls `close()`, the client waits for replies to all outstanding writes to the file. In order to ensure that the crash of a primary will not undo writes, the primary waits for the backups to acknowledge the update before it replies to the client. The WheelFS storage nodes 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 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 blocks from the primary storage node.

WheelFS provides similar consistency for directory operations: after the return of an application system call that modifies a directory (creates or deletes a file or subdirectory), applications on other clients are guaranteed to see the modification. WheelFS clients implement this consistency by always 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.

The downside to close-to-open consistency is that if the primary for an object is not available, all read and write operations will block until the configuration service chooses a new slice allocation with a new primary for the object. The configuration service must wait until the primary's slice lease expires, up to ten minutes. The cues described in Section 5 allow the application to avoid these delays at the expense of consistency.

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.

4.5 Write locally

The default data placement policy in WheelFS is to *write locally*, *i.e.*, store the contents of a newly created file on the local node's disk if possible. This policy requires that application nodes run both a WheelFS storage node as well as a client node. The write-local policy allows large-file writes at the speed of the local disk.

Modifying an existing file is not always fast, because the file's primary storage node might be far away. Applications desiring fast writes should store output in unique new files, so that the local client node will be able to create a new object ID for which the local storage node will be 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.

The write-local policy may not balance the storage load across storage nodes, depending on whether each node creates and writes files at roughly the same rate. Applications can use cues to control where files are stored.

4.6 Security

WheelFS is designed to be easily adaptable to a variety of deployment scenarios. WheelFS maintains and enforces user-based file permissions, but the definition of what constitutes a user and how WheelFS verifies user identities can vary according to the needs of each deployment. In its most basic mode of operation, WheelFS provides minimal security, using unencrypted connections and trusting client nodes to accurately identify users by local user name or a client-based mapping. It takes no measures to prevent a malicious client from impersonating a privileged user. This provides a low-overhead model for local testing or deployment on networks with strong isolation from the public Internet. Other environments, however, require stronger security models.

In the security model currently in development for PlanetLab, where nodes have unrestricted access to the Internet, an ordinary PlanetLab user can run WheelFS storage nodes in their own slices on multiple machines. This user is the trusted administrator to the WheelFS instance, but needs no special permissions on PlanetLab itself. The administrator gives each machine an individual private SSH key and the public keys for the other storage nodes. With these keys, storage nodes can create secure SSH connections to each other and prevent any malicious servers from joining the system.

To add a (possibly untrusted) user to the system, the administrator creates a file in WheelFS with one or more SSH public keys for that user, and gives the user public keys for one or more of the storage nodes. Since PlanetLab already uses SSH keys for authentication, these same keys may conveniently be re-used to connect to WheelFS, potentially forwarded by an SSH agent. For unattended PlanetLab applications, the user will need to put a private

key in his or her individual slice for the WheelFS client module to access. The user then runs the WheelFS client in their own PlanetLab slice. With the storage nodes' public keys and access to user's private key, the client can create secure SSH connections to the various storage nodes, ensuring that stored data is not accessible to outside observers and allowing users to prove their identity to the server. Client-to-client data distribution can similarly use the user public keys available in WheelFS to create secure connections.

5 Semantic cues

To achieve good performance over the wide area, applications must be able to control where data is stored as well as the desired tradeoff between consistency and availability. WheelFS allows applications to exert control using *semantic cues*. This section describes how applications specify cues and the set of cues that WheelFS provides.

5.1 Specifying cues

WheelFS allows applications to specify semantic cues in pathnames; for example, /wfs/**.EventualConsistency**/data refers to /wfs/data with the cue **.EventualConsistency**. This approach allows applications to store data in the wide area with much less specialized code than, for example, adding a new system call to communicate cues.

A pathname might contain several cues. WheelFS uses the following rules to combine them: (1) a cue applies to all files and directories in the pathname below the cue; and (2) cues that are specified later in a pathname may override cues in the same category appearing earlier. Table 1 lists the cues and the categories into which they are grouped.

The main advantage of embedding cues in pathnames is that it keeps the POSIX interface unchanged. This choice allows developers to program with an interface with which they are familiar and to re-use software easily. For example, Section 6 shows how to build a caching CDN based partially on unmodified Apache.

A disadvantage of cues is that they may break software that parse pathnames and incorrectly assume that a cue is a directory. We have not encountered examples of this problem.

