Flexible, Wide-Area Storage for Distributed Systems Using Semantic Cues

by

Jeremy Andrew Stribling

B.S., University of California, Berkeley (2002)
S.M., Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author . . . .
Department of Electrical Engineering and Computer Science
August 31, 2009

Certified by . . . .
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by . . . .
Robert Morris
Professor
Thesis Supervisor

Certified by . . . .
Jinyang Li
Assistant Professor, NYU
Thesis Supervisor

Accepted by . . . .
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Flexible, Wide-Area Storage for Distributed Systems Using Semantic Cues
by
Jeremy Andrew Stribling

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

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 dissertation explores the use of a new file system (WheelFS) specifically designed to be the storage infrastructure for wide-area distributed services.

WheelFS allows applications to adjust the semantics of their data via *semantic cues*, which provide application control over consistency, failure handling, and file and replica placement. This dissertation describes a particular set of semantic cues that reflect the specific challenges that storing data over the wide-area network entails: high-latency and low-bandwidth links, coupled with increased node and link failures, when compared to local-area networks. By augmenting a familiar POSIX interface with support for semantic cues, WheelFS provides a wide-area distributed storage system intended to help multi-site applications share data and gain fault tolerance, in the form of a distributed file system. 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 is implemented as a user-level file system and is deployed on PlanetLab and Emulab. Six applications (an all-pairs-pings script, a distributed Web cache, an email service, large file distribution, distributed compilation, and protein sequence alignment software) 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.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Robert Morris
Title: Professor

Thesis Supervisor: Jinyang Li
Title: Assistant Professor, NYU

Prior Publication

Much of this thesis was previously published in a conference paper [73], and represents the joint work of the coauthors of that paper. An earlier design of WheelFS was published in a workshop paper [72]. Finally, an expanded version of some components of WheelFS appear in two recent Master's theses [57, 86].

Acknowledgments

Sometimes I look back at these last six years of graduate school and wonder whether I did any work at all, or if I just managed to channel the energy, intellect, and support of everyone around me into a coherent curriculum vitae, attaching my name to it in the process. If MIT allowed it, I would list everyone in this section as coauthors, but I guess they'll have to settle for a brief, inadequate mention here at the beginning of this dissertation.

My advisors, Frans Kaashoek and Robert Morris, are incredible. They patiently turned me from an arrogant, smart-mouthed California prankster into an arrogant, smart-mouthed California prankster who might actually know something about distributed systems. Frans's constant enthusiasm for research fueled me during my time at MIT, and I suspect that most of the good ideas I had were secretly planted there by Frans, and he convinced me they were my own. He was overly generous with his time and his genius, and I forgive him for making me organize graduate student visit weekend for five years in a row. Robert balanced his careful, thorough skepticism of every word, graph, and thought I presented him with a strong undercurrent of encouragement and support, which improved me greatly as a researcher and a person. Without his dry wit and occasional toy distribution, I might not have survived MIT. They both tolerated, and sometimes even supported, the distracting side projects I tend to succumb to, and always let me find my own way back to research productivity. In my mind, they contributed ten times more to this dissertation than I did, and seemingly with a tenth of the effort.

The title page of this dissertation lists Jinyang Li as a thesis supervisor, but that doesn't do her justice. She was a mentor to me from the very beginning of graduate school, and let me tag along on countless research projects when I was just learning the ropes. It was always a pleasure writing code and papers with her, even if they sometimes ended up in the trash can. Her meticulous approach to research and willingness to speak her mind has been a huge influence, in this dissertation and in my life.

I've had so much helping building and evaluating WheelFS that it seems like I was just along for the ride. Robert built the initial, elegant, small prototype that became the WheelFS beast, including the RPC code described in Section 6.1.1. Frans Kaashoek wrote the initial Paxos and replicated state machine code used in the configuration service (Section 6.1.2), and inspired the `frans_cachelock` variable buried deep within the WheelFS cache code. Jinyang built the first version of the cooperative client described in Section 6.1.4, and ran the BLAST experiments presented in Section 7.7. Yair Sovran was a pleasure to work with remotely on several parts of the codebase, including the configuration service and the WheelFS logging infrastructure; furthermore, Yair ran the PlanetLab CoralCDN versus WheelFS experiments presented in Section 7.3. Our WheelFS Master's students, Xavid Pretzer and Irene Zhang, did great work on numerous parts of the system [57, 86], and I hope that they found my inexperienced attempts at supervision charming, rather than messy and unconstructive. Emil Sit helped us forge a vision, and a name, for WheelFS back when it was just a jumble of thoughts scribbled down during a red-eye flight.

Sam Madden graciously contributed his time as a reader on my thesis committee, and his fresh insight, unbiased by PDOS tunnel vision, greatly improved this work. His willingness to entrust Milo to me occasionally cheered me up beyond measure. Nickolai Zeldovich sat through many WheelFS practice talks and discussions, and his abundant advice was always on target. Michael Freedman helped us realize what it takes to write a real distributed system, and helped us get our footing. Jeremy Kepner helped us understand how WheelFS could fit into the world of scientific computing, Bud Mishra and Shaila Musharoff from NYU instructed us in how BLAST is used by biologists, and Luis F. G. Sarmenta provided some helpful related work pointers. Jeff Dean, the shepherd of the WheelFS NSDI paper and all-around distributed systems virtuoso, put a great deal of thought into what must have seemed like a cute, research-grade storage system, and helped us improve it immensely. The anonymous reviewers of all four submitted WheelFS publications deserve much credit in shaping the final product. You know who you are.

WheelFS also had an impressive field team, people who helped us deploy and evaluate it at no benefit to themselves. The PlanetLab development team, in particular Larry Peterson, Marc Fiuczynski, Sapan Bhatia, Daniel Hokka Zakrisson, and Faiyaz Ahmed, went above and beyond in assisting our deployment of a dangerous, unknown, FUSE-based mess. Without beautiful, malleable Emulab, we could never have gotten WheelFS off the ground. Many thanks to the Emulab staff, and in particular Jay Lepreau, who I fondly remember encouraging my work on WheelFS after a long day of bike-riding through Oregon. Jay, you are missed. In addition to PlanetLab and Emulab, WheelFS ran on machines at Stanford, thanks to Jinyuan Li and David Mazières. Michael Puskar and the NYU ITS team provided us with the NYU mail traces that drive the evaluation in Section 7.4. I am also indebted to the open source developers of the FUSE and `libssh` projects.

This research was supported by NSF grant CNS-0720644 and by Microsoft Research Asia and Tsinghua University.

I couldn't have made it through graduate school without the protective embrace of the PDOS research group. Max Krohn's brilliance and pessimism kept me in check since visit weekend 2003, and he was always generous with his bug fixes, advice, context-free grammar skills, Drudge rumors, and poker chips. Dan Aguayo left us too early, but his legacy will forever live on in the Dan Aguayo Memorial Vob Store, housed in the Dan Aguayo Laboratory. I thank Frank Dabek for helping me in countless ways, at MIT and at Google, and for calming down and not shooting me in the face when I showed up at MIT. Russ Cox was an inexhaustible fount of knowledge, and always had exactly the answer I needed. Russ, Emil, Frank and Max selflessly administered PDOS's servers during most of my time here, and made all my work at MIT possible. Alex Yip, Silas Boyd-Wickizer, Alex Pesterev, and Neha Narula helped me (among many other things) decide where to eat lunch every day, which has been more important to me than it sounds. Thomer Gil was supporting my n00b hacking skills since the summer before I officially started at MIT, and found a typo in the abstract of this dissertation. I don't have the space or time to thank every PDOS member with whom I overlapped individually, but they've all made my time here amazing: Athicha Muthitacharoen, Austin Clements, Benjie Chen, Bryan Ford, Chris Lesniewski-Laas, Chuck Blake, Cliff Frey, Dave Andersen, Doug DeCouto, Jacob Strauss, Jayashree Subramanian, John Bicket, Kevin Fu, Micah Brodsky, Michael Kaminsky, Petar Maymounkov, Sanjit Biswas, Xi Wang, and whoever else I stupidly forgot. Many PDOS alums gave assistance to the weird new kid, without any reason to: David Mazières, Eddie Kohler, Alex Snoeren, John Jannotti, and Emmett Witchel come to mind in particular. Neena Lyall reimbursed me, scheduled me, booked me, and postcarded me since the beginning. Finally, I'd be remiss if I didn't acknowledge NTT Class of 1974, as others have before me.

Hundreds of others CSAILers guided and influenced me as well. Michael Walfish was the V to my JV (to quote the man himself) and introduced too many new adjectives, machines names, variable names, and pejoratives into my vocabulary to count. Hari Balakrishnan advised and supported me even though I wasn't his problem, and took a chance giving me my first TA position. David Karger was always willing to imbue my research with his brilliance. Even though I probably seemed like a PDOS recluse, everyone in NMS and PMG have been incredibly friendly and helpful to me during my time here, including (but definitely not limited to) Rodrigo Rodrigues, Magdalena Balazinska, Michel Goraczko, James Cowling, Sachin Katti, Nick Feamster, and Sheila Marian. CSAIL Exponential Run Time gave me five springs and summers of exponentially-increasing good times.

Of course, the path to my doctorate started well before MIT. My time as an undergraduate and post-undergraduate with John Kubiatowicz and Anthony Joseph at UC Berkeley was crucial, and I was lucky to fall into their orbits when I was so impressionable. The graduate students I worked with on the Oceanstore and Tapestry projects put up with me and showed me what graduate school really looked like. Sean Rhea is particularly responsible for getting me where I am today; Ben Zhao, Hakim Weatherspoon, Dennis Geels, and Patrick Eaton share the blame as well. Scott Shenker, who I didn't know while I was in Berkeley, became a huge force in my research career during my time at MIT, and can't be thanked enough for that. Leo (Shyh-Wei) Luan at IBM Almaden connected me, mostly indirectly, to everyone I've listed up to this point, and started me off in research in the first place.

Nagib Callaos deserves much of the credit for my fifteen minutes of fame [71], as do Max and Dan, the SCIgen donors, Dr. Thaddeus Westerson, Dr. Mark Zarqawi, and Franz T. Shenkrishnan, PhD.

And those are just the people who influenced my research directly. There's no way I can ever list all of the amazing people in my life who got me this far. My friends in the Bay Area, Cambridge, in between and beyond kept me sane, grounded, and laughing throughout everything. My entire extended family, but especially my brother, sister, and parents have been nothing but supportive of me in my long absence from home, and I've been lucky to have their confidence, patience, pride, nurturing and love throughout my life. And without Eleanor Baird by my side through the most difficult times of this journey, the accomplishment wouldn't be half as sweet.

Thank you, everyone.

Contents

1 Introduction **17**
1.1 Motivation . . . 18
1.2 Challenges . . . 18
1.3 Approach . . . 19
1.3.1 Semantic cues . . . 19
1.3.2 File system interface . . . 19
1.4 WheelFS . . . 20
1.5 Contributions . . . 21
1.6 How to Read this Dissertation . . . 22

2 Wide-Area Storage Overview **23**
2.1 General Properties . . . 23
2.2 Examples of Wide-Area Applications . . . 24
2.2.1 A distributed Web cache . . . 24
2.2.2 A distributed mail service . . . 24
2.3 WheelFS Overview . . . 25
2.3.1 System model . . . 25
2.3.2 System overview . . . 25

3 Semantic cues **27**
3.1 Specifying Cues . . . 27
3.1.1 Cues in pathnames . . . 27
3.1.2 Alternative methods . . . 28
3.2 Categories . . . 28
3.2.1 Persistence vs. transience . . . 28
3.2.2 Using cues in practice . . . 30
3.3 Placement . . . 31
3.4 Durability . . . 31
3.5 Consistency . . . 31
3.6 Large Reads . . . 32
3.7 Discussion . . . 33

4 WheelFS Design **35**
4.1 Components . . . 35
4.2 Data Storage Assignment . . . 35
4.2.1 Slice size . . . 36
4.2.2 Configuration service . . . 37

4.2.3 Placing a new file or directory . . . 38
4.2.4 Write-local policy . . . 38
4.3 Primary/Backup Replication . . . 38
4.4 Consistency . . . 39
4.5 Large Reads . . . 40
4.6 Security . . . 41

5 Applications **43**
5.1 All-Pairs-Pings . . . 43
5.2 Distributed Web Cache . . . 43
5.3 Mail Service . . . 44
5.4 File Distribution . . . 45
5.5 Distributed Compilation . . . 45
5.6 Distributed BLAST . . . 46
5.7 Discussion . . . 46

6 Implementation **49**
6.1 Software Components . . . 49
6.1.1 RPC . . . 49
6.1.2 Configuration service . . . 50
6.1.3 Storage . . . 52
6.1.4 Client . . . 53
6.1.5 Security . . . 54
6.2 Deployment . . . 54
6.2.1 Architecture . . . 55
6.2.2 Expected bottlenecks . . . 56

7 Evaluation **57**
7.1 Experimental Method and Setup . . . 57
7.2 Scalability . . . 58
7.3 Distributed Web Cache . . . 59
7.3.1 Performance under normal conditions . . . 59
7.3.2 Performance under failures . . . 61
7.4 Mail . . . 63
7.5 File Distribution . . . 66
7.6 `distmake` Under Failures . . . 66
7.7 Distributed BLAST . . . 67
7.8 Implementation Ease . . . 68
7.9 Expressiveness . . . 69
7.9.1 Feature comparison . . . 69
7.9.2 Storage equivalence . . . 71
7.10 Discussion . . . 72

8 Related Work **73**
8.1 Storage Systems with Wide-Area Tradeoffs . . . 73
8.2 Distributed File Systems . . . 74
8.2.1 Centralized servers . . . 74
8.2.2 Local-area distributed file systems . . . 75

8.2.3 Wide-area distributed file systems 75
8.3 Miscellaneous Distributed Storage Systems 76

9 Discussion, Future Work and Conclusion **77**
9.1 Semantic Cues 77
9.2 Another Distributed File System? 78
9.3 Future Directions 78
9.4 Conclusion 79

List of Figures

2-1 WheelFS's deployment architecture. 26

4-1 Placement and interaction of WheelFS components. 36

5-1 A shell script implementation of All-Pairs-Pings using WheelFS. 44

6-1 WheelFS's software architecture. The configuration service, storage, and client components use the RPC component (and optional security component) to communicate. The components in the top layer can, but do not have to, run on the same computer. 50

6-2 WheelFS's PlanetLab deployment architecture. MIT runs WheelFS storage servers within its vserver on various nodes, and other PlanetLab users (such as Alice) can connect to those storage servers to mount WheelFS within their own vserver. Each node must know its own private key and the public keys of the other nodes and authorized users. The configuration service is not pictured. 55

7-1 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`. 58

7-2 The aggregate client service rate for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab. 60

7-3 The origin server load for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab. 60

7-4 The CDF for the client request latencies of both CoralCDN and the WheelFS-based Web cache, running on PlanetLab. 61

7-5 The WheelFS-based Web cache running on Emulab with failures, using the **.EventualConsistency** cue. Gray regions indicate the duration of a failure. 62

7-6 The WheelFS-based Web cache running on Emulab with failures, with close-to-open consistency. Gray regions indicate the duration of a failure. 62

7-7 The aggregate client service rate and origin server load for the WheelFS-based Web cache, running on Emulab, without failures. 63

7-8 The throughput of Wheemail compared with the static system, on the Emulab testbed. . . . 64

7-9 The average latencies of individual SMTP requests, for both Wheemail and the static system, on Emulab. 65

7-10 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`. 66

7-11 The average execution speedup for a batch of 19 BLAST queries achieved by `parablast` as a function of the number of compute nodes. It takes a single compute node 1292 secs to finish all 19 queries. The error bars represent the 20^{th}- and 80^{th}-percentile speedup factors. 68

List of Tables

1.1 Example scenarios in which to use WheelFS, and the full WheelFS paths, including semantic cues, appropriate for those situations. This table assumes WheelFS is mounted at `/wfs`. Semantic cues are in bold. 21

3.1 Semantic cues. A cue can be either **P**ersistent or **T**ransient (*Section 3.5 discusses a caveat for **.EventualConsistency**). 29

3.2 The POSIX file system API calls affected by each cue. 29

6.1 WheelFS data structures. 51

7.1 The time to compile and link an executable on WheelFS using `distmake` with different build-directory cues, both with and without a node failing after compiling an object. Numbers are the average of 10 trials, and **.RepLevel=2** is used in all cases. 67

7.2 Number of lines of changes to adapt applications to use WheelFS. 68

7.3 A comparison of the features of several wide-area storage systems. For each feature, the table indicates whether the system includes the feature and, if applicable, a summary of the techniques from which an application can choose when using the feature. Section 7.9.1 describes the features in more detail. 70

Chapter 1

Introduction

This dissertation investigates the challenge of building data storage systems for geographically distributed services. Modern Internet-based applications store and process vast amounts of data to serve their users, and the component of the application that manages that data can greatly impact the application's success. Furthermore, in order to meet ambitious fault tolerance and performance requirements, these services often operate across multiple geographically-separated (*i.e.*, wide-area) sites. A storage system with an intuitive interface hiding the complexities of wide-area data transfer, replication, and consistency would be of great value to these services; the design and implementation of such a storage system is the focus of this dissertation.

Building a distributed storage system that works well across multiple wide-area sites is difficult, however. To transfer data between nodes at different sites, the storage system must cope with problems such as long latencies, constrained bandwidth, and unreliable networks, at scales not seen within a single site. Storage system developers often find that techniques developed for a single site work poorly across multiple sites.

As an example of this fundamental property of wide-area networks, consider a company with a service spanning two data centers: one on the west coast of the United States, and one on the east coast. An update to a data item written to a node in the west coast data center can easily and quickly be read by west coast users accessing the service. However, users accessing the service through the east coast data center must either wait for the data item to be transferred to the east coast, or be satisfied with reading a previously-cached (and therefore out-of-date) version of the data item. In this example, the desire to share data over the wide-area network creates a tradeoff between low read latency and data freshness; no storage system can offer both, because the freshest copy of the data must travel across a high-latency link to be read by distant users. Similar tradeoffs are also present for single-site applications, but the orders-of-magnitude difference in latency, bandwidth, and failures in the wide-area make them crucial.

