

# Improving Access to Remote Storage for Weakly Connected Users

*Patrick Randolph Eaton*

Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2007-11

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-11.html>

January 11, 2007



Copyright © 2007, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**Improving Access to Remote Storage for Weakly Connected Users**

by

Patrick Randolph Eaton

B.S. (University of Illinois, Urbana) 1999  
M.S. (University of California, Berkeley), 2002

A dissertation submitted in partial satisfaction of the  
requirements for the degree of  
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION  
of the  
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:  
Professor John Kubiawicz, Chair  
Professor Eric Brewer  
Professor Ray Larson

Spring 2007

The dissertation of Patrick Randolph Eaton is approved:

---

Chair

Date

---

Date

---

Date

University of California, Berkeley

Spring 2007

# **Improving Access to Remote Storage for Weakly Connected Users**

Copyright 2007

by

Patrick Randolph Eaton

## Abstract

### Improving Access to Remote Storage for Weakly Connected Users

by

Patrick Randolph Eaton

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Kubiatowicz, Chair

Storage researchers and service providers have been developing a new type of large-scale, distributed storage system. These systems aggregate storage from a large number of components, exploiting the content-addressable interface to address and manage data stored in the system. Potentially, such systems may be able to offer managed storage services to many new user groups, but they must yet overcome many challenges. In this thesis, we consider several important challenges that these new content-addressable storage systems confront when serving weakly connected clients.

First, we justify why weakly connected users are an attractive target user population for these new storage systems. We then analyze the usage patterns of these users to understand the most important issues that they face. This background leads us to focus on techniques to improve the effective bandwidth and latency of the “last-mile” network link while protecting the privacy and integrity of data from untrusted agents in the infrastructure.

We present three mechanisms to address these challenges. We describe privacy-preserving delta-encoding that can improve the effective bandwidth of the network link without sacrificing data privacy. We introduce intention updates, which allow clients to implement flexible link scheduling policies to reduce the effective latency of the link without sacrificing data consistency or visibility. Finally, we propose a new interface that raises the granularity of content-addressable storage from individual blocks to collections of blocks, reducing the cost managing data for both clients and storage systems.

Finally, we describe and evaluate Moxie, a prototype that demonstrates how to integrate these mechanisms into a single storage system. Moxie provides a user-level file system client that allows end users to interact with a content-address storage system through the traditional file system

interface. The file system client executes requests, transparently applying delta-encoding and intention updates when appropriate. Moxie improves client-perceived performance for weakly connected users by reducing the amount of data that must be transferred across the weak link and moving the transfers that must occur off of the critical path for most operations.

---

Professor John Kubiawicz  
Dissertation Committee Chair

FOR COURTNEY



# Contents

<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>xiii</b>
<b>I Defining the Problem</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Remote Storage Systems, Features, and Limitations . . . . .	3
1.2 Underserved User Populations . . . . .	5
1.3 Important Technology and Social Trends . . . . .	7
1.4 System Model . . . . .	8
1.4.1 System Architecture . . . . .	9
1.4.2 Constraints . . . . .	10
1.4.3 Problems . . . . .	10
1.5 Goals . . . . .	11
1.6 Current Approaches . . . . .	13
1.7 Contributions . . . . .	15
<b>2 Understanding the Problem</b>	<b>17</b>
2.1 Weak Networks . . . . .	17
2.2 Metrics . . . . .	19
2.3 Potential for Data Compression . . . . .	21
2.4 Impact of Mobility . . . . .	24
2.4.1 Synthesizing Mobile Workloads . . . . .	25
2.4.2 Simulation Approach . . . . .	26
2.4.3 Workload Demands of Mobile Users . . . . .	27
2.4.4 Discussion . . . . .	34
<b>3 Related Work</b>	<b>36</b>

<b>II</b>	<b>Techniques for Improving Access for Weakly Connected Clients</b>	<b>42</b>
<b>4</b>	<b>Bandwidth-Efficient, Privacy-Preserving Updates for Legacy Applications</b>	<b>43</b>
4.1	A Data Structure for Bandwidth-Efficient, Privacy-Preserving Updates . . . . .	44
4.1.1	Requirements . . . . .	44
4.1.2	A Basic Solution . . . . .	46
4.1.3	Supporting Arbitrary Insertions . . . . .	48
4.2	Extracting Bandwidth-Efficient Updates for Legacy Applications . . . . .	50
4.3	Evaluation . . . . .	55
4.3.1	Simulation Overview . . . . .	55
4.3.2	Workloads . . . . .	56
4.3.3	Workload Commonality . . . . .	58
4.3.4	Simulation Results . . . . .	58
<b>5</b>	<b>Intention Updates</b>	<b>64</b>
5.1	Background: Write-back . . . . .	64
5.2	Using Intent to Decrease Latency . . . . .	68
5.3	Intention Update Variations . . . . .	72
5.4	Implementation Issues . . . . .	75
5.4.1	Architectural Requirements . . . . .	75
5.4.2	Failure Modes . . . . .	76
5.5	Analytical Evaluation . . . . .	76
5.5.1	Preliminaries . . . . .	77
5.5.2	Preemptive Priority Queueing . . . . .	78
5.5.3	Fluid Approximation . . . . .	80
5.6	Evaluation via Simulation . . . . .	87
5.6.1	Simulation Methodology . . . . .	87
5.6.2	Modeling Client Protocols . . . . .	89
5.6.3	Simulating Synthetic Workloads . . . . .	92
5.6.4	Simulating Traced Workloads . . . . .	95
5.7	Simulating Variations . . . . .	97
5.8	Discussion . . . . .	99
<b>6</b>	<b>Extent-Based Content-Addressable Storage</b>	<b>101</b>
6.1	Background . . . . .	101
6.1.1	Self-Verifying Data . . . . .	102
6.1.2	Content-Addressable Storage Systems . . . . .	102
6.1.3	Consequences of the CAS Interface . . . . .	105
6.2	Improving CAS with Aggregation . . . . .	106
6.3	Extent-Based Content-Addressable Storage . . . . .	107
6.3.1	Overview of Extent-Based CAS . . . . .	107
6.3.2	An API for Extent-Based CAS . . . . .	109
6.3.3	An Example Use of the Extent-Based API . . . . .	112
6.4	A Secure, Append-Only Log Abstraction . . . . .	114