5.2 Eventual Consistency

Applications can specify the **.EventualConsistency** cue to relax WheelFS's default close-to-open consistency. WheelFS's eventual consistency semantics are intended to allow a client to proceed despite unreachability of the primary storage node, and in some cases the backups as well. For reads and pathname lookups, eventual consistency 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, eventual consistency allows a client to write to a backup if the primary is not available. A consequence of eventual consistency is that clients may not see each other's updates if they cannot all reliably contact the primary. Many applications such as CDNs or email systems can tolerate eventual consistency without significantly compromising their users' experience, and in return can decrease delays and minimize service unavailability when a primary or its network link are unreliable.

When reading files or using directory contents with eventual consistency, a client may have a choice between the contents of its cache, replies to queries to one or more backup servers, and a reply from the primary. A client will use 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=X** cue (in units of milliseconds). If **.MaxTime** is used without eventual consistency, the WheelFS client will yield an error if it cannot contact the primary after the indicated time.

The `maint` component periodically reconciles a primary and its backups so that they eventually contain the same data for each file and directory. `maint` may need to resolve conflicting versions of objects. For a 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, `maint` chooses the one with the smallest ID and renames the other files. For a file, `maint` chooses arbitrarily among the replicas that have the highest version number; this may cause writes to be lost.

WheelFS's eventual consistency ensures only that all replicas of an object will eventually have identical images, which is weaker than in systems like Bayou [40]. For example, `maint` may cause a recently-deleted file to reappear. On the plus side, WheelFS's eventual consistency can be implemented with little change to the core design and doesn't require applications to provide resolvers for conflicting writes.

5.3 Durability

WheelFS allows applications to express durability preferences with two cues: **.RepLevel=N** and **.SyncLevel=N**. These cues can only be specified when an object is first created, their values are stored in the object's meta-data, and they apply to all operations on the object.

The **.RepLevel=N** cue causes the primary to store the object on $N - 1$ backups instead of on the default two backups. The maximum replication level is four.

The **.SyncLevel=N** cue causes the primary to wait only for the acknowledgment of successful writes to N backup servers before acknowledging the client's request, reducing durability but also reducing delays if some servers are slow or unreachable. If this cue is specified for a file and the `Client` module contacts the primary on a `close` for

Cue Category	Cue Name	Meaning (and Tradeoffs)
Consistency	**.EventualConsistency**	Relax close-to-open consistency, thus trading off strong semantics for maximum availability in the face of failures. Applications might see different contents under the same file and directory name and the re-appearance of newly-deleted files.
	.MaxTime=N	Limit a file system operation to take no more than N ms in trying to contact the primary.
Placement	**.Site=X**	Store data on a server at the site named X, thus trading off write performance for faster reads if applications know that clients in site X will access the data.
	.KeepTogether	Store all files in a directory subtree on the same set of servers.
	.RepSites=N	Store replicas across N different sites.
Durability	**.RepLevel=N**	Keep N replicas for a data object. The maximum allowed replicas is four.
	.SyncLevel=N	Make an update return as soon as N out of all replicas have accepted it, thus trading durability for write performance.
Large reads	**.WholeFile**	Enable pre-fetching of an entire file upon the first read request.
	.Hotspot	Maximize the aggregate read throughput of many clients by cooperatively fetching random file blocks from clients' cache.

Table 1: Semantic cues.

that file, the primary acknowledges any outstanding writes as soon as it has written them to its local storage module, and that storage module has received $N - 1$ acknowledgments from backup servers.

5.4 Placement

Because of the high latency and low bandwidth of wide-area networks, applications can benefit from being able to store 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.

WheelFS allows applications to specify placement information upon the creation of a file or directory. WheelFS supports three different placement cues, all of which we have found useful. First, 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 **.Site=rice.edu**. If the remote site is temporarily unreachable, WheelFS will store the newly created file on the local node if the file also has the **.EventualConsistency** cue. `maint` will eventually transfer the misplaced file to the desired site. If no eventual consistency is specified, file creation will fail or block if the remote site is unreachable.

Second, the **.KeepTogether** cue indicates that an entire sub-tree should reside on as few storage nodes as possible. Clustering a set of files together is useful to reduce the delay for operations that access multiple files in the set. For example, an email system would like to store all of a given user's message files on few nodes so that it takes less time for a user to get a list of all messages.

Third, the **.RepSites=N** cue indicates how many different sites should have copies of the data. The number of replica sites must be smaller than the replication level. For example, specifying **.RepSites=2** with a replication level of three causes two replicas to be kept at one site and the third replica at a different site. Replicating data at fewer wide-area sites improves write performance because less data is transferred over the wide area. On the other hand, keeping all replicas at a single site makes applications more susceptible to correlated failures within a site.

The configuration service and the clients cooperate to implement these cues. The configuration service knows, for each storage node, the name of the site at which it resides. The configuration service has a policy governing, for each slice, from which site it must choose the slice's primary, and from how many distinct sites it must choose the backups. It includes the current policy in the slice assignment that all clients fetch. When creating a new object with a placement cue, a client consults the policy to find a slice suitable for the object.