This dissertation argues that an application should be able to choose how its storage system handles the tradeoffs inherent in wide-area data management. The main contribution of this dissertation is the design, implementation, and evaluation of a distributed file system called WheelFS, which gives applications direct control over these tradeoffs. The rest of this chapter discusses the motivation for WheelFS, its basic approach to wide-area data storage, the challenges inherent to that approach, and a high-level look at WheelFS's design and contributions.

1.1 Motivation

A rising number of Internet-based services are too big, or too important, to run at a single site. For example, large Web sites such as Google and Yahoo! deploy their services on multiple data centers, both to place services geographically near their many customers to increase capacity and improve latency, and also to protect against site-wide failures due to network problems, maintenance, or misconfiguration. Data growth is also tremendous at these large sites: Facebook (a popular social-networking site) currently grows at a rate of over 700,000 users a day by some estimates [30]. These users generate a massive amount of data – for instance, Facebook currently stores more than 1.5 petabytes of photos, growing at 25 terabytes per week [77]. Google, in the last ten years, has grown the number of documents it indexes by two orders of magnitude, and the number of queries it processes by three orders of magnitude [23]. To store and serve this amount of data for a global base of users requires a massive engineering effort.

Similarly, researchers developing and testing new distributed applications deploy them on PlanetLab [8], which as of this writing comprises over one thousand nodes at 481 distinct sites. CoralCDN [32] and CoDeeN [80] are two popular PlanetLab-based cooperative content distribution networks. By the end of 2005, CoralCDN served as a Web proxy for over 20 million page requests per day, for one million unique visitors [33]; each of those cached pages must be visible to each instance of CoralCDN, running on any PlanetLab node.

The need to share data across wide-area networks is common to all of these applications, however no standard wide-area storage system exists. Developers tend to build storage systems for applications, designed specifically for the domain of the application. For example, Yahoo! developed PNUTS [18] to support simple read/write queries for geographically-distributed data. Facebook's Cassandra [15], a distributed key-value store, provides support for replicating data to multiple sites. CoralCDN uses a custom distributed hash table to ensure cached copies of Web pages are distributed and found quickly. These systems vary in their details, but share the same storage goals; at a high level, each of these systems represents a newly-invented wheel, in the sense that they could all be potentially replaced by a single storage system designed to accommodate a wide range of desired behaviors. Existing storage systems differ in their behaviors because their motivating applications require specific responses to challenges inherent in storing data over the wide-area network. Thus WheelFS, a new storage system that aims to be used by a wide range of distributed applications, has a design that explicitly incorporates these challenges.

1.2 Challenges

A wide-area storage system faces a tension between sharing and site independence [81]. The system must support sharing, so that data stored by one site may be retrieved by others. However, a primary goal of multi-site operation is fault tolerance, and if the most recent version of a data item is located at an unreachable site, an application requiring perfectly fresh data must wait for the site to come back online before allowing further reads of that data item. Thus, sharing the data between sites creates an opportunity to scale the application globally, but at the same time it creates a dependence between the sites that might severely impact application performance. This tension arises from specific properties of wide-area networks. High latencies and low bandwidth between sites, and a high rate of transient network failures, cause wide-area networks to be slower and less reliable than local-area networks.

The design of any wide-area storage system implicitly affects the sharing/independence tradeoff. One example is the storage system's consistency model: stronger forms of consistency usually involve servers or quorums of servers that serialize all storage operations, whose unreliability may force delays at other sites [37]. 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.

There is no right answer for these tradeoffs, as different applications might prefer different points along the sharing/independence spectrum; a single application might even prefer different tradeoffs for different pieces of data that it manages. The choice of tradeoff is generally dictated by the semantics and performance that the application delivers to its users. While some models (*e.g.*, TACT [85] and PRACTI [9]) allow system designers to build storage systems that are tailored to particular tradeoffs, no existing storage system allows applications to control the tradeoffs directly in response to wide-area challenges.

1.3 Approach

Applications access their data from a wide-area storage system through a specific storage interface. The approach the storage system designers take to its interface determines not only its ease-of-use for programmers, but also the amount of control an application has over the performance, consistency, and replication of data.

1.3.1 Semantic cues

The motivation above implies that a wide-area storage system, in order to be useful to many different applications, should provide a simple, but flexible, interface that allows applications to make tradeoffs explicitly. Existing wide-area storage systems, however, offer control over these tradeoffs either through specialized library interfaces [18,74], or they require the system designer to make the tradeoff statically, before the storage system is deployed [9, 10, 85]. No existing wide-area storage solution offers a general, run-time solution for controlling wide-area data tradeoffs.

This dissertation proposes that wide-area storage systems augment their interface with *semantic cues*, which allow an application to control the behavior of its data in the wide-area. By specifying a cue for a particular file or directory, the application can alter the default behavior of the file system for that particular piece of data. As this dissertation shows, semantic cues can be supplied within the context of standard storage interface calls, allowing applications explicitly to make a tradeoff between performance and consistency in the style of PRACTI [9] and PADS [10], but in the context of a well-known interface.

1.3.2 File system interface

There are many reasons a storage system should offer a familiar interface by which applications can manage their data. Interfaces supported by a large number of operating systems and programming languages already have an established developer base, making it easy for programmers to get started using the storage system. In addition, there is the consideration of backward compatibility: existing software, built to perform a certain task, may be able to be repurposed as a distributed, more scalable and more fault tolerant application. If the developer can avoid rewriting significant portions of software and focus instead on optimizing the performance of the application in the wide-area, getting his or her new application off the ground can proceed much more rapidly.

Previous work in wide-area distributed storage has used mainly object- or table-based models. PNUTS [18], CRAQ [74], Dynamo [24] and distributed hash tables [22, 87], for example, offer a read/write (or put/get) interface into a flat namespace of objects, while systems like Cassandra [15] offer row and column view of its data, separated into tables.

The wide-area storage system presented in this dissertation, on the other hand, instead offers a *file system* interface: a hierarchy of directories and files. This is a familiar interface supported by all operating systems and programming languages, and thus programmers can quickly develop new applications on top of it. Furthermore, a distributed storage system with a file-system interface helps systems designers to deploy existing software built to use a local file system in a distributed setting (a property previously discussed by the designers of TierStore [25]). Additionally, a wide-area file system is a useful tool for large-scale distributed testbeds: an often-lamented shortcoming of PlanetLab is the lack of common file system among the nodes [5], and the designers of a forthcoming testbed, GENI, have articulated a need for one as well [35].

Distributed file systems, of course, are nothing new. Decades of research have produced countless examples, from client/server systems like NFS [63] and AFS [64] to local-area decentralized systems like xFS [6], Farsite [1] and GFS [36]. These examples, however, are not suitable as scalable storage systems over the wide-area for reasons mentioned earlier: techniques that work well in a single site do not necessarily guarantee good performance in the wide area. As a result, different applications may have different preferences regarding the behavior of their data, and so no existing distributed file system is suitable for all applications. While object-based storage architectures offering wide-area tradeoffs, such as PADS [10] or CRAQ [74], can be used as the base layer of a mountable file system, their interfaces are not currently general enough to specify a complete set of tradeoffs at run-time, at the granularity of an individual file reference. By augmenting a file system interface with semantic cues, the storage system presented in this dissertation aims to bring fine-grained data control to applications.

1.4 WheelFS

The wide-area file system introduced in this dissertation, WheelFS, allows application control over the sharing/independence tradeoff, including consistency, failure handling, and replica placement. Central decisions in the design of WheelFS include 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 distributed file systems (*e.g.*, every file or directory has a primary storage node) [1, 12, 62, 76]. 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. WheelFS allows the cues to be expressed in the pathname, avoiding any change to the standard POSIX interface. Table 1.1 illustrates several examples of semantic cues in action.

A prototype of WheelFS runs on FreeBSD, Linux, and MacOS. The client exports a file system to local applications using FUSE [34]. 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 [72].

Scenario	WheelFS Path
Relaxed consistency: The application wants to limit network timeouts to one second when accessing files and subdirectories in the directory `dir`. If WheelFS cannot fetch the most up-to-date contents of those files and directories within the time limit, the application wants to read any out-of-date contents available instead.	`/wfs/`**.EventualConsistency/.MaxTime=1000**`/dir/`
Popular file: The application wants to read the entirety of the file `binary` on many different nodes at once.	`/wfs/`**.WholeFile/.HotSpot**`/binary`
Replica placement: The application wants all files and subdirectories created under the directory `user` to have their primary replica placed on a node at the site "closest_site".	`/wfs/`**.Site=closest_site/.KeepTogether**`/user/`

Table 1.1: Example scenarios in which to use WheelFS, and the full WheelFS paths, including semantic cues, appropriate for those situations. This table assumes WheelFS is mounted at `/wfs`. Semantic cues are in bold.

Measurements show that WheelFS offers more scalable performance on PlanetLab than an implementation of NFSv4 [55], 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 download times than BitTorrent [17] by using network coordinates to download from the caches of nearby clients.

A public deployment of WheelFS on PlanetLab allows researchers to share storage between nodes easily, and the source code for WheelFS is available at `http://pdos.csail.mit.edu/wheelfs`.

1.5 Contributions

The main contributions of this dissertation are as follows:

- The notion of semantic cues as a convenient, powerful way for applications to express the desired behavior of the underlying storage system. Semantic cues, as a concept, are useful in any wide-area distributed storage system. Furthermore, this dissertation identifies a specific set of cues that allows applications to control the file system's consistency and availability tradeoffs. Grouping a broad set of controls into a single, coherent mechanism, applied at a per-object granularity, has not been done before by previous storage systems.

- WheelFS, a new file system that assists in the construction of wide-area distributed applications, using semantic cues.

- A set of wide-area applications showing that semantic cues and WheelFS can be used to build wide-area applications easily. Applications that find WheelFS useful tend to have relatively small deployments (hundreds to tens of hundreds of nodes), use file and directories rather than databases, and do not require application-specific merging algorithms for divergent replicas.

- An evaluation demonstrating that wide-area applications can achieve good performance and failure behavior by using WheelFS, and perform competitively with applications built using special-purpose storage systems.

1.6 How to Read this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 outlines the goals of general wide-area storage layers, then introduces WheelFS and its overall design. Chapter 3 describes WheelFS's cues, and Chapter 4 presents WheelFS's detailed design. Chapter 5 illustrates some example applications, Chapter 6 describes the implementation of WheelFS, and Chapter 7 measures the performance of WheelFS and the applications. Chapter 8 discusses related work, and Chapter 9 concludes.

Chapter 2

Wide-Area Storage Overview

This chapter discusses several general properties of wide-area storage systems, and then presents more specific properties that two example distributed applications require of their storage systems. Following the examples is a broader discussion regarding the types of applications and environments for which WheelFS would be used, and an introduction to the basic system design of WheelFS, to help the reader follow the design proposed in subsequent chapters.

2.1 General Properties

A wide-area storage system must have a few key properties in order to be practical.

Usability. A storage system 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 (*e.g.*, replication, liveness monitoring, load balancing, etc.). The applications should be able to focus on higher-level issues of function and features, without getting bogged down in the details of data management. Many of the initial challenges of designing a distributed application come from understanding its data requirements, and an ideal storage system would not require a significant learning curve for a developer to move from an idea to a proof-of-concept prototype. From there, the storage system should allow the developer to test different storage tradeoffs quickly and easily, eventually transitioning the application to a fully-featured project optimized for use over wide-area networks.

Data sharing. A wide-area storage system must allow inter-site access to data when needed, as long as the health of the wide-area network allows. The ability for two nodes at different sites to access and update the same data is the crux of any cooperative distributed application. An application that partitions data (and the users of that data) statically between sites cannot easily add resources at new sites to scale further, rebalance uneven loads as data popularity changes, or hope to survive any site-wide outages. Many of the benefits of a distributed storage system comes from any node being able to read or write any piece of data.

Fault tolerance. When the site storing some data is not reachable, a 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. This is a crucial property of a storage system, as it allows the application to avoid the complexities of handling transient network and node failures. This frees the application to focus on higher-level issues, such as its requirements for data consistency in the presence of these failures.

Data placement. Finally, applications may need to control the site(s) at which data are stored in order to achieve fault-tolerance and performance goals. For example, the application might wish to replicate one category of its data at many different sites to ensure high availability for any node in the network, while specifying another category of data, more temporary in nature but more sensitive to update latency, that is only replicated within a single site. The number of replicas, and the number of sites these replicas are spread across, can be useful controls for the application, as they allow for a direct tradeoff between read performance, write performance, and fault tolerance. Furthermore, only the application knows how to make this tradeoff; a storage system cannot effectively make this tradeoff without application knowledge.

2.2 Examples of Wide-Area Applications

An application depends on the general properties of a storage system, but may also have specific requirements for its data in order to achieve acceptable performance across the wide area. Moreover, different applications might have different requirements. Next, this section discusses two examples of wide-area applications with specific storage requirements.

2.2.1 A distributed Web cache

A distributed Web cache is a wide-area application whose primary goal is to reduce the load on the origin servers of popular Web pages [32, 80]. Each participating node 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.

Because of these specific requirements, many existing storage systems do not fit the bill. As such, distributed Web caches such as CoralCDN and CoDeeN implement their own specialized storage systems that provide low delays in the presence of failures and the ability to find nearby copies of cached pages in order to spread the load of serving popular files across many nodes and improve performance.

2.2.2 A distributed mail service

A distributed mail service (consider, for example, a wide-area version of Porcupine [61]) aims to provide email for a global base of users. The mail service uses a distributed storage system to spread the load of storing, sending, and reading emails over nodes in many data centers. This geographic spread assures that users can still access their email during site-wide outages. Also, for performance reasons, users should be able to access their mail using nodes at a nearby site; under normal operation, it would be disastrous for users to connect to a server halfway around the world just to read their new messages.

The mail service requires that replicas of users' emails are placed in multiple sites, and more specifically it requires that at least one of those replicas is placed at an exact site: for a particular user, one replica of the user's emails should be placed at the site nearest to the user's usual geographic location. Data consistency requirements, as in the distributed Web cache example, are relaxed, since emails are generally write-once data items. A storage system flexible enough to handle all of these requirements would considerably ease the development of a distributed mail service.

Of course, 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 to distribute large binaries to many nodes simultaneously. 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.

2.3 WheelFS Overview

WheelFS, the new distributed file system presented in this dissertation, meets the general goals of wide-area storage systems, and provides the controls necessary for applications to tune the storage to their liking. This section first discusses under what circumstances WheelFS is useful, and briefly outlines WheelFS's system design.

2.3.1 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 [57]; it does not guard against Byzantine failures in participating servers [7,46]. 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.

2.3.2 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, and zero or more backup storage servers, that are 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, though the application can change this behavior using semantic cues. 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. To detect conflicting writes to objects that

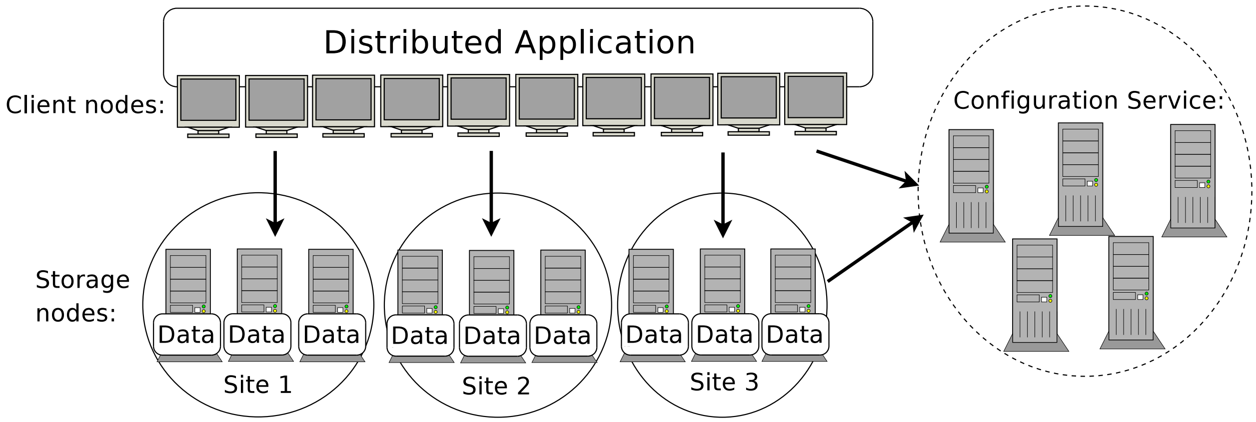

Figure 2-1: WheelFS's deployment architecture.

might occur during network partitions, 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 implies a mapping of each object to one primary and zero or more backup servers. WheelFS nodes cache a copy of this table. When a primary fails, one of the backup replicas for each of that primary's objects promotes itself to primary using the configuration service. Chapter 4 presents the design of the configuration service.

By default, 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.

Figure 2-1 shows the overall deployment paradigm for WheelFS. The nodes involved in running a distributed application each mount the same WheelFS instance at a local mountpoint, and become WheelFS clients. Storage servers (which can, but do not have to, run on the same physical computer as clients) store data and service requests from the client nodes. Another set of nodes acts as the configuration service, maintaining the membership of the system and the object primary assignments.

Chapter 3

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 [7] or a wide-area version of BigTable [16]. This section describes how applications specify cues and what effect they have on file system operations.

3.1 Specifying Cues

WheelFS aims to allow specifying semantic cues with minimal application involvement. This section first describes WheelFS's approach, and then enumerates and discusses several possible alternatives.

3.1.1 Cues in pathnames

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. A final disadvantage is that, if an application wanted to change some of the permanent properties bestowed on a file by cues at creation time (see Section 3.2), the application must explicitly copy the file to a new location, using new cues. We have not encountered examples of these problems in practice.