6.5	Example Application: Versioning Back-up . . . . .	116
6.6	Prototype Implementation and Evaluation . . . . .	119
6.6.1	Antiquity: A Distributed, Extent-Based CAS System . . . . .	119
6.6.2	Experimental environment . . . . .	120
6.6.3	Performance results . . . . .	121
<b>III</b>	<b>Putting It All Together</b>	<b>125</b>
<b>7</b>	<b>Moxie: A Prototype Remote Storage System for Weakly Connected Clients</b>	<b>126</b>
7.1	Moxie Overview . . . . .	126
7.2	Moxie Implementation . . . . .	128
7.2.1	User-Level File System Toolkits . . . . .	129
7.2.2	Moxie Client . . . . .	130
7.2.3	Moxie Server . . . . .	131
7.3	Implementation Issues . . . . .	132
7.3.1	File Summaries and Delta Encoding . . . . .	132
7.3.2	Control of Data Write-back . . . . .	133
7.3.3	Supporting Intents in the Secure Log . . . . .	134
7.4	Experimental Environment . . . . .	135
7.4.1	User-Level NFS Implementation . . . . .	135
7.5	Experimental Results . . . . .	136
7.5.1	Delta-Encoded Updates . . . . .	136
7.5.2	Intention Updates . . . . .	137
7.5.3	Reordering Data Transfers . . . . .	140
7.5.4	The Tcl Benchmark . . . . .	143
7.5.5	The Cost of Security . . . . .	149
7.6	Interfaces for Intention Updates . . . . .	151
7.6.1	Defining Data Freshness Requirements . . . . .	151
7.6.2	Defining Write-back Requirements . . . . .	153
7.6.3	Failure Recovery . . . . .	154
<b>8</b>	<b>Future Work and Conclusion</b>	<b>156</b>
8.1	Future Work . . . . .	156
8.2	Key Lessons . . . . .	157
8.3	Summary . . . . .	158
	<b>Bibliography</b>	<b>160</b>

# List of Figures

1.1	<b>Remote Storage System:</b> Remote storage systems store data on behalf of users. Client machines access data stored in remote storage systems by sending requests to the storage system via the network. . . . .	4
1.2	<b>System Model:</b> We assume a client-server architecture. The server resides in the well-connected core of the network. Clients connect via weak connections at the edge of the network. Client-perceived performance is primarily dependent on speed of the weak link. . . . .	9
2.1	<b>Issue Time Penalty:</b> The issue time penalty is the difference between the time an event is issued in two systems. Here we compare an “ideal” system that services requests with zero latency to a realistic system. This measurement rewards systems that are responsive enough that they do not block user requests. . . . .	20
2.2	<b>Conflicts:</b> Conflicts occur when requests in a real system reach the server in an order that differs from the execution order of an ideal system. In this example, in the “ideal” system, write $W_{F_n}$ replaces $F_{n-1}$ ; later, another user issues read $R_{F_n}$ to read that file and write $W_{F_{n+1}}$ to update the file. In a system constrained by weak networks, a user issues $W_{F_n}$ but transferring the data to the server is slow. When another user reads the file $R_{F_{n-1}}$ , the server returns the last known version $F_{n-1}$ resulting in a conflict. The subsequent write $W_{F_{n+1}}$ results in another conflict because it overwrites a version, $F_{n-1}$ , different from the ideal case. When the initial write, $W_{F_n}$ , finally completes, it creates another conflict because again the wrong version, $F_{n+1}$ , is overwritten. . . . .	21
2.3	<b>Modelling Mobile User Workloads:</b> To synthesize workloads of mobile users, we augment traditional file system traces to reflect accesses from multiple locations and devices. We begin by identifying <i>sessions</i> , sequences of operations with no idle period longer than the <i>timeout</i> period. We then apply a mobility model to assign the sessions to different devices. . . . .	25
2.4	<b>Session Characteristics:</b> (a) The cumulative distribution function of session duration as the session timeout varies. (b) The cumulative distribution function of session operation count as the session timeout varies. . . . .	29

2.5	<b>Impact of Mobility on Cache Management Traffic:</b> The amount a cache management traffic needed to ensure that all devices access up-to-data varies with the defined timeout period. The amount of traffic is normalized by the amount of data transferred when all traffic for a given user originates from a single device. . . . .	30
2.6	<b>Impact of Weak Connectivity on Performance:</b> The issue time penalty, reflecting user-perceived slowdown, of clients accessing remote storage using synchronous and asynchronous write-back. . . . .	31
2.7	<b>Impact of Weak Connectivity on Durability:</b> The durability latency measures the time until data is stored durably at the server using asynchronous write-back. . . .	32
4.1	<b>A Data Structure to Support Bandwidth-Efficient, Privacy-Preserving Updates:</b> A B-tree that indexes data by <i>relative</i> offsets enables bandwidth-efficient updates. (a) The tree maintains an index over six blocks of data, shown at the leaves. The amount of user data stored in each data node is indicated in the figure. The root shows that the first 175 bytes of data can be found via the left subtree; data between offset 175 and 350 can be accessed via the right subtree. Note, both base internal nodes contain the same counts because the data structure uses relative indexing. (b) Maintaining relative offsets enables efficient incremental insert operations. Shown is the result of inserting 75 bytes at offset 100 in the data structure shown in (a). Blocks modified during the operation are shaded. . . . .	46
4.2	<b>Data Structure Extension to Support Arbitrary Insertions:</b> By maintaining an “offset” which points into the middle of a block of ciphertext, the data structure can support the insertion and deletion of data at arbitrary offsets. This extension might enable more efficient updates. . . . .	49
4.3	<b>Client-Side Delta Encoding:</b> When an application accesses a file via the <code>open()</code> operation, the client machine reads the file, if necessary, and caches it locally. It computes the chunk boundaries for the object and constructs a file summary. In this example, the fingerprinting algorithm divides the file into five chunks; the file summary has one entry for each chunk. On <code>write()</code> operations, the client simply modifies the file in the local cache. The figure indicates that the application has written to two regions of the file. On <code>close()</code> , the client computes the chunk boundaries for the new version of the file and constructs a new file summary. Here, the new version contains four chunks. The first and last chunk remain unchanged from the previous version. The client then compares the new file summary against the previous file summary and computes an update to submit to the server. File offsets ‘x’ and ‘y’ referenced in the update are indicated on the chunked copy. . . .	51
4.4	<b>Contents of a Data Block that Supports Encryption:</b> Before shipping a chunk to the server, the client packages it in a block that contains the length in bytes of actual user data (not including padding or initialization vectors), the byte array containing the initialization vector, and the ciphertext. . . . .	53
4.5	<b>A Simulator for Evaluating Bandwidth-Efficient Updates:</b> Our evaluation is based on a simple simulator in which the client and server directly via an emulated low-bandwidth network link. . . . .	55