5.5 Large reads

WheelFS provides two cues to enable large-file read optimizations: **.WholeFile** and **.Hotspot**. The **.WholeFile** cue instructs the `Client` module to start pre-fetching the entire file into the client cache when the application first opens the file. The **.Hotspot** cue indicates that a file might be read by many nodes within a short time window (*e.g.*, binaries at application startup time). To optimize for these read hotspots,[1] client nodes fetch from each other's caches in a style similar to BitTorrent [14] and Shark [9]. To implement the **.Hotspot** cue, a file's primary maintains a list of `Client` modules that have recently cached parts of the file. `Client` modules use Vivaldi coordinates [15] to choose nearby nodes from which to fetch data.

When the cues **.WholeFile** and **.Hotspot** are used together, `Client` modules pre-fetch blocks at random rather than sequentially from other clients' caches, to optimize the collective throughput [9]. Unlike the cues de-

[1] We considered naming this cue **.slashdot**, but decided its inclusion in paths might lead to rather hard-to-pronounce pathnames.

```
1  NAME=`hostname`
2  TIME=`date +%s`
3  FILE=$TIME.$NAME.dat
4  D=/wfs/ping
5  NODES=$D/nodes
6  RESULTS=$D/results/
7  BIN=$D/bin/.EventualConsistency/.
   MaxTime=5000/.HotSpot/.WholeFile
8  PING=$BIN/ping
9  PROCESS=$BIN/process
10 TMP=/tmp/$FILE
11 DATA=$D/.EventualConsistency/dat
12 mkdir -p $DATA/$NAME
13 cd $DATA/$NAME
14 xargs -n1 $PING -c 10 < $NODES > $TMP
15 cp $TMP $FILE
16 rm $TMP
17 if [ $NAME = "node1" ]; then
18   mkdir -p $RESULTS
19   $PROCESS * > $RESULTS/$TIME.out
20 fi
```

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

scribed earlier, the **.WholeFile** and **.Hotspot** cues 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.

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 fulfils that goal by describing four applications that have been built using it.

All-Pairs-Pings. All-Pairs-Pings [38] 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 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 easy distributed tasks easy, while providing good failure behavior. 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.

Caching CDN. The WheelFS-based CDN consists of hosts running Apache 2.2.4 caching proxies (`mod_disk_cache`) and WheelFS. 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 desired page in the cache directory on WheelFS, it fetches it from the origin Web server and writes a copy in the WheelFS directory as well as serving it to the requesting browser. Other CDN nodes will then be able to read the page from WheelFS instead of contacting the origin server, thereby reducing its load. To cope with popular web files, the line in the configuration file specifies the **.Hotspot** cue so that WheelFS clients fetch from each others' caches to increase total throughput. The **.EventualConsistency** cue allows clients to read files and directory entries, and to create new files, even if the primary server cannot be contacted. The **MaxTime** cue ensures that WheelFS won't delay Apache by more than a second. If Apache tries to read a file that WheelFS cannot find quickly, WheelFS will return an error, causing Apache to fetch the page from the origin Web server.

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

Mail service. We use WheelFS to construct a large-scale email service with standard protocols (SMTP and IMAP). All nodes in WheelFS-email run as a file server and WheelFS-email stores messages as individual files replicated over multiple sites. Each node runs an unmodified sendmail process to accept incoming mail. Sendmail stores messages in maildir format via procmail that writes each new email in a separate file. The separate files help avoid conflicts from concurrent message arrivals, since any site could receive mail for any user. Procmail stores message files for user U in the following directory: /wfs /mail**/.EventualConsistency,Site=X,KeepTogether,RepSites=2** /U/Maildir/new/. Each node also runs a Dovecot IMAP server [17], and a user retrieves mail via a nearby IMAP server using a locality-preserving DNS service [18].

The **.EventualConsistency** cue maximizes availability: Any node can accept a message for storage even if all nodes in the replica set are down. If the primary server is unavailable, a user can continue reading emails from backup servers. The **.Site=X** cue indicates the desired primary location for a user's emails. In particular, X is configured for be the closest site to the user's usual location so that reading emails is fast. The **.KeepTogether** cue causes

all emails of the same user to be stored on a single replica set. Minimizing the spread of file storage improves the latency of operations involving multiple files such as listing multiple emails in a user's inbox [30]. WheelFS-email uses the default replication level of three but adds an additional **.RepSites=2** cue to keep at least one off-site replica of each mail. To avoid the unnecessary replication of temporary lock files and soft state such as an IMAP index, Dovecot uses **.RepLevel=1** for much of its internal data

WheelFS-email has the same goal as the cluster email service Porcupine [30], namely, to provide scalable email storage and retrieval with high availability. Unlike Porcupine, WheelFS 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 WheelFS-email allows it to re-use existing software like sendmail, procmail, and Dovecot for writing and retrieving emails. Both Porcupine and WheelFS-email use eventual consistency to maximize availability but Porcupine has a better reconciliation policy as its "deletion record" prevents deleted emails from re-appearing.