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. Cues are reserved keywords in the file namespace, and clients attempting to create a file with the same name as one of the cues will receive an error.

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 5 for a more sophisticated version of this application.

3.1.2 Alternative methods

One alternative to expressing cues in pathnames is to add an operating system call analogous to `fadvise()`, to inform WheelFS directly how the application plans to use a particular file handle. This would require operating system and FUSE support, but fits well into existing applications that already using `fadvise()` to control file system behavior. In comparison to pathname specification, which many times requires only changes to application configuration files, an `fadvise()`-like mechanism would require substantial code changes in the application.

Another method for specifying cues within the context of a mounted file system is to borrow an idea from the likes of CVS (`.cvsignore`) and `make` (`Makefile`), and have a special `.cues` file within every directory. `.cues` would list the cues associated with that directory and, if needed, for each individual file within the directory. This scheme has the advantage that permanent properties of a file or directory, such as the number of replicas, can be changed easily without having to copy the file to a new location. However, it would require extra intelligence in applications to read and write an additional file, and it would not allow for different references to the same file to have different cues.

A final alternative to expressing cues in pathnames is to change the way WheelFS is used altogether: instead of mounting it as part of a local file system using FUSE, WheelFS could be used as a library, and the application could specify cues as part of method calls into that library. This would necessitate writing special code within the application to make these library calls, and would probably exclude repurposing existing applications built for use on local file systems. However, this would be a reasonable interface choice for new applications built from scratch, and in fact the current WheelFS code could, with some effort, be refactored to allow for library access.

One key take-away from this discussion is that semantic cues are not tied to the specific pathname mechanism presented in this dissertation. Indeed, for some classes of applications, some of these alternative methods might be better fits. Semantic cues are a general strategy that can be implemented in a number of ways; WheelFS chooses to implement them as pathname components because of programmer familiarity and the fact that they enable easy reuse of existing software.

3.2 Categories

Table 3.1 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 wide-area storage properties discussed in Section 2.1, and the wide-area network challenges outlined in Section 1.2. 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 3.2 shows which POSIX file system API calls are affected by which of these cues.

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.

3.2.1 Persistence vs. transience

Each cue is either *persistent* or *transient*, as noted in Table 3.1. A persistent cue is permanently associated with the object, and affects 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

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. This cue can also affect consistency, since backups promoted to primary might not have the latest version of a file.
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. Fetch multiple blocks in parallel if used with **.WholeFile**.

Table 3.1: Semantic cues. A cue can be either **P**ersistent or **T**ransient (*Section 3.5 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 3.2: The POSIX file system API calls affected by each cue.

creating the object. Persistent cues are immutable after object creation. If the application uses a persistent cue when making a new reference to an existing file or directory, that cue is ignored and has no effect.

If an application specifies a transient cue in a pathname passed to an `open()` call, the cue only applies to file system operations that use the file descriptor returned by that `open()` call. Future operations (or concurrent operations on different clients) to the same file that do not include the cue in the file's path are not affected. In general, transient cues do not overlap with permanent cues, and thus the application does not need to worry about conflicts with cues specified when the object was created. The only exception to this rule is the **.EventualConsistency** cue, which is covered in more detail in Section 3.5.

Cues are persistent when they affect the location and maintenance of objects – properties that may need to be determined by background processes without the benefit of a given application file descriptor, or that need to apply recursively to future entries in a directory. For example, in order to maintain the correct number of replicas of an object, WheelFS's background maintenance process must be able to determine what that correct number is, and can only do so if that number is associated with the object in some permanent way. Transient cues, on the other hand, affect decisions that are independent of the background maintenance process, such as whether an application chooses to read an out-of-date replica or wait for a write to complete at all replicas.

3.2.2 Using cues in practice

This section discusses a number of issues involved with using semantic cues, such specifying multiple cues in a pathname, using different cues for the same file on different clients, and specifying a default set of cues for all data within an application.

As mentioned above, a cue specified in a path overrides all other cues governing the same property that appeared earlier in the path. For example, consider what happens when a user runs the following command from a shell, when the `/wfs/` directory is empty:

```
mkdir -p /wfs/.Site=westcoast/foo/.Site=eastcoast/bar
```

The primary for the directory `foo` will be placed on a node at the "westcoast" site, while the primary for directory `bar` will be placed on a node at the "eastcoast" site. Cues that govern different properties, even those within the same category, do not conflict and may appear in any combination within a path. Note that for each cue listed in Table 3.1 there is an opposite cue (not shown) that the application can use to turn off earlier cues. For example, **.WholeFile** can be overridden later in the path by **.NotWholeFile**.

In most cases, it is safe and correct for different clients to access the same file using different transient cues. For example, different clients might want to specify different timeouts when accessing the same file, because the workloads on the clients are different; this works as expected. However, if some clients use **.EventualConsistency** and others use WheelFS's default strict consistency to access the same file, unexpected behavior can occur. If one client writes to a file using **.EventualConsistency** and a failure occurs, that write could be applied at a backup replica, but not the primary. Another client accessing the file using strict consistency might then read an old version of the file from the primary, even though the first client has already closed its reference to the file. A similar situation can occur when one node writes a file using a **.SyncLevel** cue to write to fewer that N_{RL} replicas, and another node reads the file using strict consistency. Applications must carefully consider the implications of accessing the same file with different consistency cues.

Often an application will want a certain set of cues applied uniformly to all of its data. This is easy to achieve with WheelFS: if the application reads in a top-level directory from a configuration file or command-line parameter, that specified directory can simply include the desired cues. Then, as long as the application always accesses its data through the supplied top-level directory, all data will be affected by those cues. Similarly, if a user on a client wanted a set of cues consistently applied to data in his or her home directory, he or she could add those cues to the directory specified in `/etc/passwd`. Other alternatives, such as configuring a particular instance of WheelFS to have different default behavior, are possible but not as elegant.

3.3 Placement

Applications can reduce latency by storing data at a WheelFS storage server near the clients that 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 **.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 sub-tree 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$_{\boldsymbol{RS}}$** cue indicates how many different sites should have copies of the data. **N$_{\boldsymbol{RS}}$** only has an effect when it is less than the replication level (see Section 3.4), 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.

3.4 Durability

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

The **.RepLevel=N$_{\boldsymbol{RL}}$** cue causes the primary to store the object on $\mathrm{N}_{RL}-1$ backups; by default, $\mathrm{N}_{RL}=3$. The WheelFS prototype imposes a maximum of ten replicas (see Section 4.2 for the effects of this limit).

The **.SyncLevel=N$_{\boldsymbol{SL}}$** 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, $\mathrm{N}_{SL}=\mathrm{N}_{RL}$.

3.5 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. These situations can occur because a backup has not yet taken over the responsibilities of a failed primary (see Section 4.3), or because a network partition causes a client to be unable to contact a live primary.

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.*, [31]) used by systems like Bayou [75], 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 limits each of its communications to remote nodes to a given time limit, and uses the data with the highest version that it finds. (Section 4.4 discusses version numbers in more detail.) Clients consider a reply from the primary to be authoritative, and can use the primary's response without checking with or waiting for responses from the backups. The default time limit is ten seconds, 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, because if different updates to a file were written to different storage servers during a network partition, they may have the same version number. WheelFS does not support application-specific conflict resolution algorithms, and instead picks one version of the file to supersede all others completely.

For an eventually-consistent directory, the maintenance process 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. As noted above, this merging technique could cause deleted directory entries to reappear in the reconciled replica. 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.

3.6 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 multiple 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. Tread [86] is a mechanism for WheelFS that provides adaptive read-ahead behavior, though it still requires the application to specify the **.WholeFile** and **.Hotspot** cues in most cases. We leave fully-automated behavior during large-file reads to future work.

3.7 Discussion

This section discusses several remaining questions that the reader might have about semantic cues.

- **Can semantic cues be applied to non-file-system storage layers?** Semantic cues as a concept – a set of controls that applications can use to specify tradeoffs in the face of wide-area challenges – can indeed be applied to other storage systems. For example, a record-based, library-interfaced system such as PNUTS [18] could include options for setting semantic cues along with each call in its API, or provide a more general object handle class in which cues can be set internally and that can be passed to different functions instead of a flat object ID. In fact, PNUTS already does allow some controls to be set using its API, but only with regards to consistency; by supporting a wider range of controls in the form of semantic cues, a wider range of applications could use PNUTS in a disciplined way.

- **Would the specific set of cues detailed in this chapter still be useful in a different type of storage system?** Though the idea of semantic cues is applicable to many different types of wide-area storage systems, the specific set of cues offered might vary based on the system's data interface. For example, supporting a **.WholeFile** cue in a system that only manages objects of a small maximum size (like DHash [22]) does not make any sense. Storage system designers must carefully consider the tradeoffs that applications using their system would need to make in the wide-area, and include only cues that directly control those tradeoffs.

- **Are all semantic cues fully composable?** Developers may question whether all combinations of cues, used together during the same reference to a file, make sense. As discussed in Section 3.2.2, because different cues govern different tradeoffs and cues that appear later in pathnames override those that appear earlier, all combinations of cues do in fact make sense for a given operation on a given reference to a file (though some cue *parameter* combinations do not, such as setting $N_{SL} > N_{RL}$ – these can be caught at run-time). However, developers must take care when using different consistency cues for different references to the same file, and when writing to a file with a low **.SyncLevel** but reading it with strict consistency.

- **Do applications really need reference-granularity control?** Whether an application needs to apply different cues to the same file at different times is of course entirely dependent on the application, but there are conceivable uses for such an ability. For example, when a user updates her Facebook status message, subsequent page loads for that user must include the new status message; this implies that the storage layer must use strict consistency when accessing a user's own data. Other users do not need to see this status change reflected immediately, and can view that user's data using more relaxed consistency. Without extensive use of semantic cues by other developers, the usefulness of this ability is not fully clear, but it could prove to be a powerful control for certain applications.

- **Do all semantic cues need to be specified by the application?** To keep a storage system's interface simple, a system designer might hope to determine wide-area tradeoffs automatically, based on the behavior of the application. In general, for the cue categories discussed in this chapter, this is impossible. How can the storage system know whether an application wants the freshest copy of a particular piece of data, unless the application explicitly informs it? Some specific parameters can perhaps be chosen in an autonomous way, though often there is still a larger tradeoff that must be made by the application. For example, a storage system might be able to determine the number of replicas needed to a data item durable, but whether that data item *needs* to be durable at all is something only the application can answer. An exception to this generality, mentioned previously, is the large reads category of cues: that these need to be specified by the application is more an artifact of our WheelFS design and implementation than it is a fundamental property of the wide-area network.

Chapter 4

WheelFS Design

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

- How does WheelFS assign storage responsibility for data objects among participating servers? (Section 4.2)
- How does WheelFS ensure an application's desired level of durability for its data? (Section 4.3)
- 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 4.4)
- How does WheelFS permit peer-to-peer communication to take advantage of nearby cached data? (Section 4.5)
- How does WheelFS authenticate users and perform access control? (Section 4.6)

4.1 Components

A WheelFS deployment (see Figure 4-1) consists of clients and servers; a single host often plays both roles. The WheelFS client software uses FUSE [34] 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.

4.2 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.

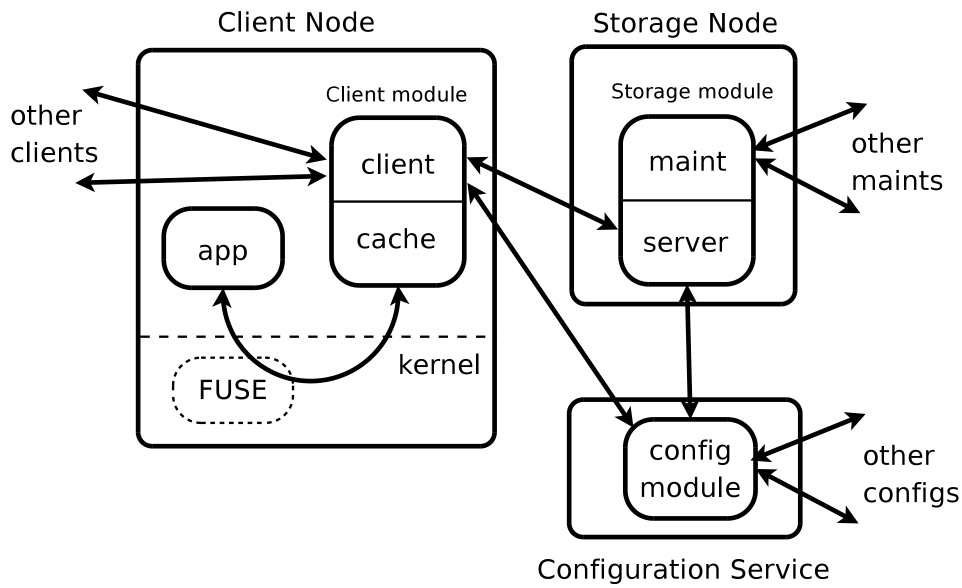

Figure 4-1: Placement and interaction of WheelFS components.

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 $\mathrm{N}_{RL}-1$ backups. The configuration associates replication policies with slices, rather than at the granularity of individual files or directories, so that it does not have to maintain state for each object in the system; it assumes that there will be many fewer unique replication policies than unique objects.

WheelFS uses a configuration service to determine the nodes responsible for a given object, rather than a local algorithm such as consistency hashing [42] as in many previous systems [21, 24, 72] for several reasons. The expected size of WheelFS deployments is hundreds to tens of hundreds of nodes, in contrast to the millions expected by other systems, which makes it possible to maintain the object-to-replica mapping on a small set of nodes. Furthermore, using an explicit replica list, rather than implicit replicas designated by an algorithm like consistent hashing, allows for richer replica placement policies, such as guaranteeing that the system spreads replicas for a given file across a given number of sites.

4.2.1 Slice size

The choice of S affects many different aspects of WheelFS: the amount of state managed by the configuration service, the number of WheelFS storage nodes that can participate in an instance of WheelFS, the number of unique replication policies that can users can specify for files, and how well data can be load-balanced across multiple storage servers.

The number of possible unique slices must be large enough to be able to assign one slice for each unique replication policy specified by the application, on each storage node in the system. For example, if the application used three different replication levels (**.RepLevel=**$\{$**1,2,3**$\}$) and three different replica-spread levels (**.RepSites=**$\{$**1,2,3**$\}$), and created new files with all sensible combinations of these parameters at each of 100 WheelFS storage nodes, it would use $6 * 100 = 600$

unique slice identifiers.

Our current implementation uses 64-bit object IDs, with $S = 20$. WheelFS's design targets deployments of up to a few thousand nodes. Even if a WheelFS instance comprises 4096 WheelFS storage nodes, using $S = 20$ allows for 256 unique replication policies, which would allow it to support more than enough combinations of **.RepLevel** and **.RepSites** cues to satisfy most conceivable applications.

4.2.2 Configuration service

The configuration service is a replicated state machine [66], and uses Paxos [45] to elect a new master whenever its membership changes. The master serializes updates to 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 [14]. 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 4.3 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.

Once a node has a complete cached copy of the slice table, subsequent updates can be fast: the configuration service need only send the changes in the table since the node's last fetch. Each version of the slice table has a *viewstamp*, consisting of the current *view number* of the replicated state machine (incremented each time the configuration service's membership changes) and a *sequence number* (incremented each time a slice is acquired or changed, and set to zero when the view number is incremented). On a fetch, the fetching node provides the viewstamp of its last fetch, and the configuration service sends only the updates that have happened since that viewstamp. The configuration service stores these changes as records in a log, checkpointing the log occasionally when the number of records gets large.

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 on a server 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 (*e.g.*, in order to choose a replica for a new object that uses the **random** placement policy) by taking the union of the replica lists of all slices.

Because WheelFS stores all objects within a slice on the same set of replicas, if a set of objects within a particular slice becomes too popular or too large, the primary associated with that slice may find it difficult to handle the load. WheelFS does not yet have a story for splitting slices across multiple nodes to load balance, though we expect that techniques developed for Farsite's directory service [27] might be useful. In Farsite, a node may delegate portions of its responsibility to other nodes when its load becomes excessive – it does this by informing the other node to take over a sub-portion of the namespace for which it is currently responsible. WheelFS may be able to adapt this idea using variable-length slice IDs, and allow other replicas in a slice to become the primary for a smaller portion of the slice's files by locking a slice ID with a longer prefix than the original slice ID. Further research into this method of load-balancing is future work.

4.2.3 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 and picking ones from sites that match the desired placement parameters for the object. When there are multiple choices for a site, or a node within a given site, the server chooses a site or node at random. 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.

4.2.4 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 [61]). Developing such a strategy is future work; for now, applications can use cues to control where data are stored.

4.3 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 $N_{SL}-1$ 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. If a server receives a request for an object for which it is not the primary (for example, if the client sending the request had an out-of-date slice table), the server rejects the request with a special error code, forcing the sending client to update its slice table.

In order to detect failure of a primary or backup, a server pings all other replicas of its slices every five minutes by default. 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 $N_{SL}<N_{RL}$ 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.

4.4 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()` or flushed client write (see Section 6.1.4) and after each change to the object's meta-data. A version number consists of a monotonically-increasing integer and the identity of the storage server that handled the update creating the new version. Upon creating a new version of a file, the storage server increments the file's version integer and sets the version identity to be itself. WheelFS totally orders versions of a particular file first by the version integer, and then by the version identity to break ties (since file versions created under **.EventualConsistency** can potentially have the same version integer, but only if the updates are handled by different servers).

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 3.5, 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).

4.5 Large Reads

WheelFS implements Tread [86], a wide-area pre-fetching system designed to improve file distribution. By default, Tread offers read-ahead pre-fetching, which reads the block located past the one the application has requested, in the hopes that the applications will want that block next. If the application explicitly 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. Tread also offers adaptive rate-limiting for the number outstanding pre-fetch requests in flight at any one time, in order to make the best use of scarce, wide-area resources.

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 [20]. 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 [7].

4.6 Security

WheelFS enforces three main security properties [57]. 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 possible 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.