4.6	<b>Micro-benchmark Update Latency:</b> The client-perceived write latency depends on the amount of commonality detected in the workload and the characteristics of the network. . . . .	59
4.7	<b>Micro-benchmark Read Latency:</b> The client-perceived read latency also depends on the commonality in the workload and the network characteristics. The results differ from the update latency because clients must retrieve the internal nodes of the data structure. . . . .	60
4.8	<b>Client-Side Computational Overhead:</b> The graphs show the relative cost of additional client-side computation for various network speeds. For fast connections, the computation overhead can be as much as 5%. For slower networks, the overhead is negligible. . . . .	61
4.9	<b>Client-Side Computational Profile:</b> The charts show the relative cost of different phases of computation. For both traces, data encryption is the most expensive phase. . . . .	62
4.10	<b>Storage Overhead:</b> The storage overhead of the data structure is significant for small files. For larger files, the overhead is much smaller. . . . .	62
5.1	<b>Popular Write-back Strategies:</b> (a) Early remote file systems like NFS transmit data to the server on each <code>write()</code> operation. (b) AFS implements write-on-close. Clients buffer changes locally until the application closes the file; on <code>close()</code> , the client blocks until data can be transferred to the server. (c) Coda uses a write-on-close approach similar to AFS. To support weakly connected clients, Coda transfers data to the server asynchronously without blocking the application. . . . .	65
5.2	<b>An Example Demonstrating Different Write-back Strategies:</b> A user writes a big file, <b>B</b> , and a small file, <b>S</b> , and then shares the small file with another user. (a) In traditional file systems, like AFS, write operations block while changes are propagated to the server. (b) With Coda's trickle reintegration, write operations complete immediately and data is transferred to the server asynchronously in the background. . . . .	67
5.3	<b>A Comparison of Coda's Trickle Reintegration and Intention Updates:</b> With trickle reintegration, a response to the update request signals neither visibility or durability which are both tied to the transfer of the update message. Using intention updates, a response guarantees update visibility; durability is not assured until the completion of the data transfer phase. . . . .	69
5.4	<b>Behavior of Intention Updates to Example Scenario:</b> Notice of writes by Client 1 are propagated to the server quickly, and the client does not receive a response until the update is visible. When Client 2 attempts to retrieve the changes to file <i>S</i> , the system blocks the request until data corresponding to the visible changes are available at the server, ensuring the Client 2 fetches the latest copy of the data. . . .	71
5.5	<b>Example Demonstrating the Benefits of Reordering and Fragmenting Data Transfers:</b> When a client requests data that is visible at the server but not yet durable, the server can suggest to the originating client that the requested data be transferred with higher priority. The originating client can execute data transfers in any order and even pausing in the middle of a transfer. Compare this example to Figure 5.4 to see how reordering and fragmenting transfers can facilitate sharing. . .	73

5.6	<b>Modeling Intention Updates:</b> Updates are divided into two events of different priority and different size. The service time is assumed to be the transmission delay due to the the weak network connection between the client and the server. . . . .	79
5.7	<b>Analysis of Average System Time:</b> The client has 128 kbit/s of upload bandwidth and submits updates with average size of 10 KB. When using intention updates, the intent is 1 KB. Message sizes are normally distributed with $\sigma^2 = 1$ . The plot has been split to reveal detail at both upper and lower limits of utilization. . . . .	80
5.8	<b>Fluid Approximation Example:</b> An example illustrating the utility of the fluid approximation for queues. (a) During overload, a time-varying arrival rate can exceed the service rate. (b) To compute the queue length, we first compute the cumulative number of event arrivals and departures. The length of the queue is represented by the area between the curves. (c) A queue begins to form when the instantaneous utilization factor exceed unity and is at its longest when utilization factor drops back below the service rate. . . . .	83
5.9	<b>Approximating Queue Length with Intention Updates:</b> (a) The request stream exhibits bursts of activity at $3 < t < 5$ and $13 < t < 15$ . (b) Each request is converted into two network-level requests that each require a different amount of work to process. The graph plots the cumulative amount of work in the system of each type and the amount of work serviced. (c) With the first burst of activity, queues containing both types of requests begin to form. The intent queue is drained, and the client receives a response to the last request of the first burst, at time $t = 8$ . The system then works to transfer data until it is interrupted by the next burst. The client receives a response to the last request of the second burst at time $t = 16$ . Data transfer is finally completed at time $t = 43$ . . . . .	86
5.10	<b>Approximating Queue Length with Unified Updates:</b> An analysis of the queue length of the system shown in Figure 5.9 assuming traditional, unified updates. The queue is drained, and the client receives a response to the last request, at time $t = 35$ . . . . .	87
5.11	<b>Simulator Architecture:</b> The simulator executes an identical stream of requests on two systems in parallel. Typically, the “reference” system is configured to simulate ideal network links (with no latency and infinite bandwidth), while the “test” system models more realistic connections. Servers process requests without adding delay. After the event is executed in both system, the simulator compares the outcomes and records the results. . . . .	88
5.12	<b>The Bursty Workload:</b> The bursty workload is a synthetic workload that can be useful for studying the response of storage systems in overload. In this workload, clients alternate between idle periods and periods of activity. Parameters control the intensity of the bursty periods and characteristics of the file set. . . . .	93
5.13	<b>CDF of Store Operation Performance with Uniform File Size:</b> The synthetic bursty workload is created with 10-minute cycle time, 30-second bursty period, and 10-second average interarrival time. Each client has a file set of 1000 4KB files, and 20% of accesses are writes. The reference system uses ideal links; the test system uses telephone modems. . . . .	94