File Distribution. Any large file can be distributed to many clients efficiently by simply storing the file in WheelFS and using the large read cues, (*i.e.*, /wfs/**.WholeFile,Hotspot/**largefile). The **.Hotspot** cue maximizes throughput and improves scalability for flash-crowds by fetching blocks from other clients' caches. The **.WholeFile** cue prefetches all of the file, so that WheelFS can fetch blocks while the application is busy and so that blocks do not have to be fetched in the order the application is reading the file. When the two cues are used together, WheelFS fetches blocks in a random order, so clients can utilize each others' caches even if they start reading almost simultaneously.

We envision efficient file distribution to be particularly useful in distributing binaries for wide-area experiments, in the spirit of Shark [9] and CoBlitz [27]. 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 clients, so the nodes have very high throughput to each other, but throughput to clients is limited by the CDN.

7 Implementation

The current prototype of WheelFS consists of 14,600 lines of C++ code, using pthreads and STL. In addition, the implementation uses a new RPC library (2,600 lines) that implements Vivaldi network coordinates [15].

The `Client` module uses FUSE's "low level" interface to get access to FUSE identifiers, which it translates into WheelFS-wide unique object IDs. The WFS cache layer in `Client` module buffers writes in memory and caches file blocks in memory and on disk.

The configuration service is implemented as a set of replicated state machines. For maximum availability, we use five configuration servers distributed across three sites.

Permissions and access control are implemented. Secure SSH connections are not yet implemented.

8 Evaluation

This section shows that the WheelFS applications of Section 6 perform well enough across the wide-area to achieve their main goals. Ideally, we would like to compare to the original applications in their intended environments, but for many of the applications we don't have access to the original source code (*e.g.*, Porcupine, CoDeen, CoBlitz) or we don't have a large-enough testbed (*e.g.*, CoralCDN), because only a small subset of the PlanetLab nodes supports FUSE. We can demonstrate, however, that the WheelFS versions achieve their primary goals (e.g., reducing the load on the server in the case of a CDN), and that WheelFS handles much of the burden of dealing with the wide-area network. The experiments involve the CDN, mail, and file distribution applications on both Emulab and a small wide-area Internet testbed, and include comparisons with CoralCDN and BitTorrent.

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, and a default replication level of three (the responsible server plus two replicas), unless stated otherwise. Each node runs both a WheelFS server and a WheelFS client process, with one node also acting as a configuration server.

Even though WheelFS and the applications run on PlanetLab, we don't use PlanetLab in this evaluation, because we cannot isolate WheelFS's performance from other applications running on Planetlab—instead of being bottlenecked by the disk or the wide-area network the applications are bottlenecked by CPU performance. In addition, only a very small set of PlanetLab nodes currently support FUSE-based file systems. For experiments in the real world, we use a small, private, wide-area testbed consisting of 20 nodes located at three different sites: MIT, NYU, and Stanford. These nodes all run on a Linux 2.6 kernel, vary in processing speed between 2.4 and 3.4 GHz, and contain a single SCSI or IDE disk.

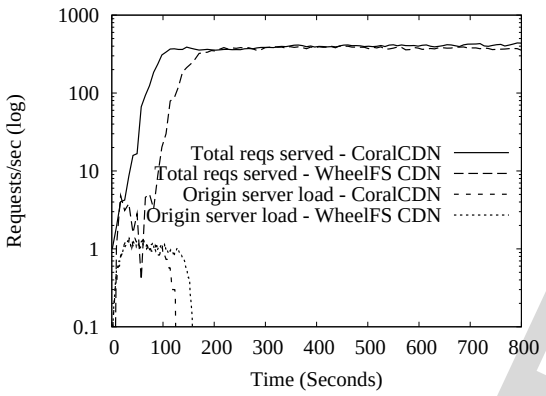

Figure 3: The aggregate client service rate and origin server load for both CoralCDN and WheelFS CDN, running on the real-world testbed.