Chapter 5

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 chapter evaluates how well WheelFS fulfills that goal by describing six applications that have been built using it.

5.1 All-Pairs-Pings

All-Pairs-Pings [70] monitors the network delays among a set of hosts. Figure 5-1 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 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.

5.2 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 cannot 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.

```
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 5-1: A shell script implementation of All-Pairs-Pings using WheelFS.

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 certain optimizations 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.

5.3 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**/**.RepSites=2**/*user*/**.KeepTogether**`/Mail/`

Each node runs a Dovecot IMAP server [28] to serve users their messages. A user retrieves mail via a nearby node using a locality-preserving DNS service [33].

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 the primary and all backups are down. The **.Site=X** cue indicates that a user's messages should be stored at site X, which the mail system administrator (or sufficiently-intelligent registration system) chooses 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 [61]. 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 soft state.

Wheemail has goals similar to those of Porcupine [61], 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.

5.4 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 [7] and CoBlitz [54]. 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. Furthermore, WheelFS uses network coordinates to find nearby replicas, while Shark uses clustering techniques. 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.

5.5 Distributed Compilation

The `distmake` [26] tool speeds up the compilation time of large programs by compiling on many machines. Currently, it assumes the use of a shared file system such as NFS or AFS, both to provide source files, headers, libraries, and tools as inputs, and to forward intermediate files from producers to consumers. Configuring `distmake` to use WheelFS instead of NFS offers two advantages: WheelFS can aggregate storage space from many nodes, and WheelFS offers availability through replication. We configure `distmake` to rely on the default close-to-open consistency for reading and writing source files. However, the build directory containing the object files can tolerate weaker consistency semantics. `distmake` operates by sending `make` commands to worker nodes. When a worker node executes a `make` command dependent on an object file, `make` first checks if that object exists and is up-to-date by comparing the object file's modification time with that of the source file's; if not, `make` automatically recompiles the dependency first. For this application, we assume all WheelFS nodes have synchronized their clocks (*e.g.*, using NTP).

The source directory uses a path without cues, such as:

`/wfs/src/`

The build directory uses relaxed consistency cues for object files, and keeps fewer replicas for performance reasons:

`/wfs/`**.EventualConsistency**`/`**.MaxTime=1000**`/`**.RepLevel=2**`/obj/`

The object directory uses **.RepLevel=2** to protect against temporary primary failures, and by also using the **.EventualConsistency**, the application can be sure that a backup replica will be read or written if the primary fails. This choice means, however, that an old version of an object file might be read later, either during another failure event or as the result of the merging of two divergent replicas. Missing objects or stale object modification times are perfectly acceptable in this application, as `make` will automatically recompile the relevant source files. Although `distmake` can be run over a collection of wide-area nodes, distributed compilation makes the most sense in a cluster environment.

5.6 Distributed BLAST

BLAST, a popular Grid application, finds gene and protein sequence alignments [4]. Most existing systems parallelize BLAST by partitioning the sequence database, but they require users to manually copy the database partition files to remote machines or to use a custom-built data transfer mechanism [44]. Our program, `parablast`, uses WheelFS to store sequence partitions and intermediate results and consists of 354 lines of Perl code. `parablast` starts BLAST on each node, specifying a batch of queries and a different partition file as input. The BLAST on each node writes its output to a unique filename in WheelFS. When a node finishes its partition, `parablast` assigns it another. Once there are fewer partitions left than compute nodes, `parablast` runs the remaining partitions on multiple nodes in order to avoid slow nodes holding up completion. The final phase of `parablast` merges the results from each partition by reading the result files from WheelFS. `parablast` stores database and query files (which do not change) in the following directory:

`/wfs/`**.WholeFile**`/`**.EventualConsistency**`/input/`

.EventualConsistency allows the compute nodes to read from cached copies of these files even when their responsible nodes are temporarily unreachable, since the input files are stored under unique names and will never be updated. **.WholeFile** enables pre-fetching, since compute nodes read the entirety of an input file for a job. Similarly, result files are stored with unique names in the following directory:

`/wfs/`**.EventualConsistency**`/output/`

.EventualConsistency assures that writes will always succeed despite transient unavailability of remote nodes. In contrast, the BLAST binary program is stored with the default close-to-open semantics so that all computations are guaranteed to have used the same up-to-date BLAST binary.

5.7 Discussion

WheelFS is a useful wide-area storage system for many distributed applications, but some applications might be better served by other systems. The following list of properties describes an application that might find WheelFS useful.

- **It uses files and directories.** This is an obvious point, but eliminates a large class of applications that rely on flat namespaces or database-like transactions over structured data items. In some cases, it may be possible to restructure the application to use a hierarchical file system.

- **It will be deployed on up to a few thousand nodes, of relatively infrequent churn.** WheelFS was not designed to scale up to millions of nodes, and it was not designed to run on home computers running behind NATs or unreliable networks. The configuration service acts as a point of centralization for the system, and all nodes must contact it to acquire and renew slices, and to fetch the slice table – this is not a highly scalable design. WheelFS could potentially be augmented using techniques for limited-size routing tables [69, 87] or namespace delegation [27], but the design described in this dissertation is intended to be used in relatively small, stable deployments.

- **It does not require application-specific merging.** Section 3.5 described WheelFS's merging policy for inconsistent files and directories. If an application requires a smarter policy for recovering data in the face of inconsistent replicas, they will not be able to use WheelFS in its current form. In many cases, an application can be restructured to take advantage of WheelFS's specialized directory-merging solution. For example, the distributed mail application above stores each email in its own file to take advantage of WheelFS's eventually-consistent directories. Specifically, an application will find WheelFS's merging policies most useful when each file has a single writer, and either each file is written only once or the whole file is re-written with each update and the application can tolerate the loss of some file versions. WheelFS does not currently support general, application-supplied merging techniques.

- **It only needs to control the tradeoffs covered by the semantic cues.** WheelFS does not allow applications to define their own arbitrary semantic cues, to avoid the the complexities involved in running application-specified code within WheelFS at arbitrary points. WheelFS's cues are designed to meet a large fraction of the needs of wide-area applications, but do not cover every possible case. Applications with specialized needs might considering using a storage framework such as PADS [10].

Though WheelFS does make it easy for developers to prototype new wide-area applications quickly, developers will need a good understanding of the full interface and design of WheelFS to achieve good performance for their applications. This is the case with any generic storage system, and WheelFS is no different – so much of the performance of an application is dependent on the storage system that it is difficult to disentangle them from each other cleanly. For WheelFS in particular, a developer may need to re-think the way an application structures its on-disk data so that it will work well with WheelFS. For example, consider the distributed mail system from Section 5.3. In order to take advantage of WheelFS's simple directory merging policies under **.EventualConsistency**, Wheemail uses the Maildir format for storing each email in its own file, rather than storing entire folders of emails in the same file, as is more standard. Choosing the latter strategy would mean that a network partition, and a subsequent reconciliation, could cause some emails to be lost: an unacceptable outcome for a mail system.

Thus, to achieve a production-level application with correct semantics and ideal performance, an application developer may have to become an expert in using WheelFS, though in this regard WheelFS is no different from any other wide-area storage system the developer might choose.

Chapter 6

Implementation

This chapter discusses the software implementation of WheelFS, as a guide to the open source code published at `http://pdos.csail.mit.edu/wheelfs` under an MIT license. Application developers and future WheelFS developers can use this chapter as documentation to the 0.1-1 release of WheelFS.

6.1 Software Components

The WheelFS prototype consists of over 22,000 lines of C++ code, using pthreads and STL. In addition, the implementation uses a new RPC library (4,600 lines) that implements Vivaldi network coordinates [20]. Furthermore, in order to support secure connections between nodes, the prototype includes a modified version of `libssh` [47], including more than 1200 lines of added or modified C code when compared to `libssh` Subversion revision number 198 [48].

The software comprises four major components: RPC, configuration service, storage module, and client module. These modules share a utility library, defining many of the classes, constants, and cues shared between the major components. An additional security layer, which uses `libssh` to establish secure connections between nodes, can be enabled at run-time. Figure 6-1 depicts these components, and the main modules acting within each component. Table 6.1 summarizes the main data structures maintained by each of the components.

The current release uses 64-bit object identifiers, 20 bits of which are reserved for a slice identifier (S=20); the evaluation presented in Chapter 7, however, uses 32-bit identifiers and $S = 12$, though results show no significant difference in performance. WheelFS reads data in 64 KB blocks to provide some data batching in the presence of high, wide-area latencies, so that each operating system read request does not result in a separate RPC. If the application plans to read the entire file, it can use the **.WholeFile** cue to enable pre-fetching.

6.1.1 RPC

WheelFS uses a custom RPC library for a variety of reasons. To inter-operate smoothly with the rest of the code, the RPC library must be multi-threaded and export a C++ interface. It must also allow for fine-grained (to the millisecond) control over timeouts, in order to support the **.MaxTime** cue. Finally, as described in Section 6.1.5, the RPC library must support public/private key pair authentication, preferably in the style of SSH to work well on PlanetLab.

The WheelFS RPC library provides at-most-once RPC calls [13], meaning that a server will never execute the same RPC more than once for a particular client. When a client initiates a con-

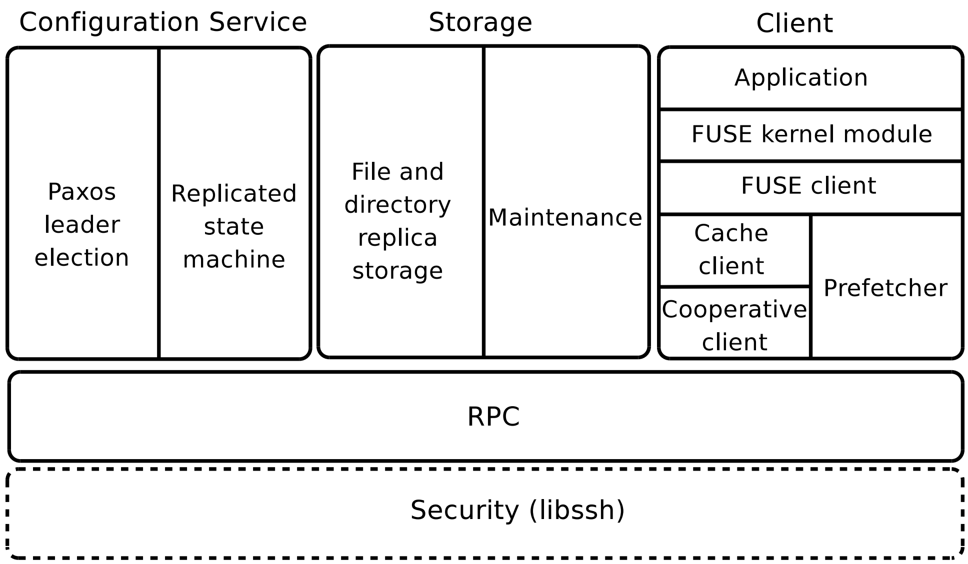

Figure 6-1: WheelFS's software architecture. The configuration service, storage, and client components use the RPC component (and optional security component) to communicate. The components in the top layer can, but do not have to, run on the same computer.

nection, it chooses a unique nonce for each server with which it communicates, and associates that nonce and a monotonically-increasing sequence number with each call to the server. For each nonce/sequence number pair the server executes the RPC only once, storing the computed result in a local cache until acknowledged by the client in a future request.

For each connection to a server, the client creates a TCP connection, an output thread for sending requests, and an input thread for receiving replies. The server creates input and output threads for each client as well, and launches a new thread for processing each RPC it receives. To control the total number of threads and file descriptors in use on both the clients and the servers, WheelFS enforces a limit on the number of outstanding connections; when that limit is exceeded, it closes old connections using a least-recently-used algorithm. Furthermore, the server limits the number of RPCs it will process in parallel, to avoid an explosion in threads; RPCs beyond that limit wait in a queue.

Making an RPC is simple: one of the higher-level components (configuration service, storage, or client) obtains an RPC client (`rpcc`) for the desired node. A utility class manages `rpcc`s, ensuring that only one exists for each server at a time. The component then executes the `rpcc`'s `call()` method, providing a unique procedure number, a list of argument parameters, a reference to a reply parameter that will be filled in after the call, and a timeout. Each type of RPC argument must have marshalling and unmarshalling methods associated with it, though the RPC library provides these methods for most basic types. The RPC system executes the call on the server by invoking a method registered under the procedure number, and then sending the reply argument and an error code back to the client. When the client receives the reply, or when the timeout passes, the `call()` method returns the error code and, if successful, the filled-in reply argument.

6.1.2 Configuration service

Section 4.2 describes the high-level design of the configuration service. The nodes running the configuration service use the Paxos [45] protocol to elect a leader among themselves. That leader

Component	Structure Name	Function
RPC	Connections	A collection of established TCP channels to other WheelFS nodes.
	Reply cache	A reply cache on the server side for unacknowledged RPCs.
	Handlers	A table on the server side mapping procedure number to local processing function.
Configuration service	Slice table	A table mapping a WheelFS slice to the slice's replication policy and replica list.
	RSM member list	The other members of the configuration service, including which is currently the master.
	Request log	An append-only log of requests executed by the replicated state machine. Periodically checkpointed.
	Keys	A list of the public keys for all authorized configuration service nodes, storage servers, and users.
Storage	Data	A local directory containing two files (a meta-data file and a contents file) for each object for which the node is a replica.
	Cache locations	A table mapping file object IDs to a list of clients that may have blocks of that file cached.
	Leases	A table mapping object IDs to a list of clients currently leasing the object, and the expiration time of those leases.
Client	Inumbers	A table mapping FUSE inumbers to WheelFS file handles.
	Cache	A table containing copies of file contents, file and directory meta-data, and positive and negative directory entries, keyed by version number. Can be in-memory or on-disk.
	Leases	A table mapping storage server identities and object IDs to the expiration time of the client's lease for the object/storage server combination.

Table 6.1: WheelFS data structures.

acts as the master of a replicated state machine, used by WheelFS nodes to manage their slice table. The configuration nodes periodically ping all live members of the service. As soon as one node discovers another is unreachable, it calls for a new leader election. During a leader election, configuration nodes will refuse to serve client requests (queueing or rejecting them). If the current master cannot reach the rest of the configuration nodes and they elect a new master, it will correctly refuse client requests, because it will be unable to determine a master without a majority in its own partition.

WheelFS nodes know of one member of the configuration service initially, and contact that member to learn about the other members, including which member is the master. Once the master is known, the WheelFS nodes make `acquire`, `renew`, `release`, and `fetch` calls to the master. If the master ever becomes unreachable, or responds with an error code indicating it is no longer the master, the WheelFS nodes try directly contacting the other members of the configuration service from a cached list until they discover the new master.