5.14	<b>CDF of Store Operation Performance with Workload from Roselli Traces:</b> The plots show the response of different protocols to the workload recorded in the March, 1997 segment of the Roselli research trace. The reference system uses ideal links; the test system uses cable modems. . . . .	96
5.15	<b>Performance of Asynchronous Intents:</b> Simulating clients from the Roselli traces accessing remote storage over cable modems, we compare the behavior of synchronous and asynchronous intents. Asynchronous intents reduce the client-perceived response time. The visibility latency penalty and durability latency penalty are roughly equivalent. . . . .	98
5.16	<b>Re-ordering Data Transfers:</b> The user accesses a file set of 1000 files with size distributed as in the “large” workload. We simulate a bursty workload with 10-minute cycle time, 30-second bursty period, and 5-second average interarrival time. 20% of access are writes. The secondary device accesses every 100th file touched by the primary device. Both devices use 56 kb/s network connections. The graph plots the latency of accesses from the secondary device. . . . .	99
6.1	<b>Components of a Content-Addressable Storage System:</b> A distributed content-addressable storage system contains three primary components. Clients issue read and write requests to the system. The storage servers are responsible for actually storing the data. The query router routes client requests to the storage server that can fulfill the request. . . . .	103
6.2	<b>Use of Self-Verifying Data in CAS Systems:</b> Clients divide data into small blocks that are combined into Merkle trees. A key-verified block points to the root of the structure. To update an object, a client overwrites the key-verified block to point to the new root. ( $V$ = version, $R$ = version root, $I$ = indirect node, $B$ = data block) . . .	104
6.3	<b>Comparison of Storage Server Contents in Traditional and Extent-Based CAS Systems:</b> In traditional CAS systems, storage servers store and manage individual application-level blocks. In an extent-based CAS system, storage servers store extents, which contain a collection of application-level blocks and a certificate that identifies the owner of the data. . . . .	108
6.4	<b>The Two-Level Look-up Process:</b> In an extent-based CAS system, reading data requires a two-level look-up process. Here a client requests to read block $B1$ from extent $E1$ . First, the query router routes the request to a storage server that stores extent $E1$ . Then, the storage server extracts block $B1$ from the extent and returns it to the client. . . . .	109
6.5	<b>Procedure for Computing Extent Verifiers:</b> To compute the verifier for an extent, the system uses the recurrence relation $N_i = H(N_{i-1} + H(D_i))$ . $N_{-1} = H(PK)$ where $PK$ is a public key. . . . .	110
6.6	<b>A Comparison of the Traditional and Extent-Based CAS Interface:</b> The expanded interface of Table 6.2 provides different commands to operate on different types of self-verifying data. Commands <code>create()</code> , <code>append()</code> , and <code>truncate()</code> operate on key-verified extents while <code>snapshot()</code> converts a key-verified extent into a hash-verified extent. . . . .	111



6.7	<b>Layering the Secure Log on Top of Extents:</b> Shown here is a log that extends across four extents. The head of the log is at the left; the oldest extent of the log is on the right. The first (bottom) block in each extent is a metadata block that maintains the state of the log. The metadata block (named $M_0$ , $M_1$ , etc.) contains the extent's sequence number ( $s = 0$ , $s = 1$ , etc.) in the log, and a set of mappings to previous extents and metadata blocks in the log. . . . .	115
6.8	<b>An Example Directory Tree:</b> A simple file system used as a running example. . .	116
6.9	<b>Archiving the Initial Version of a File System:</b> (a) The back-up application translates the file system into a Merkle tree. The verifiable pointers are of the form ( <i>extent_sequence_number</i> , <i>block_name</i> ). Verifiable pointers refer to extent by sequence number because the permanent, hash-verifiable name is not known until later in the archiving process. (b) The Merkle tree is stored in two extents. The first extent, $E_0$ , is filled and has been converted to a hash-verified extent. The second extent, $H(PK)$ , is a partially filled key-verified extent. Notice that the first block in extent $H(PK)$ contains metadata including a reference to the metadata block, $M_0$ of the previous extent. . . . .	117
6.10	<b>Archiving an Update to the File System:</b> (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the extent chain after storing blocks of the updated file system. . . . .	118
6.11	<b>Components of Antiquity:</b> Each node in Antiquity serves as a storage node and a gateway for nearby clients. The administrator identifies sets of storage nodes to host extents. . . . .	120
6.12	<b>Throughput Improvement Due to Extent-Based API:</b> Client throughput increases with update size by allowing the clients to amortize the cost of producing certificates over multiple blocks of data. . . . .	122
6.13	<b>Operation Latency of Extent-Based API:</b> The CDF of operation latency for a single client through a single gateway on (a) the cluster and (b) PlanetLab. (Block Size = 4 KB, Update Size = 32 KB, Extent Size = 1 MB) . . . . .	124
7.1	<b>Components of the Moxie Prototype:</b> The Moxie prototype include an implementation of the client library and the middleware server. The server writes data to a secure log. . . . .	127
7.2	<b>The FUSE Architecture:</b> The FUSE toolkit includes a kernel module and user-level library. The kernel module receives file system requests through the VFS interface and passes the requests to the user-level library through a domain socket. The library then passes the requests to the file system implementation that has registered with the library. Results pass through components in the reverse order before reaching the application. . . . .	130
7.3	<b>Key Components in the Moxie Client:</b> The Moxie client is implemented by several by several cooperating stages. The Moxie Client Logic oversees the execution of each request, relying on other stages for handling specialized tasks. All components shown in the figure execute with a single JVM. . . . .	131

7.4	<b>Key Components in the Moxie Middleware File Server:</b> The Moxie server is implemented by a collection of cooperating stages. While the Moxie Server logic oversees the execution of each request, most of the complexity at the server serves to interact with the log. All components shown in the figure execute in a single JVM.	132
7.5	<b>Effect of Delta Encoding on Runtime:</b> Delta-encoding improves the total runtime of a workload depending on the amount of commonality in the workload and the bandwidth of the network. Comparing the copy phase of the workload shows that delta-encoding imposes little overhead when copying files into the file system. Delta-encoding decreases the runtime of the update phase by transferring fewer bytes to the server.	137
7.6	<b>Effect of Intention Updates on Runtime:</b> Intention updates reduce the visibility latency and response time by more than an order of magnitude compared to traditional, synchronous, unified updates. With intention updates, the durability latency is roughly equal to the response time of traditional updates. The current implementation introduces a slight additional delay in durability.	138
7.7	<b>Visibility and Durability Timelines:</b> The timelines illustrate how the execution of individual updates leads to the behavior reported in Figure 7.6. Each horizontal line corresponds to an update; the points on the line highlight important points in the life of the update. (a) For unified updates, the points show when the update is issued and when the response is received. (b) For intention updates, the points represent when an update is issued, when the update is made visible, and when the update is made durable.	139
7.8	<b>The Benefits of Reordering Data Transfers:</b> Intention updates afford the client great flexibility in ordering how data is sent to the server. This sharing microbenchmark shows how the server can suggest reordering to improve service for clients reading data from the server. Two users share a set of 100 8-KB files. The user constantly writes files at random; the other repeatedly reads random files. The writer is weakly connected, using a telephone modem; the reader is strongly connected.	141
7.9	<b>Comparing Moxie Against NFS:</b> With the Tcl benchmark, we compare the behavior of Moxie and NFS. The benchmark contains mostly small files which limit the effectiveness of Moxie's mechanisms. However, the benchmark demonstrates a surprising benefit from delta-encoding in the configuration and compilation phases of the benchmark.	144
7.10	<b>Operation Latency Distribution in Various Configurations:</b> The series of plots show the distribution of latency for various operations as write-back strategy and network speed vary. These plots demonstrate how intention updates reduce update latency compared to unified updates. They also illustrate the problems with Moxie's network scheduling implementation. Because it does not have direct access to the network interface's buffer, the Moxie client is not able to ensure that message priorities are strictly observed. Consequently, some high-priority messages are queued behind background transfer traffic.	146