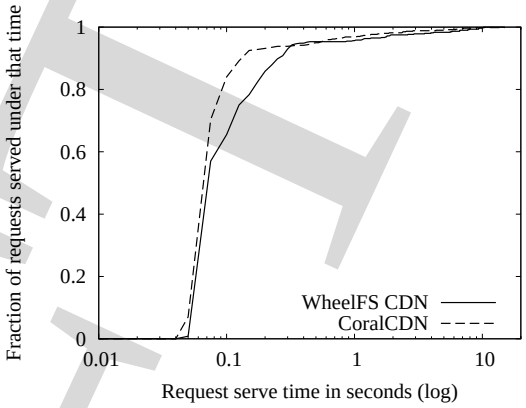

Figure 4: The CDF for the client request latencies of both CoralCDN and WheelFS CDN, running on the real-world testbed.

For more control over the network topology, we also run experiments on the Emulab [44] 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 Caching CDN

Performance under normal conditions. These experiments compare the performance of CoralCDN and the WheelFS CDN (as described in Section 6). The main goal of the CDN 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 the private Internet testbed discussed in Section 8.1. A Web server, located at NYU behind an emulated slow link (shaped using Click [22] to be 400 Kbps and have a 100 ms delay), serves 100 unique 41KB Web pages. Each of the 20 testbed nodes runs a web proxy. PlanetLab nodes located less than 10 ms from each testbed site act as clients; for each proxy node, there is a client process continually requesting randomly selected pages from the NYU web server. This experiment, inspired by one in the CoralCDN paper [18], models a flash crowd where a set of files on an under-provisioned server become popular very quickly.

Figures 3 and 4 show the results of these experiments. Figure 3 plots both the request load seen by the origin server and the aggregate number of client requests served by each CDN over time. While CoralCDN reduces the origin server load, and caches copies of the pages on all proxies, more quickly, WheelFS performs comparably and reaches a similar sustained aggregate client service rate. Figure 4 plots the cumulative distribution function (CDF) of the individual request latencies seen by the clients throughout the course of the experiment. WheelFS closely matches CoralCDN for the majority of requests.

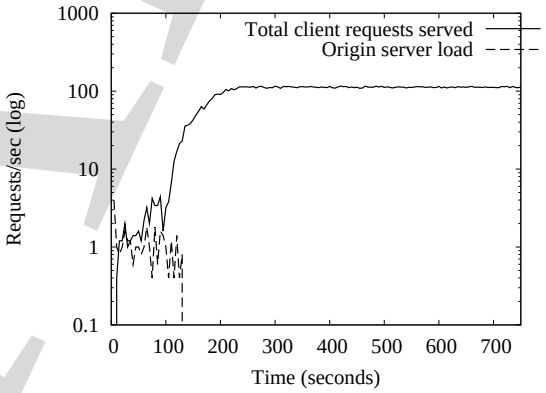

Figure 5: The aggregate client service rate and origin server load for WheelFS CDN, running on Emulab, without failures.

Because CoralCDN contains many optimizations specific to a distributed caching proxy system, it outperforms WheelFS in some respects. For instance, a CoralCDN proxy initially registers its intent to download a specific page before doing so, preventing other nodes from needlessly downloading the same page; Apache, running on WheelFS, has no such mechanism, so initially several nodes may download the same page before Apache caches the data in WheelFS. However, we believe many of these optimizations could be implemented in Apache with minor modifications (in this case, a simple lock file would do the trick).

Performance under failures. Wide-area network problems that prevent WheelFS from contacting storage nodes should not translate into long delays for proxy clients; 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**

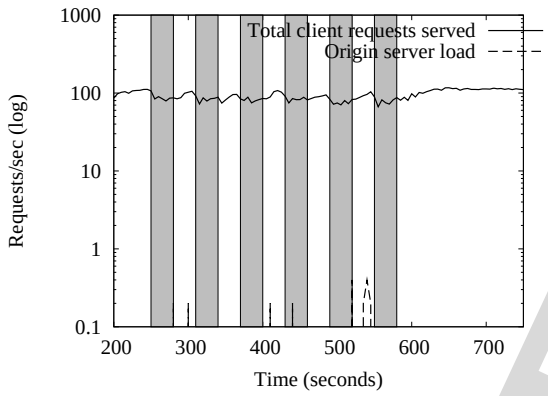

Figure 6: WheelFS CDN running on Emulab with failures, using the **.EventualConsistency** cue. Gray regions indicate the duration of a failure.