The master handles two kinds of requests: update requests and read-only requests. For update requests from a WheelFS node, the master increments the sequence number of the viewstamp and forwards the request and its viewstamp to all other configuration nodes in parallel. These nodes apply the update to their local version of the slice table, append the request to a log, and respond to the master. Once the master receives successful replies from all other members, it executes the request locally, appends the request to its log, and replies to the request's originator. Read-only requests are handled exclusively by the master, without requiring communication with other configuration service members. Such requests do not increment the sequence number in the replicated state machine's viewstamp, and so do not require any updates to the WheelFS nodes' cached slice tables.

`acquire` and `release` calls are update requests. A `renew` call is an update request only if the renewing node changes something about the slice table, such as the replica list or lock period. Thus, in the normal case, `renew` requests are not forwarded to the other configuration service members, in order to reduce latency for this frequent operation. This choice means, however, that if the master fails and a different member is elected master, that new member will not know which slices in the table have been recently renewed. As a conservative measure, the new master automatically renews all slices in its table for one lock period. This does not affect correctness of the system, since this process will not change lock owners, but only renew locks for slices that might already be free.

`fetch` requests are read-only. The master, in its copy of the slice table, keeps the viewstamp of the last time that entry has been modified. The node making the `fetch` request supplies the replicated state machine's viewstamp from its last `fetch`, and the master returns only the entries that have changed, or expired, since that viewstamp, along with the current viewstamp. Using its cached copy of the slice table, and the fact that WheelFS storage servers will always try to keep one slice locked at all times, WheelFS nodes can construct the entire membership list of WheelFS storage servers.

6.1.3 Storage

The storage component is responsible for managing data in WheelFS, and providing access to that data to WheelFS clients. Each storage server accepts RPCs both from WheelFS clients, and from other storage servers, via their maintenance modules.

WheelFS clients use cached copies of the slice table to determine which storage server is currently the primary for a given object, sending RPC requests directly to that server. The server executes each RPC using a local storage component, and takes out a lock on each object involved in the satisfying the RPC, to ensure that updates to each object are serialized.

The storage component manages WheelFS's on-disk copies of application data. Currently, WheelFS uses a single directory in the local file system to store all of the objects for which the local node is a primary or replica. It stores each object in a file named by the object's ID. A separate file stores the meta-data associated with this object, including the object's permanent semantic cues, and access control list. The interface to the storage component is general enough to support different storage schemes: future implementations might use more efficient on-disk data structures to store each object, rather than keeping two files per object in one large directory.

The maintenance module running on each storage server periodically communicates with the other nodes associated with each slice for which the server is a primary or a replica. It exchanges a summary list of the objects in slices that the nodes have in common, including the version number, meta-data information, and checksums for any directories, in order to efficiently decide when to merge inconsistent directories. When the storage server notes that one of its local objects is missing or out-of-date, it requests a new copy of the object from the other replica. If a backup replica is unable to reach the primary for one of its slices, it attempts to acquire the lock for that slice. If a primary is unable to reach a backup replica for one of its slices, it chooses a new replica, sends it the slice's data, and renews the slice with an updated replica list.

Finally, the storage server must maintain a set of leases for each object, and revoke all leases for an object before that object is updated. Whenever a WheelFS client node accesses an object, the server implicitly gives a lease for that object to the node, saving the node's IP address and port in an in-memory lease data structure associated with the object. When the object changes, the storage server contacts each node in the lease data structure to invalidate the lease. If any of the nodes are unreachable, it must wait for that node's lease to expire before updating the object. Clients maintain leases on a *per-primary* basis, so if the primary for an object changes, clients automatically invalidate their own leases; this design means that the storage server's lease data structures are only soft state, and need not to be replicated. Clients use their cached slice tables to determine when a given primary has lost its status; however, an explicit `release` by a primary (which happens only in the rare circumstance that a current primary wants to transfer responsibility to an older, revived primary for a given slice) could cause small periods of lease inconsistency.

6.1.4 Client

The client component consists of several sub-components: a FUSE client, a cache client, a cooperative client, and a pre-fetcher.

FUSE client. The FUSE client receives application requests from the operating system's VFS layer using FUSE's "low level" interface. For each object requested by the application, the client software assigns an opaque *inumber*. In practice, this inumber is a memory address pointing to a WheelFS *file handle*, which includes the WheelFS object ID and set of semantic cues associated with the reference. A single object accessed through different paths (*i.e.*, using different sets of semantic cues) have different inumbers, mapping to file handles with the same WheelFS object ID but different sets of semantic cues. The FUSE client passes requests to the cache client after translating FUSE inumbers into WheelFS file handles.

Cache client. The cache client executes each request using information cached locally only, whenever possible. It keeps an in-memory cache of all object meta-data, positive and negative directory entries, and file data, spilling out to disk when the in-memory cache fills up. It also maintains a per-primary lease for each cached item, which eventually either expires or is invalidated by the storage server when the item changes. If the request requires an object that does not have a valid lease, the cache client forwards the request to the object's primary storage server, and saves the response in the cache. The cache client proactively renews leases for objects that it has used recently.

The cache client also buffers writes for objects in memory, until either the application explicitly `fsync()`s or `close()`s the object, or until the size of the buffered writes grows too large. At that point, the set of buffered writes is sent to the object's primary storage server.

All requests requiring communication to a storage server are governed by the semantic cues set in the object's file handle. Under strict consistency, the cache client must communicate only with the object's primary, as determined by the client's cached slice table. Under eventual consistency, however, if the cache client cannot contact the primary, it may contact backup replicas; if the backup replicas are also unavailable, the cache client may use cached objects, even ones without valid leases.

Cooperative client. The cooperative client fetches file blocks from other clients' caches, whenever the **.HotSpot** cue is present in a file handle. The client first contacts the file's primary storage server, asking for a list of clients that have previously cached a given block. The server uses the Vivaldi coordinates of the node to determine a small set of nearby clients that might have the data. Among these choices, the client picks one from which to request the block: in general, it chooses the one with the nearest coordinates, but it ensures that a particular node is not sent too many request simultaneously. As it fetches blocks from other clients' caches, it also learns cache locations for other blocks of the same file, thus avoiding future requests to the object's primary storage server whenever possible. The cooperative client has a thread that informs the object's primary whenever it successfully caches a block, so that future **.HotSpot** reads from other clients can find the cached block in this client's cache.

Pre-fetcher. The pre-fetcher performs read-ahead and **.WholeFile** requests using a fixed number of pre-fetching threads. Its implementation is detailed elsewhere [86].

6.1.5 Security

With security enabled, the WheelFS RPC system sends its data over SSH channels running on TCP, rather than over plain TCP connections. To establish an SSH connection between a client and a server requires each node to have a public/private key pair, and to have the public keys available to all nodes. WheelFS accomplishes this by distributing public keys via the configuration service, along with the slice table. To add a new node to the system, an administrator must place its public key in an `authorized_hosts` file on the configuration service master; the master periodically checks this file for updates, and when the file changes, the master replicates the new file to all other configuration service members, and assigns a new viewstamp for the operation. When WheelFS nodes fetch future updates to the slice table, they will also get changes to `authorized_hosts`. Nodes must obtain the public key for at least one authorized configuration service member out of band. This scheme ensures that only authorized storage servers and configuration service members can communicate, making it impossible for un-authorized nodes to become the primary for any slices.

WheelFS clients must access data on behalf of an authorized user. Similar to authorized storage servers, there is a public/private key pair associated with each user. The configuration service master maintains a directory of authorized user keys, where the public key of each user is stored in a file named by the user's name. This directory of authorized users gets distributed along with the slice table. When a storage server or client handles an RPC request, it receives along with the arguments of the request the name of the user associated with the client that made the request; this user was authenticated by the RPC library, using the user's private key supplied by the client. Nodes can use this user information to enforce the access control lists associated with each object.

Xavid Pretzer's thesis [57] details WheelFS's security implementation.

6.2 Deployment

The first major WheelFS deployment is on the Planetlab testbed [8]. This deployment is meant to be a service to PlanetLab developers: authorized users can run a WheelFS client to gain access to this running deployment, and use the resulting mounted directory to store and share application data. Ideally, this deployment would serve as a proper distributed file system for PlanetLab, eliminating the need for developers to write their own custom application storage layers. It would also serve as a distribution means for application binaries and configuration files.

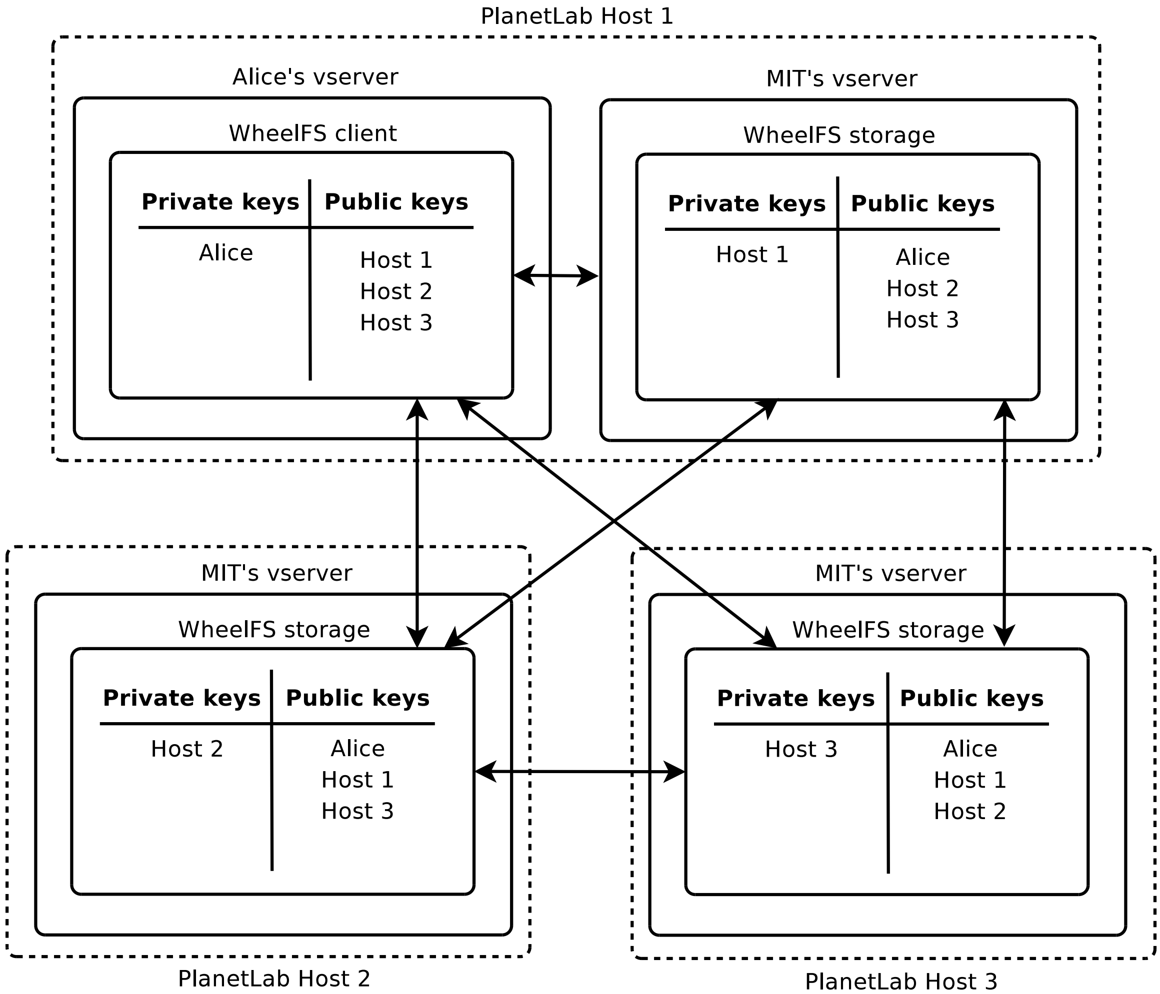

Figure 6-2: WheelFS's PlanetLab deployment architecture. MIT runs WheelFS storage servers within its vserver on various nodes, and other PlanetLab users (such as Alice) can connect to those storage servers to mount WheelFS within their own vserver. Each node must know its own private key and the public keys of the other nodes and authorized users. The configuration service is not pictured.

6.2.1 Architecture

On PlanetLab, users are identified by their *slice names* – this is a separate concept from WheelFS slices, and to avoid confusion this section will refer to PlanetLab slices as simply *users*. Users are isolated from each other in separate *vservers* (PlanetLab's method of accounting for resource usage and providing privacy on a physical node shared by many users). The storage servers for WheelFS's PlanetLab deployment run in the context of MIT's user, and other users can mount the deployment via SSH connections to processes running in MIT's vserver. Each user must register a public key with a WheelFS administrator, which is distributed to all storage servers as discussed in Section 6.1.5. The user must store the private key within its vserver on every PlanetLab node from which it wants to mount WheelFS. Storage servers enforce data access controls after authenticating a user's SSH connection, making the single deployment safe for use by multiple users, provided the users trust the MIT user and the PlanetLab node administrators.

Figure 6-2 illustrates a small PlanetLab deployment, where a PlanetLab user Alice mounts a

WheelFS instance which has storage servers running in MIT's vserver. Alice's data will be stored in MIT's vserver, and if Alice sets her file permissions such that only she can read her data, the storage servers will only allow clients knowing Alice's private key to access her data.

6.2.2 Expected bottlenecks

PlanetLab nodes are in constant use by researchers, and as such the disks and networks of the nodes are often under heavy load. The current WheelFS implementation has not yet supported a large amount of data over a long period of time under such conditions, and we expect that if many researchers begin using WheelFS on PlanetLab, portions of the WheelFS implementation may have to be re-engineered to cope with PlanetLab's load.

In particular, the maintenance module in the current implementation uses a simple algorithm when exchanging object information between two storage nodes that are replicas for the same slice. Because WheelFS stores meta-data information as separate files in a local file system, named by the file ID of the corresponding object, constructing the necessary information for an exchange requires scanning an entire directory and reading a potentially large number of small meta-data files. Furthermore, exchanging directory checksums requires reading each directory's data from disk and calculating an MD5 checksum. As the number of data items grows, the amount of work done by the maintenance module may grow too large to complete a maintenance pass in a timely matter. PlanetLab's slow disks compound this trouble: seeks can take seconds, and we have seen `fsync()`s of small files take over ten seconds in practice.

Passing Tone [68] is a maintenance system for distributed hash tables that suggests some alternative storage designs for WheelFS. Instead of constructing an entire list of meta-data information for each slice, WheelFS could use a per-slice Merkle tree [52] to optimize for the common case in which the slice replicas are already synchronized. As Passing Tone demonstrates, however, building such Merkle trees requires careful data layout. The maintenance module must be able to scan meta-data for all the objects in a given slice quickly, suggesting that meta-data should not be stored in per-object files, but instead in per-slice files or in a local database.

Passing Tone also suggests that object data be grouped together in a single append-only file by expiration date. Because WheelFS is a file system, it does not expire data and relies on applications to unlink files when they are no longer needed; thus, this particular optimization is not useful for WheelFS. Grouping files together by slice ID may allow for more efficient replica exchanges between storage nodes, but any such scheme must balance the need for efficient maintenance with the need for random update access to files (in Passing Tone, objects are immutable).

Network capacity may also have a large effect on WheelFS's ability to maintain replicas durably, as PlanetLab users are often limited to only 150 KB/s of bandwidth available on each node. Depending on the future failure rate of PlanetLab nodes and the amount of data stored by applications, WheelFS may have trouble synchronizing an entire slice of data to a new replica before the slice experiences another failure. Though this may turn out to be a fundamental limitation to the amount of data our WheelFS deployment can support reliably, data compression techniques could lessen the maintenance module's network requirements (at the expense of CPU).

We plan to monitor WheelFS's bottlenecks closely as researchers begin using our deployment, and investigate these alternative implementation choices as bottlenecks become apparent.

Chapter 7

Evaluation

This chapter 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.
- WheelFS offers an interface which is as expressive as existing wide-area storage systems.

7.1 Experimental Method and Setup

To answer the questions above, we ran several experiments using our WheelFS prototype in different scenarios. 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. Results not shown in this thesis demonstrate that using 20-bit slice IDs and 64-bit object IDs does not significantly impact performance.