7.11	<b>Scheduling Traffic in Moxie Versus Linux:</b> Figure 7.10 shows that Moxie is unable to enforce the priority of network messages at the application level. To estimate the performance of system that does schedule traffic properly, we modified Moxie to use the Linux traffic shaping tools to enforce message priorities. This plots shows the results of that experiment. By scheduling network traffic correctly, we reduce the runtime of the Tcl benchmark dramatically, especially at lower network bandwidths. . . . .	148
7.12	<b>Operation Latency Distribution Using Linux Network Scheduling:</b> These graphs show the distribution of operation latency in Moxie with network scheduling performed by the Linux traffic shaping tools. Compared to the graphs in Figure 7.10, these curves exhibit a much shorter tail, indicating that high-priority messages are transmitted quickly and not delayed by low-priority transfer messages. . . . .	149
7.13	<b>The Cost of Security:</b> The use of standard public key cryptography algorithms, like RSA, for signing requests can slow Moxie considerably, newer technologies, like elliptic curve cryptography, can provide similar security with only marginal cost. Clients can also protect data privacy by encrypting data with little cost in total runtime. . . . .	150
7.14	<b>Mock-up of a Graphical Interface for Managing Intents:</b> In a system with intention updates, a user may attempt to access data that is visible but not yet durable. A graphical interface, such as this, may be helpful in helping users understand this uncommon case. The exact implementation of such an interface should be guided by research in the field of human-computer interaction. . . . .	153

# List of Tables

2.1	<b>Network parameters:</b> Networking technologies exhibit a wide range of latency and bandwidth characteristics. The numbers presented in this table are meant as representative estimates, not exact performance measurements. . . . .	18
2.2	<b>Size of Traced Workloads:</b> To consider the potential benefits of various types of data compression, we captured file systems traces of two weeks of activity for two developers. This table shows the amount of number of files and amount of data written by each developer in the trace. . . . .	22
2.3	<b>Effectiveness of Compression and Delta-Encoding:</b> This table reports the percentage of bytes transferred in a system with no compression facilities that could be eliminated if compression were used. Compression could reduce update size in the traced workloads by nearly 95%. Delta-encoding was less effective, but could save as much as 80% of the update bandwidth. . . . .	23
2.4	<b>Impact of Timeout on Sessions:</b> As the defined timeout period decreases, the number of sessions identified in the trace increases. Moderate timeouts of 15–60 minutes result in reasonable mobility characteristics. . . . .	28
2.5	<b>Effect of Network Bandwidth on Conflict Rate:</b> Mobile users that rely on low-bandwidth network connections must cope with the long delays when accessing and storing data in network file systems. The latency can lead to inconsistent accesses. These tables show the number of files affected by this problem and the fraction of <code>open()</code> and <code>close()</code> operations that produce results different than those indicated by the original trace. . . . .	33
4.1	<b>Required Operations:</b> To support all of the techniques presented in this chapter, a data structure must support the operations listed above. . . . .	45
4.2	<b>Network Characteristics:</b> The simulated network connection is controlled to perform like several popular networking technologies. . . . .	56
4.3	<b>Micro-benchmark Characteristics:</b> The micro-benchmarks vary in size and in commonality between consecutive versions using the different chunking strategies. . . . .	57
5.1	<b>Network parameters:</b> Message delay is computed based on a link's latency and bandwidth. The parameters corresponding to different networking technologies are shown in the table. . . . .	88

5.2	<b>Parameters for Modeling Remote File System Performance:</b> To estimate the performance of different remote storage system protocols, we estimate the latency to access and store data in the system based on the network messages required by the protocol. Our analysis depends on the parameters define in this table. . . . .	90
5.3	<b>Computed Read Latencies:</b> The cost of accessing data depends on the protocol used by the client and whether the client already has a valid copy in the cache. For NFS, the computations apply to the <code>read()</code> operation; for other protocols based on whole-file caching, the computations apply to the <code>open()</code> operation. The cost of accessing data depends primarily on the transmission delay of the reply message, which depends on whether the server must supply the client with the requested data or merely validate the copy in the client's cache. . . . .	90
5.4	<b>Computed Write Latencies:</b> We analyze the cost of store operations using three metrics: visibility, durability, and response time. For NFS, the computations apply to the <code>write()</code> operations; for AFS-based protocols, the computations apply to the <code>close()</code> operation. For traditional approaches that use unified updates, the visibility latency and durability latency are equal. Intention updates reduce visibility latency and response time by sending only a small intent to the server to make changes visible. The costs shown for $Vis_{Coda-wc}$ , $Dur_{Coda-wc}$ , and $Dur_{intent}$ represent minimum possible costs; costs can increase if other requests stall low-priority write-back traffic. . . . .	91
6.1	<b>The CAS System <code>put()</code>/<code>get()</code> Interface:</b> First-generation distributed CAS systems use a simple <code>put()</code> / <code>get()</code> interface. The <code>put_hash()</code> and <code>put_key()</code> functions are often combined into a single <code>put()</code> function. $H()$ is a secure, one-way hash function; $h$ is a secure hash, as output from $H()$ . . . . .	103
6.2	<b>The Extent-Based CAS System Interface:</b> By extending the traditional <code>put()</code> / <code>get()</code> interface, CAS systems can support extents and two-level naming. . . . .	110
6.3	<b>Contents of an Extent Certificate:</b> The certificate stored with each extent includes fields to verify the contents of the contains and to identify the owner of the data. . .	111
7.1	<b>Extensions to the Extent-Based API for Intention Updates:</b> To support intention updates in an extent-based content-addressable storage system, we extend the API of Chapter6 with two new operations that reserve space in the extent and fill the reservation. . . . .	134
7.2	<b>Comparison of the Andrew and Tcl Benchmarks:</b> (a) The Andrew benchmark consists of five phases that populate a directory tree, access data stored in the tree, and then compile an application in the tree. (b) The Tcl benchmark consists of five phases that populate a directory tree, access data stored in the tree, and compile an application. The actual work performed in each phase differs slightly from the Andrew benchmark. . . . .	143
7.3	<b>Mean Latency of Write-back Operations with Different Security Configurations:</b> The cryptographic algorithms used by the Moxie client affect the latency of operations that change file system state. This table reports the latency of file write-back operations. Although general trends can be deduced from these measurements, interference from low-priority data transfer traffic prevents more detailed conclusions.	151

## Acknowledgments

Graduate school is far too overwhelming to tackle alone. Thankfully, I have been blessed by the support of many throughout this long journey.