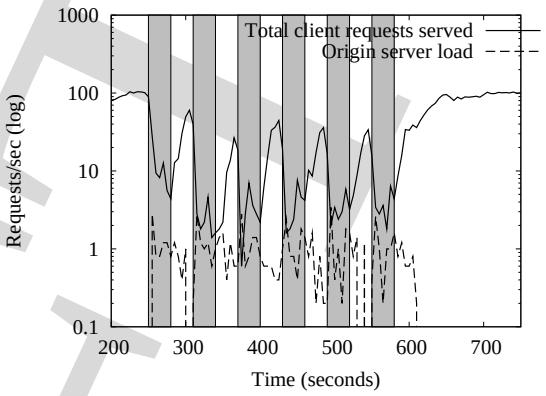

Figure 7: WheelFS CDN running on Emulab with failures, with close-to-open consistency. Gray regions indicate the duration of a failure.

yield this behavior, causing `open()` to either yield locally cached data or fail quickly if the WheelFS client cannot contact storage nodes. Apache fetches from the origin web server if the `open()` fails.

Figures 6 and 7 compare failure performance of WheelFS with the above cues to failure performance of the default close-to-open consistency with 1-second timeouts. Each minute one wide-area link connecting an entire LAN site to the rest of the network fails for thirty seconds and then revives. This failure period is not long enough to cause view changes in WheelFS, so the servers in the LAN will still be responsible for data they originated. These experiments took place on the Emulab topology described in Section 8.1, using additional client nodes connected to each site using a 10 Mbps, 20 ms link—clients maintain connectivity to their site during failures. The origin Web server run behind a 400 Kbps link, with 175 ms RTT to the client nodes. For comparison, Figure 5 shows the performance of WheelFS on this topology when there are no failures.

Under eventual consistency, a proxy can use a locally-cached copy of a Web page even if it cannot access the file's primary server (as long as the object lease for the content is still valid). Under the default close-to-open consistency, a proxy cannot use a locally cached copy of a page if the lease has expired and it cannot check its freshness because the primary server has failed, and in that circumstance will be forced to download the page anew from the origin server.

Figure 6 shows the performance of the WheelFS CDN with eventual consistency. The graph shows a period of time after the initial cache population. The gray regions indicate when a failure was present. Throughput does fall as timeouts are incurred during a failure, though overall the aggregate client service rate remains steady near 100 requests/sec. Figure 7 shows that with close-to-open consistency, throughput falls as soon as a failure occurs, and hits to the origin server increase greatly. This indicates that by exploiting the fact that a CDN does not require strong consistency, we can use WheelFS's semantic cues to build a simple CDN that performs well under wide-area conditions.

Stat	Median	Mean	Max
Email size	4.6 KB	6.85 KB	4.11 MB
Msgs rcvd/day/user	3.86	5.96	2840
IMAP logins/day/user	5.05	18.54	4791

Table 2: Statistics of our mail workload distribution, taken from NYU mail server traces.

8.3 Mail

To test the performance of our WheelFS mail system (as detailed in Section 6, we obtained a trace of SMTP and IMAP operations from the mail servers at New York University. The SMTP trace contains 43 days of logs from half of NYU's SMTP servers, while the IMAP log contains 3 days of logs for NYU's IMAP server. To get an overall estimate of the workload seen by a real mail system, we scaled the IMAP trace up to 43 days (*i.e.*, we multiplied the total number of IMAP logins seen for each user by $\frac{43}{3}$) and looked at the intersection of users that appear in both traces. The resulting distribution covers 47699 users; its salient statistics appear in Table 2.

IMAP and SMTP operations are a stressful file system benchmark. For instance, a typical IMAP operation (reading a maildir-formatted inbox and finding no new messages) generates over 600 FUSE operations on the server that handles the request. These primarily consist of lookups on directory and file names, but also includes more than 30 directory operations (creates/links/unlinks/renames), more than 30 small writes, and a few small reads.

In this experiment we use Emulab network described in Section 8.1, deploying WheelFS on 15 nodes in 5 clusters. Clients, running as threads on five additional nodes, con-

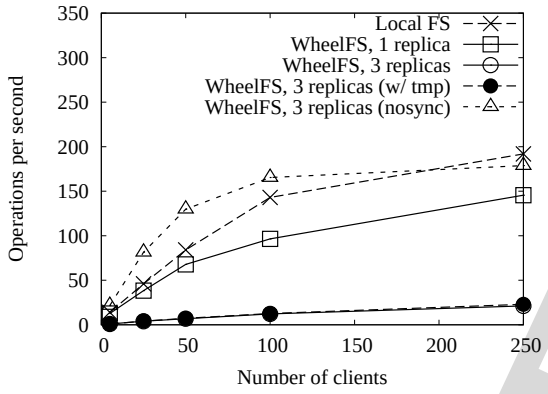

Figure 8: The throughput of our WheelFS wide-area mail system, compared with a static, unreplicated system.