Communication takes place over plain TCP, not SSH, connections. Using SSH connections reduces performance by a factor of two [57], and this chapter focuses on evaluating WheelFS's performance independent of its security layer. 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 [8] and Emulab [83]. For PlanetLab experiments, we use up to 250 nodes geographically spread across the world at more

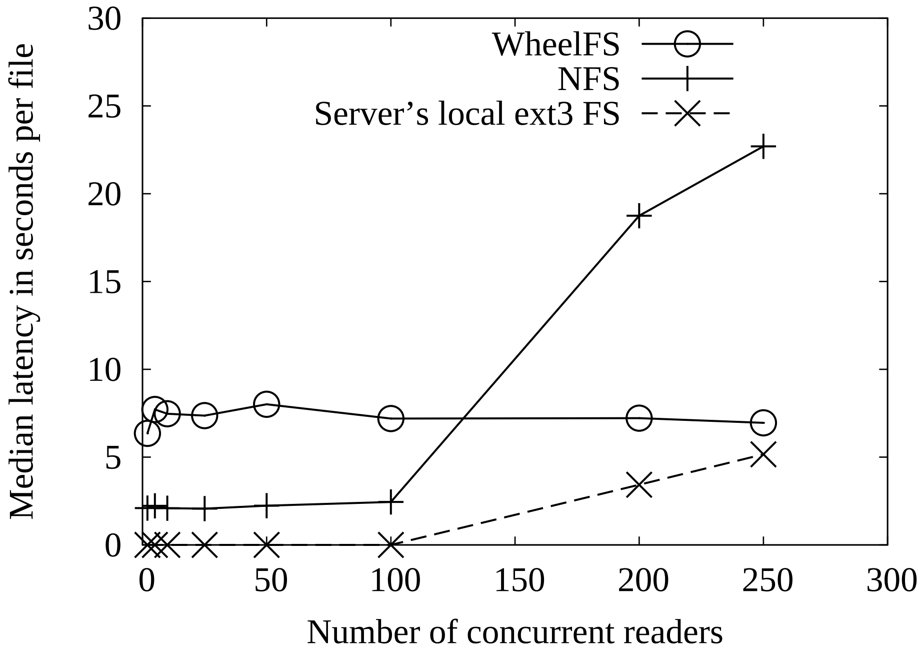

Figure 7-1: The median time for a set of PlanetLab clients to read a 1 MB file, as a function of the number of concurrently reading nodes. Also plots the median time for a set of local processes to read 1 MB files from the NFS server's local disk through `ext3`.

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 [83] 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.

7.2 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 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 7-1 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 7-1 also shows results for N concurrent processes on the NFS server, accessing the `ext3` file system directly instead of through NFS, showing a similar latency increase after 100 clients (but without any overhead from network latency or NFS timeouts). WheelFS latencies are not affected by the number of concurrent clients, since WheelFS spreads files and thus the load across many servers.

Comparing WheelFS to NFS is slightly unfair, as the centralized-server design of NFS does not allow it take advantage of resources spread across many nodes. A fairer comparison would be against an open-source distributed file system such as HDFS [41]; however, HDFS use a single meta-data server, and thus might also exhibit scalability limitations in comparison to WheelFS. Comparing WheelFS to other distributed, cluster-based file systems is an area of future work.

7.3 Distributed Web Cache

These experiments compare the performance of CoralCDN and the WheelFS distributed Web cache (as described in Section 5.2, 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.

7.3.1 Performance under normal conditions

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 [43] 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 [32], models a flash crowd where a set of files on an under-provisioned server become popular very quickly.

Figures 7-2, 7-3 and 7-4 show the results of these experiments. Figure 7-2 plots the total rate at which the proxies serve Web client requests (the y-axis is a log scale), while Figure 7-3 plots the total rate at which the proxies send requests to the origin server. WheelFS takes about twice as much time as CoralCDN to reduce the origin load to zero; both reach similar sustained aggregate Web client service rates. Figure 7-4 plots the cumulative distribution function (CDF) of the request latencies seen by the Web clients. WheelFS has somewhat higher latencies than CoralCDN.

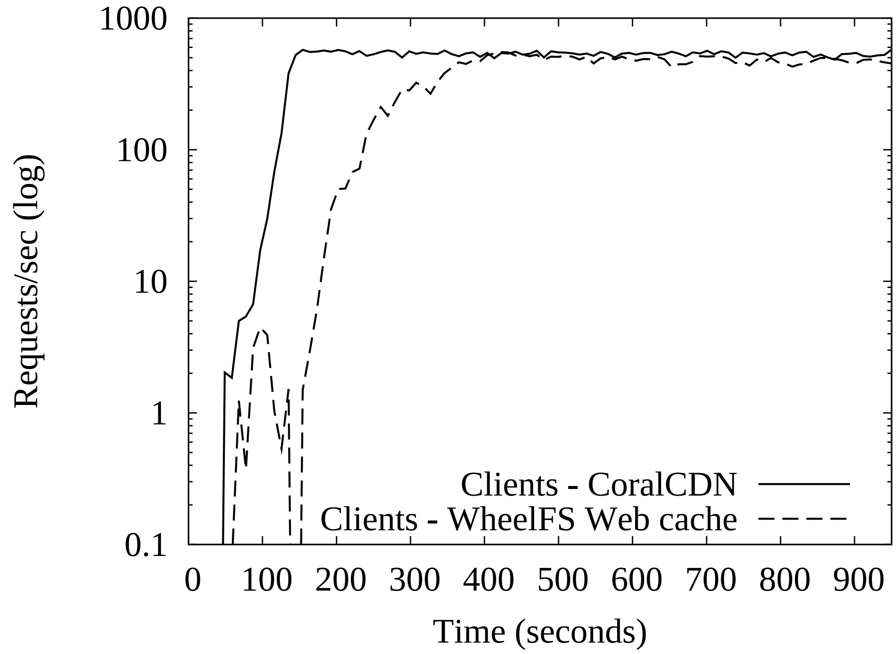

Figure 7-2: The aggregate client service rate for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

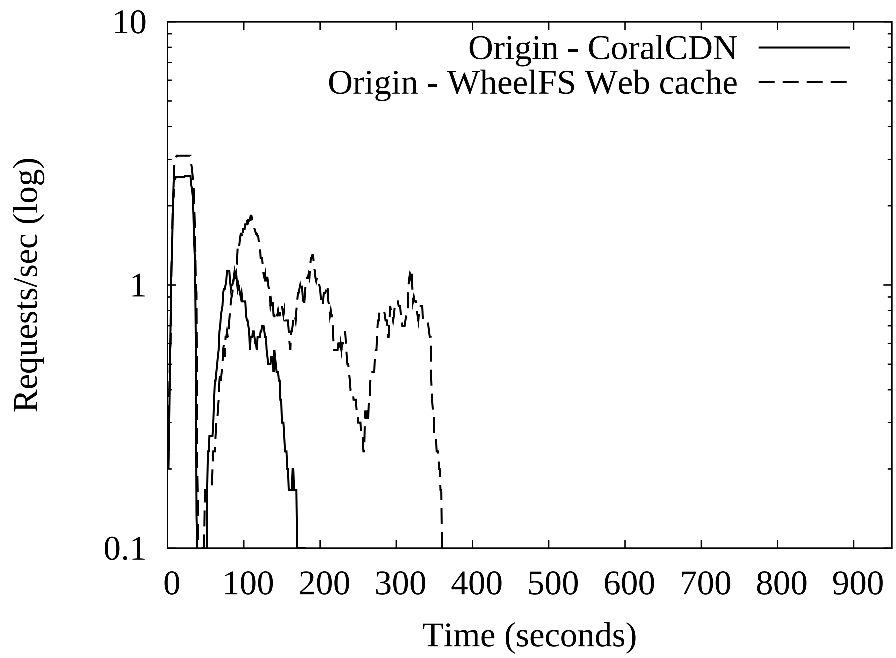

Figure 7-3: The origin server load for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

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

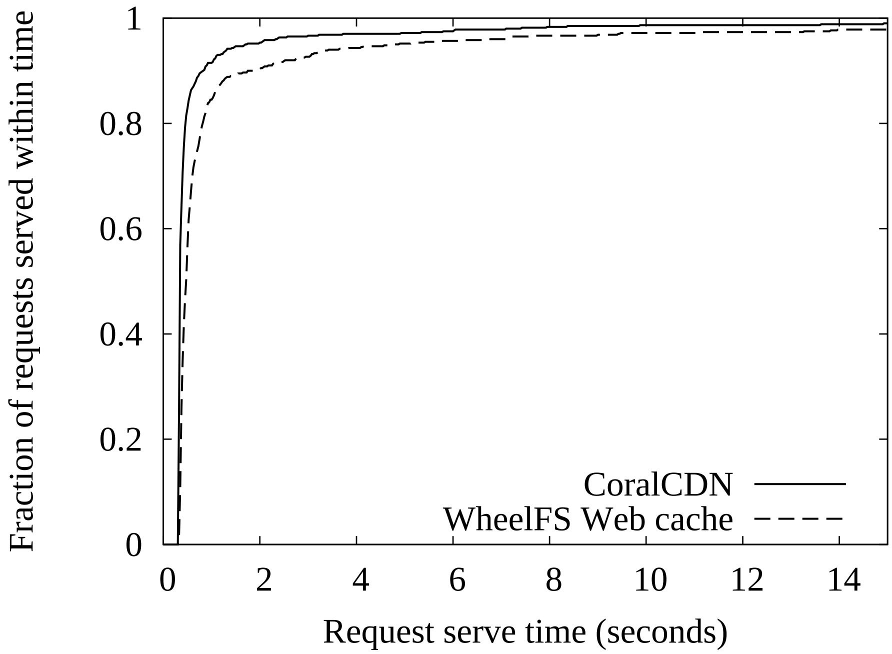

Figure 7-4: The CDF for the client request latencies of both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

nodes may download the same page before Apache caches the data in WheelFS. Similar optimizations could be implemented in Apache.

7.3.2 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 5.2, 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 7.1, 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 7-5 and 7-6 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 7-7 shows WheelFS's performance on this topology when there are no 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

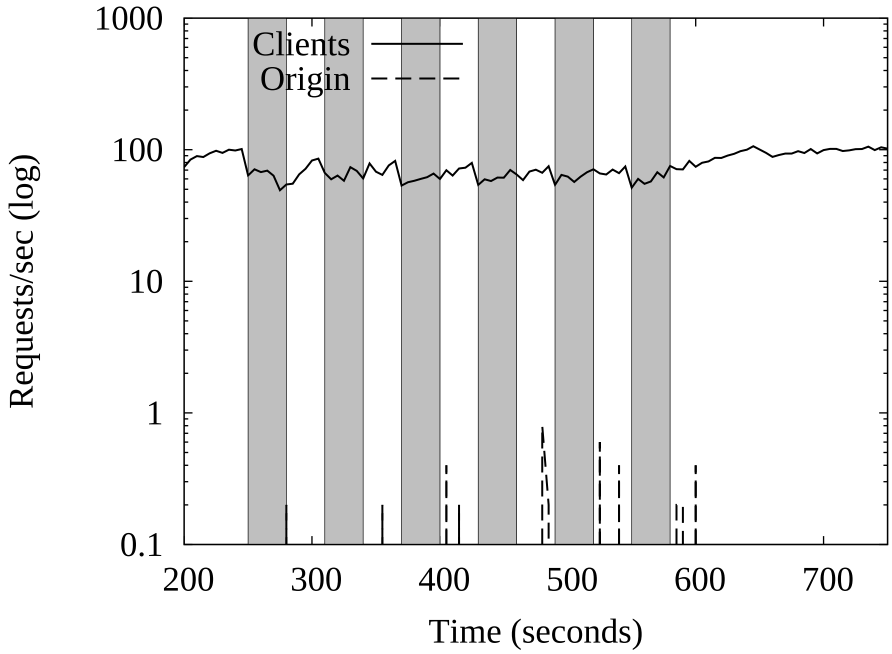

Figure 7-5: The WheelFS-based Web cache running on Emulab with failures, using the **.EventualConsistency** cue. Gray regions indicate the duration of a failure.

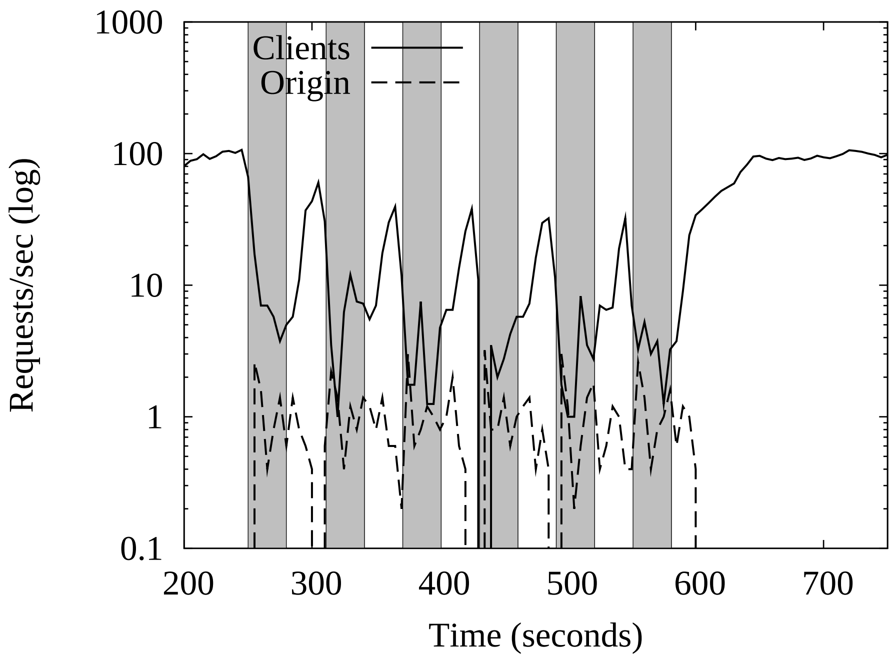

Figure 7-6: The WheelFS-based Web cache running on Emulab with failures, with close-to-open consistency. Gray regions indicate the duration of a failure.

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 client need not contact a server. If the

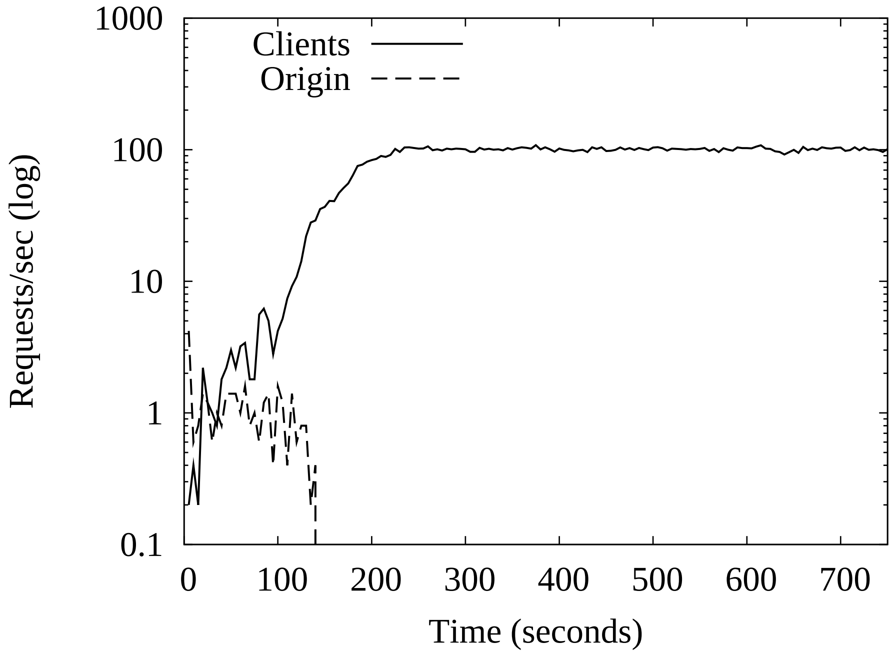

Figure 7-7: The aggregate client service rate and origin server load for the WheelFS-based Web cache, running on Emulab, without failures.

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 a timeout), the proxy must fetch the page from the origin Web server.

Figure 7-5 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 7-6 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.

7.4 Mail

The Wheemail system described in Section 5.3 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

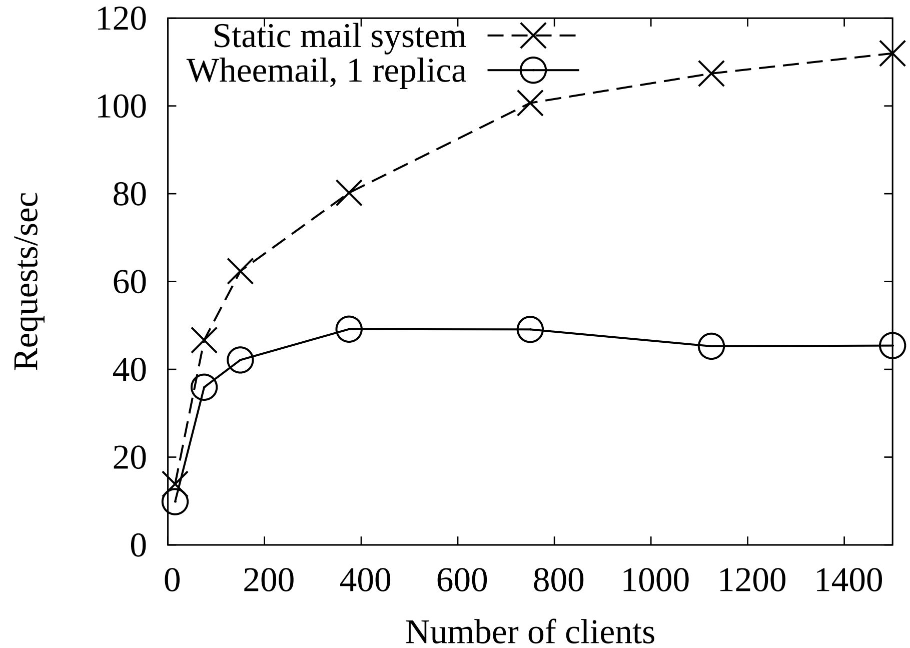

Figure 7-8: The throughput of Wheemail compared with the static system, on the Emulab testbed.

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 7.1 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 an 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 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

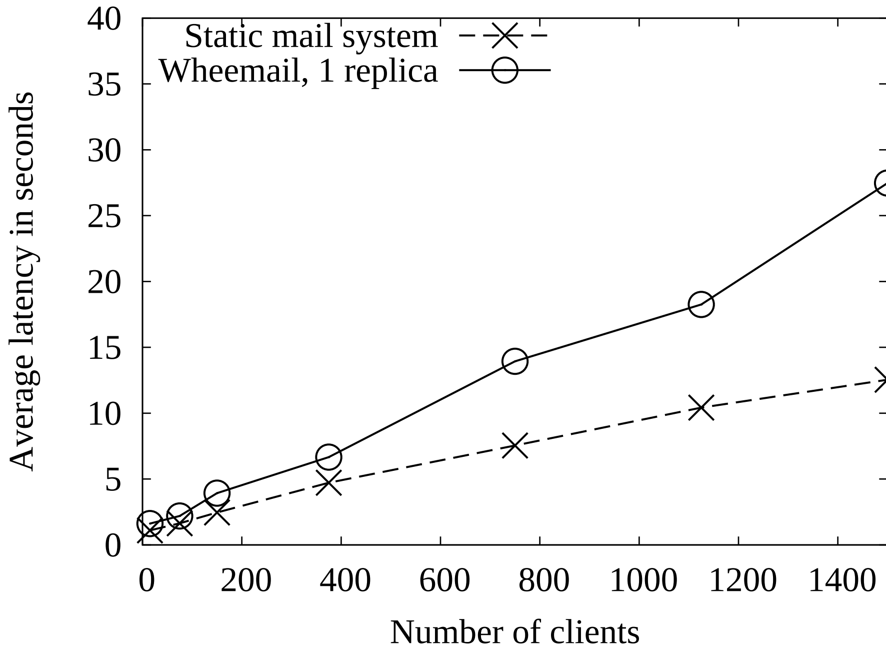

Figure 7-9: The average latencies of individual SMTP requests, for both Wheemail and the static system, on Emulab.

to that server, often across the wide-area network.

Figure 7-8 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 (messages have an average size of 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 7-9 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.

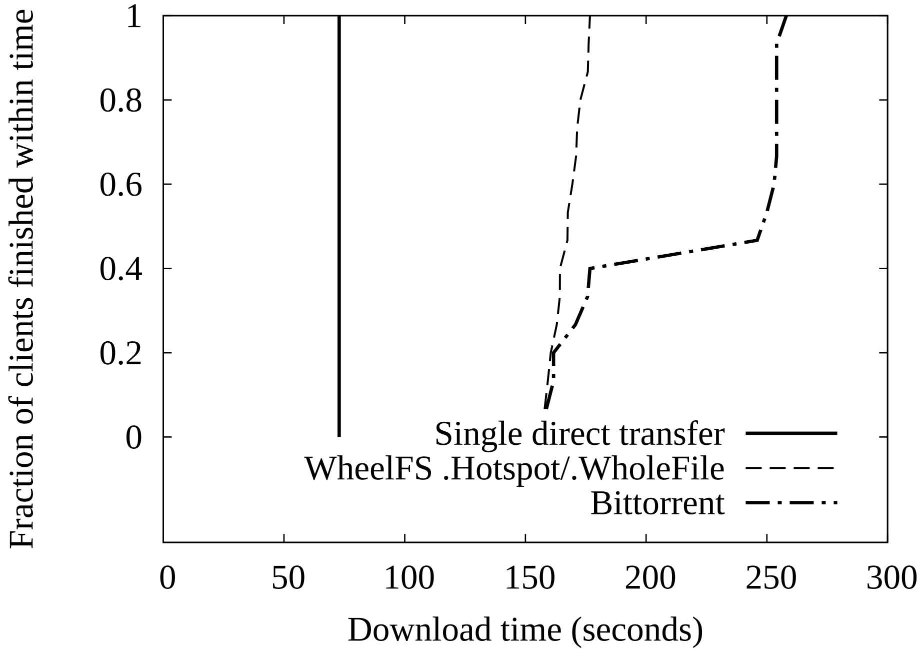

Figure 7-10: CDF of client download times of a 50 MB file using BitTorrent and WheelFS with the **.Hotspot** and **.WholeFile** cues, running on Emulab. Also shown is the time for a single client to download 50 MB directly using `ttcp`.

7.5 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 7.1. 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 [17] (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 7-10 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 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.

7.6 `distmake` Under Failures

The distributed compilation application from Section 5.5 argues that there exist applications which benefit from using different cues in different situations. This subsection runs `distmake` over WheelFS to demonstrate the improvement in failure behavior that it gains from use of different cues for different kinds of data.

Build-directory Cues	Failure?	Compile Time (sec)
Default close-to-open	No	12.2
Default close-to-open	Yes	39.9
.EventualConsistency/ .MaxTime=1000	Yes	13.5

Table 7.1: The time to compile and link an executable on WheelFS using `distmake` with different build-directory cues, both with and without a node failing after compiling an object. Numbers are the average of 10 trials, and **.RepLevel=2** is used in all cases.

In this experiment, WheelFS stores two kinds of data for `distmake`: source files in one directory, and object files and a final linked executable in a build directory. The source directory must use WheelFS's default close-to-open consistency to avoid losing programmer changes to source files or compiling from stale sources. This experiment varies the cues on the build directory and its files, using two configurations: the default close-to-open consistency, and **.EventualConsistency/.MaxTime=1000**. The expectation is that both will result in a correct final executable, but that the second set of cues will obtain better performance if worker nodes fail. This experiment runs on five Emulab nodes that are part of the same local-area cluster.

The result of a worker host failing is that it will not produce an output object file version, leaving a previous version (with an earlier date) or no readable object file at all. Depending on the given consistency cue, `make` will check for readable backup versions; if none are found, or if the application is using strict consistency, `make` will automatically re-compile the lost or unreadable file. `distmake` required a slight modification (10 lines) to provide these same semantics.

The `Makefile` specifies that four source files should be compiled to produce objects, and that the four objects (along with some existing libraries) should be linked to form an executable. There are four worker nodes (each with a 3 GHz CPU) on which `distmake` can remotely run `make`s.

Table 7.1 shows how long it takes to build the executable under different scenarios. Note that executing this `make` command without using `distmake` takes 19.9 seconds over WheelFS, on a single node. When no failures occur, the distributed build takes 12.2 seconds with either set of cues. If one of the worker nodes loses network connectivity just after finishing its compile, the times differ. For close-to-open consistency, the worker running the linker must wait for the default **.MaxTime** timeout to pass (ten seconds), and then recompile the unreadable object file, potentially incurring more timeouts to overwrite the existing file with the new object file. For **.EventualConsistency/.MaxTime=1000**, `make` on the linking worker `stat()`s the object file, WheelFS incurs a one-second timeout checking with the file's primary before failing over to the file's backup, and the backup replica satisfies the `stat()` and any future reads. Thus proper use of cues gives `distmake` the consistency it needs for source files, and good performance under faults for object files.

7.7 Distributed BLAST

WheelFS allows us to convert BLAST easily into a parallel program (see Section 5.6). We show that the resulting program, `parablast`, achieves close to ideal speedup when running over a LAN topology on Emulab.

The experiments use the protein database "`nr`" and a sample set of 19 queries from a commonly-used BLAST benchmark [19]. `parablast` partitions the database into 16 partition files, totalling

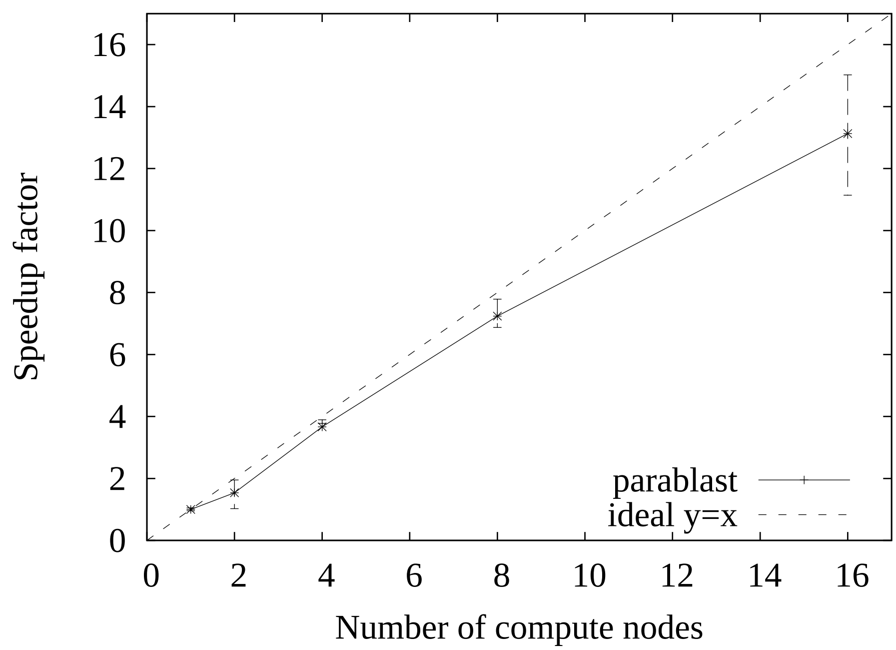

Figure 7-11: The average execution speedup for a batch of 19 BLAST queries achieved by `parablast` as a function of the number of compute nodes. It takes a single compute node 1292 secs to finish all 19 queries. The error bars represent the 20^{th}- and 80^{th}-percentile speedup factors.

Application	**LoC**	**Reuses**
CDN	1	Apache+mod_disk_cache
Mail service	4	Sendmail+Procmail+Dovecot
File distribution	N/A	Built-in to WheelFS
Dist. make	10	`distmake`
Dist. BLAST	354	BLAST
All-Pairs-Pings	13	N/A

Table 7.2: Number of lines of changes to adapt applications to use WheelFS.

a size of 673 MB. It takes a single Emulab host 1017 seconds on average to finish the batch of 19 queries using the local file system. In comparison, `parablast` would take 1292 seconds to finish when using a single compute node due to the overhead of WheelFS reads. Figure 7-11 show the average speedup of batch execution time as a function of the number of compute nodes, with error bars for the 20^{th}- and 80^{th}-percentile speedup factors. Figure 7-11 shows that `parablast` achieves achieves close to linear speedup, demonstrating that WheelFS allows CPU intensive applications like BLAST to be easily parallelized with good performance.

7.8 Implementation Ease

Table 7.2 shows the number of new or modified lines of code (LoC) we had to write for each application (excluding WheelFS itself). Table 7.2 demonstrates that developers can benefit from a POSIX file system interface and cues to build wide-area applications with ease.

7.9 Expressiveness

A number of recent wide-area storage systems offer applications control over the behavior of their data in the face of wide-area network tradeoffs. The *expressiveness* of these systems – the ability of the systems to accommodate a wide range of application-desired semantic and performance characteristics – plays a large part in determining their success in supporting many different applications. This section examines the expressiveness of WheelFS, as compared to other recent storage systems, along two axes: the features it offers, and the previous storage systems to which it can provide equivalent behavior.

7.9.1 Feature comparison

Table 7.3 compares the features offered by WheelFS and four other wide-area storage systems: PADS [10], Dynamo [24], CRAQ [74] and PNUTS [18]. It lists features that determine which wide-area applications the storage systems will be able to support. The table states whether or not a system includes a particular feature, and if so, the techniques from which an application can choose when using the feature on that system. We determined the systems' features through a close reading of their published descriptions; we have no practical experience with systems other than WheelFS.

The first four features in Table 7.3 correspond closely to the tradeoffs applications must make in the wide area (and thus, the WheelFS semantic cue categories shown in Table 3.1): placement, durability, consistency, and large reads. WheelFS provides equivalent or more expressive control than the other systems, with the exception of PADS. In PADS, system designers specify routing and blocking rules that allow their storage layer to provide arbitrary placement, durability, and consistency policies. This range of choices means that PADS developers can construct any imaginable storage layer. However, from the perspective of wide-area applications, not all of these possible configurations might be necessary. WheelFS instead focuses on the needs of actual wide-area applications, and by doing so provides a simpler storage interface than PADS. Section 7.9.2 takes a closer look at how closely WheelFS can mimic the architectures of storage systems that have been built using PADS. WheelFS is the only system that includes block-based pre-fetching and block-based cooperative caching, allowing nodes to read large and popular files efficiently.

The next features listed in the table are those that allow applications to work well in environments where nodes have intermittent connectivity to each other. Support for disconnected operation means that applications can still read and write data on a node that cannot reach any other node in the network (*e.g.*, a laptop on an airplane). Support for sticky caches means that a node can specify an explicit subset of the data in the system that should always be cached locally – when the data is updated elsewhere in the system, other nodes proactively push the updates to the node, rather than waiting for the node to request the latest copy of the data. The only system supporting either of these features is PADS. However, WheelFS does not target applications running in environments in which nodes can be disconnected for long periods of time.

Both WheelFS and PADS offer a simple, automatic method for resolving conflicts that arise due to relaxed consistency constraints. Dynamo, on the other hand, allows applications to specify custom reconciliation methods that are run whenever reads of the same object on multiple nodes return different results. We believe that many wide-area applications can structure their data in a way that makes WheelFS's simple resolution mechanism sufficient, though if it proves necessary we can extend WheelFS's cue interface to allow a choice between several alternative mechanisms.

WheelFS is the only system in Table 7.3 that allows the same object to be both read and written with different controls at different times: we refer to this as *reference-granularity control*. CRAQ and PNUTS allow reference-granularity control during data reads, but not during writes. Though

	WheelFS	PADS [10]	Dynamo [24]	CRAQ [74]	PNUTS [18]
Placement control?	Yes, over primary site, and site spread of backups	Yes, arbitrary placement based on node ID	No, default policy is random using consistent hashing	Yes, over the sites of all replicas	Yes, in that the replica with the most updates becomes primary
Durability control?	Yes, over # replicas and # replicas that must complete write	Yes, arbitrary durability policies	Yes, over # replicas and # replicas that must complete write (per instance)	Yes, over # replicas	No, fixed # of replicas, writes to replicas are always asynchronous
Consistency control?	Yes, close-to-open or eventual (with losses), per file	Yes, arbitrary consistency policies	Yes, over # replicas from which to read, # replicas to complete write (per instance)	Yes, strong or eventual (within time bound), per object	Yes, strong or eventual (with version constraint), per record
Support for large reads?	Yes, using pre-fetching and block-based cooperative caching	Yes, using whole-object cooperative caching	No	No	No
Support for disconnected operation?	No	Yes	No	No	No
Support for sticky caches?	No	Yes	No	No	No
Support for conflict resolution?	Yes, simple automatic policy	Yes, simple automatic policy	Yes, app-specified during reads	N/A (no conflicts)	No, not yet
Support for reference-granularity control?	Yes	No	No	Yes (reads)	Yes (reads)
Support for security?	Yes, using public/private key pairs for nodes and users	No, not yet	No	No	No

Table 7.3: A comparison of the features of several wide-area storage systems. For each feature, the table indicates whether the system includes the feature and, if applicable, a summary of the techniques from which an application can choose when using the feature. Section 7.9.1 describes the features in more detail.

in many cases applications will always make the same tradeoffs with respect to its data, there are some cases where reference-granularity control is needed. Section 3.7 discussed an example use of reference-granularity control in the context of Facebook.