My advisor, Professor John Kubiatawicz, has been the primary influence in my academic career. He has guided me and pushed me to grow and develop as a researcher. He has influenced how I approach problems and how I present the results. With his enthusiastic pitch of OceanStore, he rescued me from the computer architecture field and opened my eyes to the world of systems research. He patiently granted me the space to develop the skills and background I needed to contribute in that field. I hope to reflect his relentless curiosity.

I have also benefited greatly from the guidance of other Berkeley faculty. While serving on my qualifying exam and thesis committee, Professor Eric Brewer helped me understand how my research applied to domains that I had not even considered. Professor Ray Larson of the School of Information, who also served on my qualifying exam and thesis committee, provided valuable insight from a perspective outside the computer science department. Professor Robert Wilensky, after graciously and charitably agreeing to serve on my qualifying exam committee when scheduling difficulties threatened to derail my progress, offered valuable feedback on application requirements and expectations. Finally, Professor Mike Franklin has provided valuable advice, feedback, and perspective throughout my graduate student career.

Of course, real learning and growth happens in the trenches, and, for me, that time was spent with my peers in the OceanStore group: Hakim Weatherspoon, Dennis Geels, Sean Rhea, Byung-Gon Chun, Emil Ong, and Steve Czerwinski. The long hours that I shared with them struggling over new ideas, bickering over alternatives, and collaborating over solutions has been critical to my education.

The staff in the computer science department has helped me navigate the university's arcane bureaucracy and whimsical regulations. Special thanks go to Damon Hinson, Peggy Lau, and La Shana Porlaris.

I would be remiss if I did not acknowledge the people at my undergraduate alma mater, the University of Illinois at Urbana-Champaign (UIUC), who had such an impact on my decision to pursue graduate studies. Professor Wen-mei Hwu and his research group introduced me to computer science research. Then-graduate students David August, Dan Connors, Kevin Crozier, and John Sias shaped my early experiences in the field.

Outside of the ivy walls, I have been supported by many family and friends who have

been far more loyal than I deserve. Most important of all, my wife, Courtney Eaton, has been by my side for every step of the journey. She listened to me when I needed to talk; she encouraged me when I needed support; and she kicked me when I needed a jump-start. Perhaps most notably, she was patient when my progress was slower than I hoped. She has been the perfect companion for this trek.

The community at the First Presbyterian Church of Berkeley has become my family in California. I am especially grateful for the other couples with whom Courtney and I have enjoyed small group fellowship: the Lee's, Lee/Lawson's, Little's, Penner's, Shafer's, Seo/Rha's, and Yi's. They have been a critical support network.

My family provided the foundation that has shaped my identity and values. My parents, Greg and Susan Eaton, and grandparents, Al and Caroline Pohland and Milo and Francis Eaton, equipped me for my studies by nurturing the child's natural inclination for learning and giving me ample and various opportunities to explore. If only my parents had known what they were getting us all into when they enrolled me in a class about computers at the Children's Museum of Indianapolis at the age of five. Oh, I was so excited to share with them the flowchart that I had made. They, along with my brother David, have also been constant fans and supporters.

Lastly, during my first few years at Berkeley, I was supported by the National Defense Science and Engineering Graduate (NDSEG) Fellowship. This support helped ease my transition to a new city and a new university.

## **Part I**

# **Defining the Problem**



# Chapter 1

## Introduction

Researchers and service providers are planning, designing, building, and deploying a variety of new distributed storage systems. A feature common to many of these systems is that they aggregate storage from large collections of commodity servers. In some cases, the component machines are collocated in a single site; in others, the machines are distributed across the geographic wide-area. This approach was pioneered by a wave of academic projects that included CFS [DKK<sup>+</sup>01], OceanStore [KBC<sup>+</sup>00], and PAST [RD01]. These projects spurred many subsequent projects including the OpenDHT [RGK<sup>+</sup>05] public DHT service and the Venti [QD02] archival service. More recently, several corporate service providers have unveiled storage services, including Amazon's Simple Storage Service (S3) [Ama06] and Apple's .mac Backup Service [App06] that appear to be built on a similar set of principles.

The scalability of this new class of distributed storage systems offers a particularly attractive opportunity to provide remote storage resources to large numbers of users who previously have not had access to such services.

Remote storage systems have been widely used in corporate and academic environments for decades. Their ubiquity is a testament to the power and convenience that they provide to their users. Remote storage systems for corporate and academic settings are designed assuming that users communicate with the storage systems over high-bandwidth, low-latency links. Furthermore, it is assumed that trusted administrators maintain a tightly controlled environment free from malicious hosts, and communication is private to those outside the organization.

Users outside corporate and academic environments, however, have not typically had access to the same remote storage services. One reason for the lack of service has been a lack of demand—users outside of those environments have not had large collections of valuable data. Other

reasons were more technical in nature, such poor network connectivity and lack of features to ensure data privacy and integrity across public networks and multiple administrative domains.

We believe that the circumstances that have historically impeded the development of remote storage services for users outside of corporate and academic settings are waning. The number of potential users for remote storage services is increasing. In many parts of the world, most households own computers—the United States Census reports that 61.8% of all American households owned computers in 2003 [DJD05]. Also, improvements in mobile networking technologies are allowing both non-traditional users and traditional corporate and academic users to access network-hosted services from a wide range of locations. Finally, the demand for storage is increasing among non-traditional users as assets including personal photos, home videos, music, software, bank statements, and tax documents are increasingly transmitted exclusively in digital formats. In fact, Jim Gray of Microsoft Corporation claims that individuals now own 80% of the world's data [Gra05].

Motivated by the growing demand for storage services and the increasing size of the potential user community, this thesis examines some of the important technical challenges in making remote storage services available to a broader user community. We focus on techniques to facilitate access to remote storage systems for weakly connected and mobile users.