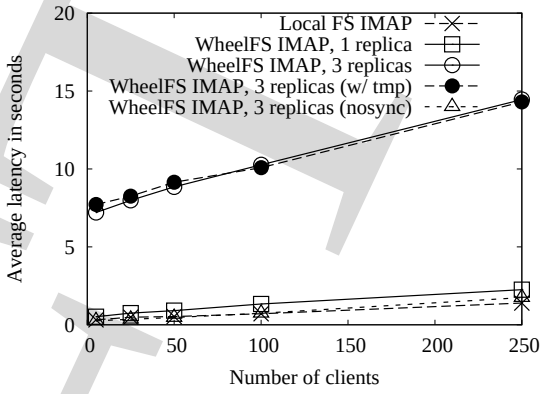

Figure 9: The average latencies of individual IMAP and SMTP operations, for both WheelFS and a static, unreplicated system.

nect to servers in their closest cluster (their "home" site) over a shared wide-area link (1 Mbps, with a 20 ms RTT to the site). We measure throughput in operations/second (either checking an inbox or sending a message), with an increasing number of concurrent clients. Each client performs 20 operations according to the distribution in Table 2. Users always read their mail from the site in which their mailbox directory resides, but send mails to random users. A node at the sender's "home" site writes these new mails to WheelFS, though the recipient's mailbox directory is likely to reside at a different site. User mailbox directories are randomly and evenly distributed across sites, and when replicated, each file is replicated at multiple unique sites.

We compare against a mail system that uses a local file system. Users still check their mail at their "home" site, but the clients have a static table (acting like DNS MX records) to look up the location of a recipient's mailbox, and forward each incoming message explicitly to that node. There is no replication, so at most one copy of every email is sent across the network and stored.

Figure 8 shows the aggregate number of operations served by the entire system per second. When WheelFS runs without replication (**.RepLevel=1**), it can handle 75% of the load handled by the mail system that uses local storage when faced with 250 concurrent clients—about 145 operations/sec. This difference in throughput, however, is explained by the fact that WheelFS stores meta-data for each object in a file on the local file system, and must update the version number stored in the meta-data for every modifying operation (creates/renames/unlinks/writes/etc.) This results in twice the amount of disk writes than is performed by the mail system on the local file system. Keeping this meta-data in memory allows WheelFS to achieve very similar performance to the local version, but a more durable solution would be to optimize carefully WheelFS's on-disk data and meta-data layout.

When files are replicated at three unique sites (**.RepLevel=3**,**.RepSites=3**), throughput drops by a factor of 3.5 when there are 250 concurrent clients, due to the extra work done by the nodes and the network.

Figure 9 shows the average latencies of individual IMAP operations for each system configuration, as the number of clients varies. We omit SMTP operation latencies due to space constraints. The unreplicated WheelFS system offers higher latencies than the ones offered by the local, static system by nearly 60%, and again this is attributable to extra disk accesses. The replicated version incurs penalties due to synchronous file replication across multiple sites.

However, if the administrator of the system is willing to use the **SyncLevel=0** cue to trade off durability for performance, then WheelFS-email achieves latencies and service rates even better than the local system. These are indicated by the "WheelFS, 3 replicas (nosync)" lines on Figures 8 and 9. In this configuration, the primary storage module does not immediately `fsync` writes to disk, and returns to the client before receiving acknowledgements from backup replicas.

As Section 6 mentions, Dovecot writes temporary lock files and soft-state index files to WheelFS. For performances reasons, these files are unreplicated by default. To highlight the benefit of being able to assign different files different replication levels in the same file system, Figures 8 and 9 also include results for when these files are replicated at multiple sites, labeled "WheelFS, 3 replicas (w/ tmp)". This configuration incurs a modest latency penalty for IMAP operations. In this case, the ability to specify replication policy via cues at a fine granularity significantly impacts performance.

8.4 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} node that is in its own cluster) using the **.Hotspot** and **.WholeFile**

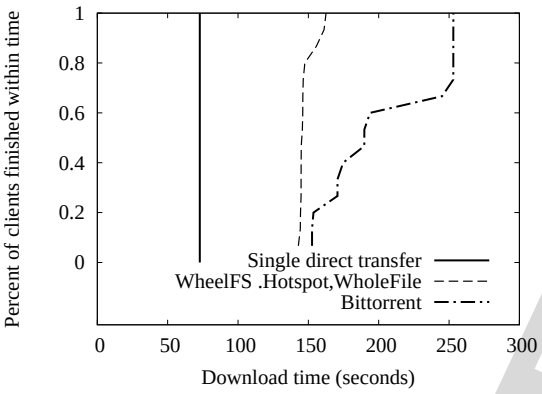

Figure 10: CDF of client download times of a 50 MB file using BitTorrent and WheelFS with the **.Hotspot** and **.WholeFile** cues. Also shown is the time for a single client to download 50 MB directly using `ttcp`.

cues. For comparison, we also fetch the file using BitTorrent [14]. We configured BitTorrent to allow unlimited uploads and to use 64 KB blocks like WheelFS (in this test, Bittorrent performs strictly worse with 256 KB blocks).