Finally, WheelFS is the only system that includes security techniques for enforcing data access controls, though PADS may include such support in the future, and the other systems explicitly target trusted environments that do not require strong security guarantees.

7.9.2 Storage equivalence

PADS evaluates the expressiveness of its interface by testing its *architectural equivalence* to seven previously-existing storage systems [10]. The goals of WheelFS place more emphasis on the needs of existing wide-area applications than on emulating the architectures of a broad range of storage systems, and as such WheelFS does not offer full control over routing and topology policies that PADS does. It can, however, mimic the behavior of different storage layers from an application's perspective. This kind of equivalence is weaker than architectural equivalence because it provides no guarantees about matching a system's overhead, but we believe discussing equivalence from an application's perspective is a useful exercise. The rest of this subsection describes how well WheelFS can emulate the behavior of the seven storage systems built on PADS.

Generic client/server system. In this system, a single server stores all of an application's data, and clients communicate with the server to access the data [51, 63, 64]. WheelFS can emulate this architecture exactly by creating all new files and directories using **.RepLevel=1** and the **.Site** (or **.KeepTogether**) cue to place data at a site that includes only one node (the server). By default, WheelFS provides partial object writes and leases as used in the "full" client/server system implemented on PADS, and can provide cooperative caching using **.HotSpot**.

Coda [65]. WheelFS can behave like Coda in many respects, including partitioning its namespace across many servers using the **.Site** cue, and providing server replication using **.RepLevel** and **.KeepTogether**. WheelFS does not provide support for sticky caches or for cooperative caching when nodes cannot reach the primary replica. However, WheelFS does not target applications where nodes can be disconnected for long periods of time. Another architectural difference is that in Coda, client themselves push updates to all replicas of an object, while in WheelFS this is the responsibility of the primary server.

Chain Replication [78]. Chain replication is a specific algorithm for managing a small number of object replicas. Though currently WheelFS uses a standard primary/backup scheme in which the primary handles both reads and writes for the object by default, there is nothing in WheelFS's application interface that ties it to this scheme; from an application's perspective, chain replication and WheelFS would be equivalent. Furthermore, WheelFS can provide very similar consistency controls to the modified version of chain replication provided by CRAQ [74].

TRIP [53]. TRIP assumes all updates happen at a single server, which pushes those updates to client replicas using a self-tuning algorithm. Though WheelFS does not include push-based client replication (similar to sticky caches), it can propagate updates asynchronously to a specified number of replicas using **.SyncLevel**, and can provide similar consistency semantics to TRIP's. WheelFS does not offer a staleness option that clients can use to specify an acceptable level of freshness for data, though such an option could conceivably be added as a semantic cue in future work.

TierStore [25]. In order to serve nodes in developing regions with unreliable network connectivity, TierStore arranges nodes in a hierarchical tree, and writers push updates to parents and children using delay-tolerant networking. WheelFS can provide weak consistency on a per-object basis like TierStore. Because WheelFS does not target a wide-area network environment with mostly-available connectivity, it does not support architectures (like TierStore's) where all nodes have replicas for entire data subsets, nor does it support delay-tolerant networking.

Bayou [56, 75]. Similarly to TierStore, all Bayou nodes eventually get a replica of every object, which WheelFS does not currently support. WheelFS can provide similar weak consistency guarantees to Bayou, though it does so at a per-object level, rather than across all objects. From an application's perspective, we have seen little evidence that this would make a difference.

Pangaea [62]. In Pangaea, each object has a small number of "gold" replicas to guarantee its durability, exactly as WheelFS's primary and backup replicas do. Other nodes that read an object, however, maintain a "bronze" replica for that object, and receive push-based updates for the object in the future. WheelFS does not support push-based updates for a large number of replicas, but on-demand client-side caching using leases functions in much the same way.

At a high level, WheelFS provides the same storage interface for applications that these seven systems do, in terms of placement, durability, and consistency controls. One recurring drawback of WheelFS is its inability to support a large number of replicas of an object. In low-connectivity networks, which WheelFS does not target, having a replica available at every node is crucial, in case a user wants to access a file while completely offline. In wide-area networks, where nodes at the same site are rarely disconnected, a more likely requirement might be that a replica exists at every site. A possible extension to our interface that could address this issue is allowing applications to specify **.RepLevel=all** for files, to ensure that at least one replica exists at every node (or **.RepSites=all**, to maintain a replica at every site). Furthermore, client-side caching and leases mitigate some, but not all, of the problem. Primarily, though, supporting large numbers of replicas is an implementation issue, and not a fundamental limitation to semantic cues.

7.10 Discussion

When possible, this chapter 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.

Chapter 8

Related Work

There is a humbling amount of past work on the tradeoffs of availability and consistency, distributed file systems, and distributed storage in general. WheelFS builds on much of this work to offer a distributed file system that gives applications control over many important tradeoffs present in the wide area. This chapter places WheelFS in the context of decades of related work.

8.1 Storage Systems with Wide-Area Tradeoffs

Some wide-area storage systems offer configuration options in order to make them suitable for a larger range of applications. Section 7.9.1 presented a feature-by-feature comparison of four of these systems against WheelFS; this section describes these systems in more detail, along with several other related systems.

PRACTI [9] is a recently-proposed framework for building storage systems with arbitrary consistency guarantees (as in TACT [85]). Like PRACTI, WheelFS maintains flexibility by separating policies from mechanisms, but it has a different goal. While PRACTI and its recent extension PADS [10] 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). PADS could potentially implement WheelFS-like behavior using a higher layer that provides a file-system interface augmented with semantic cues; however, in its current form, PADS would be unable to support different references to the same file with different cues.

Yahoo!'s data storage platform, PNUTS [18], offers a read/write interface to records of database-like tables. Records are replicated across multiple data centers, and each replica of a particular record is guaranteed to apply updates in the same order. PNUTS supports control of wide-area tradeoffs by allowing applications to choose between reading the latest version of a record, any version of a record, or a version of a record newer than a specified version – this is similar to WheelFS's implementation of the **.EventualConsistency** cue. PNUTS also supports a `test-and-set-write` primitive, which signifies that a write should only succeed if the latest version of the record is equal to a specified version, and can migrate the master replica for a record to the data center receiving the most writes for that record. Though PNUTS and WheelFS differ in their choice of interface and set of tradeoffs exposed to the application, they share many of the same high-level goals.

A recent paper [74] proposes extending chain replication [78] with apportioned queries (CRAQ). Essentially, chain replication is an object storage model that arranges the replicas for each object in a chain (potentially spread across multiple data centers), where the writes to a replica are serviced by the head of the chain and propagate to the tail, and reads are serviced by the tail. In order to improve system throughput at the cost of consistency, CRAQ allows an application to choose to read a replica from any node in the chain. Furthermore, the application can choose how out-of-date the read can be. CRAQ provides many features similar to those exposed by WheelFS's semantic cues, though it presents an object storage interface with a flat namespace.

A bevy of wide-area storage systems offer a small set of tradeoffs aimed at particular types of applications. Amazon's Dynamo [24] 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 [11] 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 [29, 75] and Pangaea [62] 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.

GVFS [88] also observes that different applications need different consistency models to perform well in the wide-area. To provide this control, GVFS allows applications to mount existing NFS deployments using specialized middleware that enforce particular semantics in the wide-area. In contrast to WheelFS, GVFS requires a new instance of middleware for each different set of semantics an application might want. It does not support the placement, durability, or large-read functions provided by WheelFS's semantic cues.

Finally, certain Grid storage systems allow for application-controlled tradeoffs of large scientific data sets. LegionFS [84], for example, provides an object model for wide-area data, where objects are extensible and each can implement their own replication management algorithms. GridDB [49] includes a mechanism for users to provide hints about which files or directories will be modified, to eliminate spurious data copying. WheelFS aspires to be usable by Grid computation applications, and provides for a more general set of application-specific controls.

8.2 Distributed File Systems

The design and implementation of WheelFS stands on the shoulders of a number of previous distributed file systems. Though these systems vary greatly in their design and target environment, they all aim to provide access to a single hierarchical file system namespace for a distributed set of clients.

8.2.1 Centralized servers

Several popular distributed file systems store entire sub-trees of their namespace on a single node. Using NFS [63], a server can export a sub-tree of its local file system to be mounted by remote clients (though the clients are generally in the same local-area network as the server). AFS [64] cells organize data into volumes and uses aggressive local caching to scale to many clients. SFS [51] follows a similar architecture but includes many security properties not present in NFS or AFS.

WheelFS allows many clients to mount the same file system namespace, but spreads the responsibility of serving files and directories across many servers. This improves scalability since

WheelFS can keep many disks busy at once, and increases fault tolerance as the failure of one node does not render all data inaccessible, but complicates administration.

8.2.2 Local-area distributed file systems

There is a large class of distributed file systems that aggregate the storage of many local servers together, to provide a single file system with greater storage and better fault tolerance than one computer could provide on its own. As with WheelFS, these systems spread files and directories across many servers, though they generally strive to provide a consistent view of data in order to serve the workloads of desktop users.

Deceit [67] is an early cluster file system, meant to be a replacement for NFS allowing for transparent file locations. Nodes accessing a file become part of a broadcast group for that file, and gets future updates for the file; only one node per group is allowed to be the writer of a file at any one time. Deceit features per-file settings for replication levels and other settings, similar in spirit to WheelFS's semantic cues but covering a different set of tradeoffs, aimed at local-area users and applications.

xFS [6] is one of the first local-area file systems to be "serverless": its goal is to not have any central servers that can act as a bottleneck. It accomplishes this by striping groups of data across multiple servers (as in Zebra [40]) and globally-replicating a mapping allowing clients to determine which nodes were responsible for which files. WheelFS's design is similar to this design at a high level. xFS also recognizes that co-locating file management with the file creator is good for performance, a principle guiding WheelFS's default data placement policy.

Frangipani [76] is similar to xFS in design, though it uses a two-layer approach to split the tasks of storing data from the task of providing a consistent file system. Frangipani places great focus on file system recovery and reconfiguration, areas that were not addressed by xFS. Neither Frangipani nor xFS allow for application-controlled tradeoffs, as they are meant to be file systems for end users.

Farsite [1] is a cluster file system aiming to replace centralized file servers, and assumes a high-bandwidth, low-latency network. Farsite provides mechanisms for maintaining data integrity in the face of Byzantine failures, a subject not addressed by the current WheelFS design. Farsite also includes sophisticated mechanisms to atomically renaming files across directories and to sub-divide directory responsibility among multiple nodes using a directory service [27], for highly scalable directories. A wide-area version of such techniques may be applicable to future versions of WheelFS.

A few cluster file systems target storage for applications as WheelFS does, as opposed to end users. Systems such as GFS [36] and Ceph [82] have demonstrated that a distributed file system can dramatically simplify the construction of distributed applications within a large cluster with good performance. Freely-available systems such as the Hadoop File System [41], Lustre [50] and GlusterFS [38] have further popularized cluster-based file systems. Extending the success of such cluster file systems to the wide-area environment, however, is challenging because they do not expose wide-area tradeoffs to applications.

8.2.3 Wide-area distributed file systems

Several distributed file systems have targeted wide-area operation, though none provide applications the control they need to combat wide-area challenges. Some centralized server file systems, such as AFS and SFS, include support for wide-area access but can not easily spread the load of serving data across multiple servers.

Echo [12] is an early wide-area file system, aiming to scale globally using specialized naming and caching techniques. In contrast to WheelFS, Echo divides data into volumes, and each volume is managed by a single computer; data cannot be spread among many nodes without extensive volume reconfiguration. Furthermore, Echo offers only strict consistency to its clients.

JetFile [39] is a highly-decentralized file system design to be a drop-in replacement for a local file system, but operate in challenging network environments like the wide area. Each file in JetFile has its own multicast address, and nodes distribute files via multicast in order to keep file locations transparent. Jetfile uses callbacks and leases, much like WheelFS, to maintain cache coherency among clients. JetFile does not offer application control of tradeoffs such as consistency; instead, the performance of the network dictates the performance of its cache coherency protocols.

Shark [7] shares with WheelFS the goal of allowing client-to-client data sharing, and uses cryptographic techniques to ensure that data integrity and security is maintained in the presence of untrusted peers. However, a centralized server is ultimately responsible for the entire set of files in Shark, limiting its scalability for applications in which nodes often operate on independent data.

TierStore [25] is a distributed file system for developing regions, in which nodes might not have reliable connectivity to each other. TierStore employs delay-tolerant networking, extensive caching, and relaxed consistency semantics to achieve good performance in the face of these extreme network challenges. The TierStore designers recognize that providing a file system interface to wide-area applications can allow for extensive code reuse in many instances. Although TierStore does not support per-file run-time tradeoffs, it does allow for developers to write extensions to its object class for objects that must behave in certain ways. The TierStore paper does not discuss whether applications can specify these extensions through the file system interface.

8.3 Miscellaneous Distributed Storage Systems

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 [3, 17, 32, 61] or adapt an existing system [54, 79]. Unfortunately, most existing storage systems, even more general ones like OceanStore/Pond [58] or S3 [60], are only suitable for a limited range of applications and still require a large amount of code to use. Distributed hash tables (DHTs) [69, 87] 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 [68] and CoralCDN [32] both use a DHT, but their DHTs differ in many details and are not interchangeable. Furthermore, many DHTs are write-only, and do not allow applications to update data items.