## 1.1 Remote Storage Systems, Features, and Limitations

A remote storage system supports access to persistent storage across a network. For users of remote storage systems, data is stored not on end host machines, but rather on a remote machine or system that is accessed across the network. When the user wishes to read data, the client machine sends a message to the remote system requesting the data. The remote storage system locates the data and returns it to the client machine via the network. When the user writes data, the client transfers the data to the storage system which stores it for later access. This approach is illustrated in Figure 1.1.

In early remote storage systems, all stored objects were accessed through the file system interface and the remote system was implemented on a single machine. These remote storage systems are often called *network file systems*, and the remote storage host is called a *file server*.

The primary features of a remote storage systems are improved data availability and durability and simpler administration. Remote storage systems enhance data availability by allowing users to access data from any machine anywhere in the network. Users need not be tied to a single physical device that stores the data, and they need not remember to copy data to portable devices

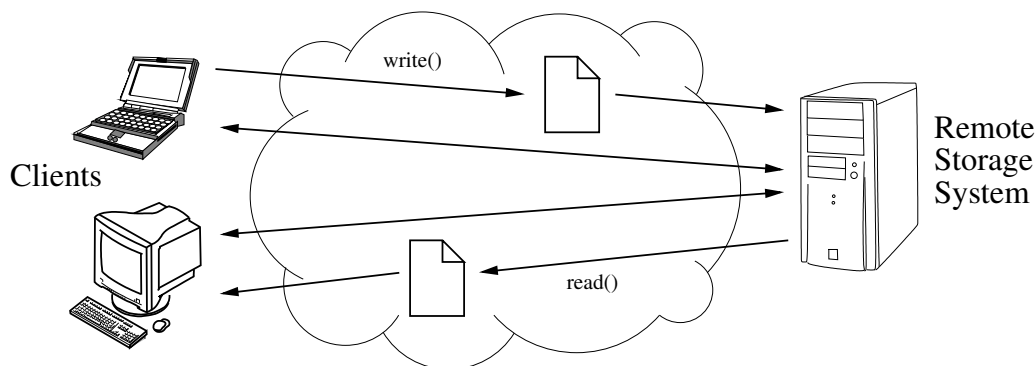


Figure 1.1: **Remote Storage System:** Remote storage systems store data on behalf of users. Client machines access data stored in remote storage systems by sending requests to the storage system via the network.

or media before traveling. Instead, they can retrieve the data they need from any machine with connectivity to the storage system. Remote storage systems also provide a convenient mechanism from sharing data with other users. Each member of a collaborative effort can retrieve the file from the storage system at any time without interaction with other members. Similarly, each member can share changes by storing a new copy of a document to the remote storage system.

Remote storage systems improve data durability in several ways. First, by storing data on machines in the infrastructure instead of end host machines, remote storage systems protect data from end host failure, loss, and theft. After replacing a failed client machine, users of remote storage systems can retrieve data from the infrastructure. Secondly, remote storage systems can improve data durability by exploiting advanced data protection mechanisms, like RAID [PGK88] or erasure coding [Rab89]. Such techniques may be prohibitively costly to implement on individual end host machines; but, because remote storage systems are typically shared resources, a remote storage system can amortize the cost of additional hardware or software over large numbers of users to make such features affordable. By taking advantage of advanced replication and coding strategies, a remote storage system can ensure better data durability than individual end host machines.

Finally, remote storage systems ease the burden of administering storage resources. Without remote storage systems, an organization must manage the storage resources on individual end host machines. This typically means that users must manage their own machines or administrators must locate and administer each machine individually. Unfortunately, it is not reasonable to expect typical users to administer their own storage resources, and it is not efficient for administrators to manage large collections of machines in this manner. Remote storage systems, alternatively, serve

as a central point where where a trained administrator can manage storage resources for a large number of users simultaneously. All users enjoy the benefits provided by patches or upgrades to storage servers. Furthermore, an administrator can back-up data for an entire organization simply by archiving the data stored on the server.

Although remote storage systems provide a number of benefits, the technology also faces several challenges. As with any shared resource, remote storage systems are an attractive target for abuse and attack. Systems must properly authenticate all users and ensure that a user's data cannot be accessed by unauthorized users.

Remote storage systems must also overcome problems with network connectivity. Fundamentally, remote storage systems must transfer large amounts data across the network; and, consequently, when clients do not have a path of strong connectivity to the system, performance can suffer dramatically. Such problems can be exacerbated if the system implements a “chatty” protocol that requires multiple round trips between client and server to respond to requests. Problems also arise when client rely on links that are of high-latency, low-bandwidth, or both. Congestion at peering points in the network or at public wireless access points can also introduce problems. We will examine these issue in detail throughout this thesis.

## **1.2 Underserved User Populations**

Historically, remote storage systems have been most widely accepted in corporate and academic environments. The reasons for this trend are both organizational and technical. Organizationally, these groups possess the financial resources to support remote storage systems. They can purchase the additional hardware required to host the remote storage systems, and they can hire competent, trained administrators to manage the systems. Technically, users in these environments typically enjoy strong network connectivity that can support good service with traditional remote storage systems. Networks in corporate and academic campuses typically support high-bandwidth communication with latency in the millisecond range. This allows client machines to communicate with storage servers quickly and transfer large amounts of data with little delay.

While remote storage systems are commonly available tools for users in corporate and academic environments, many other user communities have been noticeably under-served by the technology. In this thesis, we are particularly interested in two user groups traditionally under-served by remote storage systems: mobile users and residential users. Mobile users are those that access data from the road, exemplified by vacationers and business travellers. Residential users are

those that access data from the home, including those telecommuting to a job and end consumers.

These under-served users populations share several distinguishing characteristics. Most importantly, these users typically rely on weak connections to the larger network. The “last-mile” links at the edge of the network have high latency or low bandwidth or both. Technologies that commonly provide these “last-mile” connections include telephone modems, residential broadband (cable and DSL modems), 802.11B wireless connections, or CDMA connections. Other traits common to mobile and residential users is that they often access data from multiple devices and access points in relatively short time periods.

Providing these traditionally under-served populations with fast, reliable access to remote storage systems would grant to users many benefits presently only afforded to resource-rich users in corporate and academic settings. Mobile and residential users could access data at any time from any device. They could access data related to their job when away from the office and refer to personal data when at the office. It would provide users with durable storage for their increasingly large collections of valuable digital data including financial statements, personal photos, home videos, and purchased digital content. Finally, it would allow them to outsource the management of storage resources to professional administrators employed by their storage service provider.