Figure 10 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 146 seconds, showing that WheelFS's implementation of cooperative reading is slightly better than BitTorrent's: Bittorrent clients have a median download time of 191 seconds. The improvement is due to WheelFS clients fetching from nearby nodes according to Vivaldi coordinates[2]; Bittorrent does not use a locality mechanism. Of course, both solutions offer far better download times that 15 simultaneous direct transfers from a single node, which in this setup has a median download time of 892 seconds.

8.5 Summary

When possible, this section compared WheelFS applications to similar existing distributed applications. The WheelFS applications perform nearly as well as these custom, optimized applications, despite reusing stock software designed to run on local file systems. Used in this way, WheelFS simplifies the implementation of distributed applications without greatly sacrificing performance, by providing a distributed storage layer that offers an API familiar to many existing applications.

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 [10] is a recently-proposed framework for building storage systems with arbitrary consistency guarantees (as in TACT [46]). Like PRACTI, WheelFS maintains flexibility by separating policies from mechanisms, but it has a different goal than PRACTI. While PRACTI and its recent extension PADRE [11] 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).

There exist numerous distributed file systems, but most are designed to support a workload generated by desktop users (*e.g.*, NFS [32], AFS [33], Farsite [5], xFS [8], Frangipani [12], Ivy [25]). As such, they strive to always provide a consistent view of data, while sometimes allowing for disconnected operations (*e.g.*, Coda [34] and BlueFS [26]). Cluster file systems such as GFS [20] and Ceph [43] 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 [6] offers highly-scalable cluster storage for infrastructure applications, and allows some degree of intra-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.

Successful wide-area storage systems generally exploit application-specific knowledge to make decisions about trade-offs in the wide-area environment. As a result, many wide-area applications include their own storage layers [7, 14, 18, 30] or adapt an existing system [9, 27, 41]. Unfortunately, most existing storage systems, even more general ones like OceanStore/Pond [28] 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 [35] and the Coral Content Distribution Network (CDN) [18] both use a DHT, but their DHTs differ in many details and are not interchangeable.

Some wide-area storage systems have begun to 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 con-

[2]Without using Vivaldi coordinates, WheelFS achieves a median download time of 300 seconds. However, this leaves WheelFS without any optimizations for preferring fast nodes to slow nodes, whereas BitTorrent does.

sistency tradeoffs. By contrast, WheelFS's cues are at a higher level (*e.g.*, eventual consistency versus close-to-open consistency). Bayou [40] and Pangaea [31] provide eventual consistency by default while the latter also allows the use of a "red button" to wait for the acknowledgement 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

The results from Section 8 indicate that WheelFS meets its goals as a flexible, wide-area distributed storage layer. Using WheelFS real-world applications can be built easily by reusing existing software while performing well on wide-area networks by maintaining sufficient control over consistency, placement, and failure handling through cues. We hope that these results will encourage the adoption of file systems on testbeds and across data centers, and thereby simplify the construction of future wide-area applications.

References

[1] Amazon simple storage system. `http://aws.amazon.com/s3/`.

[2] The hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[3] The linq project. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`.

[4] MySQL. http://www.mysql.com.

[5] 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).

[6] 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).

[7] 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).

[8] 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).

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

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

[11] BELARAMANI, N., ZHENG, J., NAYATE, A., SOULÉ, R., DAHLIN, M., AND GRIMM, R. PADRE: A Policy Architecture for building Data REplication Systems, 2008. http://www.cs.utexas.edu/users/dahlin/papers/padre-may-2008-extended.pdf.

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

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

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

[21] 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.

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

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

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

[25] 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).

[26] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th OSDI* (Dec. 2004).

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

[28] 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).

[29] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[30] 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).

[31] 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).

[32] 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).

[33] 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).

[34] 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.

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

[36] Squid caching proxy. `http://www.squid-cache.org/`.

[37] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160.

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

[39] 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).

[40] 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).

[41] 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).

[42] WANG, L., PAI, V., AND PETERSON, L. The effectiveness of request redirecion on CDN robustness. In *Proceedings of the 5th OSDI* (Dec. 2002).

[43] 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).

[44] 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).

[45] YEE JIUN SONG, V. R., AND SIRER, E. G. Optimal resource utilization in content distribution networks. *Cornell University, Computing and Information Science Technical Report TR2005-2004* (Nov. 2005).

[46] 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.