Sinfonia [2] 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.

Chapter 9

Discussion, Future Work and Conclusion

This dissertation described WheelFS, a wide-area storage system with a traditional POSIX interface augmented by cues that allows distributed applications to control wide-area tradeoffs. WheelFS is the result of a nearly three-year design and implementation effort, focusing on developing a useful system to meet the wide-area storage needs of real distributed applications. The success of WheelFS will be measured mostly by the impact of two of this dissertation's main contributions: semantic cues and the idea that a file system can be the proper storage abstraction for wide-area applications. This chapter reflects on these contributions, discusses future work for WheelFS, and concludes.

9.1 Semantic Cues

This dissertation introduced the notion of semantic cues: controls offered by a storage system to an application for the purpose of choosing tradeoffs at a per-file granularity. Semantic cues, as a concept, are useful not just for WheelFS, or distributed file systems, but to any storage system. The usefulness of semantic cues comes from enabling applications to make choices about the behavior and semantics of its data at run-time; often, the application is the only possible place to make this choice. Semantic cues provide a clean, portable way for storage systems to offer this choice to applications.

It is important, however, not to overwhelm the application with choices. The success of systems that employ semantic cues will likely depend on whether they offer the "right" set of cues for the "right" set of applications. To that end, in order to be useful to applications that need wide-area storage, WheelFS offers a set of cues that control only those tradeoffs that exist as a consequence of storing data in the wide area. Moreover, rather than provide a full range of possible values for each tradeoff as previous models suggest [9, 10], WheelFS consciously scales back the range of choices available to the application. For example, eventual consistency can either be on or off for individual files or directories – consistency cannot be defined across a collection of objects, or specified using continuous numerical ranges [85]. An earlier design of WheelFS [72] even included cues for relaxing consistency on reads and writes separately, but as the design evolved, it became clear that limiting the set of cues has great value. The restrictions are based on the needs of real, deployed wide-area applications and simplify WheelFS's interface, making it easier for developers to use.

Whether WheelFS's four categories of semantic cues (placement, durability, consistency, and large reads) are indeed the "right" categories remains to be seen. This dissertation has shown that these categories are derived from fundamental wide-area network properties, and work well for six different applications (all-pairs-pings, a distributed Web cache, distributed mail, file distribution, distributed compilation, and BLAST), but the real test will come as developers try to write new applications on WheelFS. We hope that the WheelFS PlanetLab deployment will inspire developers to use WheelFS for previously-unimagined applications, testing the limits of its chosen interface.

9.2 Another Distributed File System?

Distributed file systems have existed for decades, going through numerous incarnations as application and user workloads evolve. Given the large body of past work, a reasonable question to ask is whether the world needs yet another distributed file system. Is WheelFS truly addressing an unsolved problem?

Answering that question definitively may take time, as developers try WheelFS and see how well it can support their applications. This dissertation has demonstrated, however, that many wide-area applications can be realized on WheelFS, and moreover, that significant code reuse of their existing, non-distributed counterparts can be achieved. No existing wide-area storage system has accomplished this, and so many applications build custom storage layers to suit their needs. From this standpoint WheelFS, as a new distributed file system, is a success.

Another data point in WheelFS's favor is that, despite years of active shared distributed research testbeds, there is still no wide-area file system mountable by all testbed nodes. Though this seems to be a basic need of any such testbed, the lack of its existence suggests the shortcomings of previous systems. Perhaps it is simply that solid implementations of these systems are not available, but this dissertation has argued that, without suitable application-level controls, these systems cannot hope to support the wide variety of workloads present on research testbeds. WheelFS provides these controls, and we hope that it will prove to meet the diverse storage needs of wide-area applications on these testbeds.

Another question to ask in this area, however, is whether a file system is indeed the right interface for wide-area storage systems. A file system imposes hierarchy on an application that might not otherwise need it and prevents optimizations within the storage layer that might be possible with structured data [16, 24]. Exposing only a file system interface, instead of a library interface, limits the application's ability to provide custom code (*e.g.*, for reconciling divergent replicas) or succinctly perform complex operations spanning multiple objects (*e.g.*, mini-transactions [2]). We believe that the benefits of rapid prototyping and reusing existing application code will overcome the disadvantages of the file system interface, but only extensive experience with outside developers will tell for sure.

9.3 Future Directions

The main thrust of future WheelFS development will be in supporting the PlanetLab deployment, and potentially adjusting WheelFS to meet the needs of third-party application developers. In the spirit of OpenDHT [59], WheelFS will be available as a service to PlanetLab developers, and may have to evolve to best support its users. Supporting a real application with a user base that runs on top of WheelFS is a major goal of the project.

Another area for potential future work is to improve the scalability of WheelFS beyond our initial goal of testbed-sized deployments. In the current design, the master of the configuration service's replicated state machine handles the `fetch`, `acquire`, and `renew` requests from all the nodes in the system; removing this point of centralization is an obvious place to begin improving WheelFS's scalability. Similarly, because all accesses go through a primary node by default, popular files and directories that are frequently updated can cause their primaries to be overloaded. Applying techniques from Farsite's directory service [27] may help to alleviate such hot spots, though further research is needed.

Finally, WheelFS may be able to reduce the complexity of its interface further. For example, the **.HotSpot** and **.WholeFile** cues do not necessarily have to be supplied by applications; WheelFS should be able to monitor application behavior and predict when employing such techniques would be useful. Furthermore, timeouts supplied to the **.MaxTime** cue apply only at the level of individual RPC timeouts with WheelFS, not at the application level, which may be the more reasonable behavior. Determining how to best divide an application-supplied timeout over potentially numerous remote calls is an interesting area of future work.

9.4 Conclusion

Applications that distribute data across wide-area 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 dissertation has presented WheelFS, which uses semantic cues in the context of a new distributed file system in order to meet this challenge. The cues that WheelFS supports provide control over fundamental wide-area storage tradeoffs. We hope that future wide-area application developers find WheelFS useful, and benefit from our storage "wheel" without having to re-invent their own.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[3] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, November 2005.

[4] Stephen Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David Lipman. Basic local alignment search tool (BLAST). In *Journal of Molecular Biology*, pages 403–410, October 1990. `http://www.ncbi.nlm.nih.gov/blast/`.

[5] Thomas Anderson and Timothy Roscoe. Learning from PlanetLab. In *Proceedings of the 3rd USENIX Workshop on Real Large Distributed Systems (WORLDS)*, November 2006.

[6] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[7] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[8] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating systems support for planetary-scale network services. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[9] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[10] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Mike Dahlin, and Robert Grimm. PADS: A policy architecture for building distributed storage systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[11] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total Recall: System support for automated availability management. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[12] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. *Digital Equipment Corporation, Systems Research Center, Research Report 111*, September 1993.

[13] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[14] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[15] Cassandra project. `http://incubator.apache.org/cassandra/`.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[17] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[18] Brain F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the 34th Internation Conference on Very Large Databases (VLDB)*, August 2008.

[19] George Coulouris. Blast benchmarks, 2005. `ftp://ftp.ncbi.nih.gov/blast/demo/benchmark/`.

[20] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. A decentralized network coordinate system. In *Proceedings of the 2004 ACM Special Interest Group on Data Communication (SIGCOMM)*, August 2004.

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

[22] Frank Dabek, M. Frans Kaashoek, Jinyang Li, Robert Morris, James Robertson, and Emil Sit. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[23] Jeffrey Dean. Challenges in building large-scale information retrieval systems. In *Keynote of the 2nd ACM International Conference on Web Search and Data Mining (WSDM)*, February 2009.

[24] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[25] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A distributed filesystem for challenged networks in developing regions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, February 2008.

[26] distmake: a distributed make program. `http://distmake.sourceforge.net`.

[27] John R. Douceur and Jon Howell. Distributed directory service in the Farsite file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

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

[29] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology (UIST)*, October 1997.

[30] Facebook now growing by over 700,000 users a day, and new engagement stats. `http://www.insidefacebook.com/2009/07/02/facebook-now-growing-by-over-700000-users-a-day-updated-engagement-stats/`, July 2009.

[31] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Schvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1):113–156, June 1999.

[32] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[33] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazières. OASIS: Anycast for any service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

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

[35] GENI: Global environment for network innovations. `http://www.geni.net`.

[36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, December 2003.

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. 33(2):51–59, June 2002.

[38] Gluster - home. `http://www.gluster.org`.

[39] Björn Grönvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[40] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

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

[42] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Symposium on Theory of Computing*, May 1997.

[43] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, 18(3):263–297, August 2000.

[44] Ravi Kosuri, Jay Snoddy, Stefan Kirov, and Erich Baker. IBP-BLAST: Using logistical networking to distribute BLAST databases over a wide area network. In *The 12th International Conference on Intelligent Systems for Molecular Biology (ISMB), poster session*, August 2004.

[45] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[46] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[47] libssh - the SSH Library! http://www.libssh.org.

[48] Subversion repository [0xbadc0de.be]. http://www.0xbadc0de.be/wiki/libssh:svn.

[49] David T. Liu, Ghaleb M. Abdulla, Michael J. Franklin, Jim Garlick, and Marcus Miller. Data-preservation in scientific workflow middleware. In *Proceedings of the 18th Scientific and Statistical Database Management Conference (SSDBM)*, July 2006.

[50] Lustre file system - overview. http://www.sun.com/software/products/lustre/.

[51] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[52] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO)*, August 1987.

[53] Amol Nayate, Mike Dahlin, and Arun Iyengar. Transparent information dissemination. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, October 2004.

[54] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[55] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Tharlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE)*, May 2000.

[56] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[57] Xavid Pretzer. Securing wide-area storage in WheelFS. Master's thesis, Massachusetts Institute of Technology, June 2009.

[58] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[59] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of the 2005 ACM Special Interest Group on Data Communication (SIGCOMM)*, August 2005.

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

[61] Yasushi Saito, Brian Bershad, and Henry Levy. Manageability, availability and performance in Porcupine: A highly scalable internet mail service. *ACM Transactions of Computer Systems*, 18(3):298–332, August 2000.

[62] Yasushi Saito, Christos Karamonolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[63] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, June 1985.

[64] M. Satyanarayanan, John Howard, David Nichols, Robert Sidebotham, Alfred Spector, and Michael West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, 1985.

[65] M. Satyanarayanan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Comp.*, 4(39):447–459, Apr 1990.

[66] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, December 1990.

[67] Alexander Siegel, Kenneth P. Birman, and Keith Marzullo. Deceit: A flexible distributed file system. *Cornell University Technical Report TR89-1042*, November 1989.

[68] Emil Sit, Robert Morris, and M. Frans Kaashoek. UsenetDHT: A low-overhead design for Usenet. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2008.

[69] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11:149–160, February 2003.

[70] Jeremy Stribling. PlanetLab All-Pairs-Pings. `http://pdos.csail.mit.edu/~strib/pl_app/`.

[71] Jeremy Stribling, Daniel Aguayo, and Maxwell Krohn. Rooter: A methodology for the typical unification of access points and redundancy. In *The 2005 World Multiconference on Systemics, Cybernetics and Informatics (WMSCI)*, July 2005. Accepted, but not published.

[72] Jeremy Stribling, Emil Sit, M. Frans Kaashoek, Jinyang Li, and Robert Morris. Don't give up on distributed file systems. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2007.

[73] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[74] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the 15th USENIX Techical Conference*, June 2009.

[75] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, October 1995.

[76] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[77] Peter Vajgel. Needle in a haystack: efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`, April 2009.

[78] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[79] J. Robert von Behren, Steven Czerwinski, Anthony D. Joseph, Eric A. Brewer, and John Kubiatowicz. Ninjamail: the design of a high-performance clustered, distributed e-mail system. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP)*, August 2000.

[80] Limin Wang, Vivek Pai, and Larry Peterson. The effectiveness of request redirecion on CDN robustness. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[81] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, October 1993.

[82] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[83] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[84] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, November 2001.

[85] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS*, 20(3):239–282, August 2002.

[86] Irene Y. Zhang. Efficient file distribution in a flexible, wide-area file system. Master's thesis, Massachusetts Institute of Technology, June 2009.

[87] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[88] Ming Zhao and Renato J. Figueiredo. Application-tailored cache consistency for wide-area file systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2006.