The fact that, traditionally, mobile and residential users have not been well-served by remote storage systems may suggest to some that such users are not attractive targets for the technology. On the contrary, we believe that these communities are a key potential customers for the new distributed storage infrastructures currently under development. The early adopters of these technologies will likely not be the corporate and academic organizations that already have access to similar tools. Rather, the early drivers of this new storage infrastructures will be those that have the most to gain from the systems—users that currently do not have access to durable, available storage. Certainly, storage products—including external hard drives, optical disk writers, and USB drives—targeting mobile and residential users do exist. Despite the availability of these products, surveys reveal that 25% to 60% of residential users have experienced data loss [Har02, Bru01]. Users lose data because, though products exist to improve durability, they do not fit well into users’ workflow patterns; consequently, they are not regularly used and cannot provide the intended benefit. The surveys also report that up to 85% of users are very concerned about losing data. Thus, if we can improve access to remote storage systems for weakly connected users, we can extend the availability of the valuable service to users that have significant need for the technology.

### 1.3 Important Technology and Social Trends

Throughout this research, we are guided by several important technological and social trends and their consequences. We discuss these trends below.

**Improving Network Connectivity:** Network connectivity continues to improve. Decades ago, connectivity was provided by wired networks and was a privilege enjoyed by users in corporate and academic environments. Users were forced to work in fixed locations serviced by the wired network. Today, technology has extended connectivity to more users and locations. In the corporate and academic environment, wireless networks have extended the bounds of the network to include courtyards, conference rooms, and cafeterias. Telephone modems and now broadband have extended the network to residential users. As of 2003, 55% of all American households had some form of Internet access, more than double the percentage from 5 years prior [DJD05]. As wireless technologies continue to improve, users are increasingly able to access the network from neighborhood cafes and restaurants, parks and public squares, hotel rooms, and even while travelling on public transportation.

Though network connectivity is improving and available bandwidth is increasing, we do not believe that simply waiting for technology to advance will solve all problems associated with weak connectivity. On the contrary, it seems that user demand for bandwidth continues to outpace its availability. As residential users have moved from low-bandwidth telephone modems to higher-bandwidth broadband connections, the data that users own has grown also. Digital cameras now record photos in higher resolution; music and video is encoded at higher fidelity; and application file formats seem to grow with every software release.

Indeed, connectivity is improving, and available bandwidth is increasing. However, we believe that users will continue to push the limits of available networking technologies for the foreseeable future. Certainly, there will always exist pockets of users that rely on connections that are slower or financially cheaper than the average, and the techniques we consider in this thesis can benefit these populations.

**Increased User Mobility:** Taking advantage of improving network connectivity, users are increasingly likely to access data from a variety of locations. No longer confined by the wired infrastructure, users are free to work at the locations and times most convenient to them.

A 31-day trace of requests to an IMAP server at the University of California at Berkeley

reflect increased user mobility. The trace showed that users access their mail from more than 1 location per day roughly 20% of the time. Approximately 40% of users accessed the mail from at least 5 locations (as defined by IP subnet) during the trace, with some users connecting from more than 15 locations [SCJ04].

**Increased Number of Devices:** As users exhibit increased mobility, the tools to support mobile data access are becoming more powerful, affordable, and common. No longer do workers rely solely on a single desktop machine located at the office. Users are likely to have multiple desktop and laptop machines to support personal and business use. Internet-enabled cell phones, portable email devices, and personal digital assistants (PDAs) are becoming more prevalent, allowing users to reference and modify documents as they move around the city or country. Digital cameras and embedded audio recorders provide still more content creation options. Mobile devices also continue to decrease in price dramatically; in fact, some groups are working to bring the cost of a laptop down to \$100 [Chi06].

**Increased Use of Network-Hosted Services:** As the network and access to it become ubiquitous, service providers are increasingly deploying services to be accessed via the network, and users are more likely to rely on applications and services that are hosted in the network. John Gage, co-founder of Sun Microsystems, coined the phrase “the network is the computer” in observation of the emergence of distributed computing and hosted services [Ris04]. By deploying services in the network, service providers can reduce installation and maintenance costs. They can respond quickly to user needs, pushing fixes and updates to users transparently. By relying on network-hosted services managed by trained administrators, users are relieved of the responsibility of managing complex software on an array of personal devices.

## 1.4 System Model

The system architecture that we consider for our investigation is motivated by the types of storage systems that have recently appeared in the research literature and press releases. The storage systems that guide our work include research systems like CFS [DKK<sup>+</sup>01], FARSITE [ABC<sup>+</sup>02], OceanStore [REG<sup>+</sup>03], Pangea [SKKM02], and PAST [RD01] and storage services like Apple’s .mac [App06] and Amazon’s S3 [Ama06].

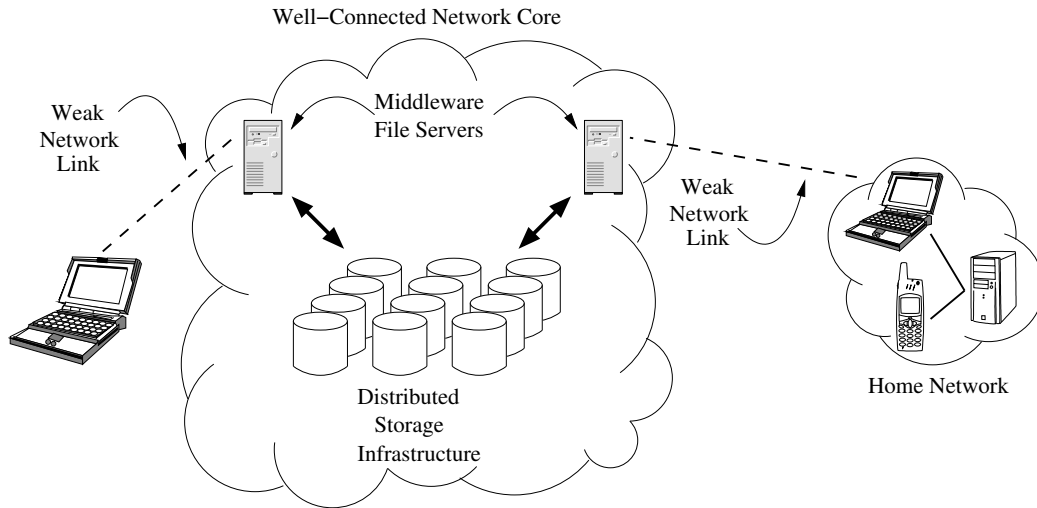


Figure 1.2: **System Model:** We assume a client-server architecture. The server resides in the well-connected core of the network. Clients connect via weak connections at the edge of the network. Client-perceived performance is primarily dependent on speed of the weak link.

### 1.4.1 System Architecture

Figure 1.2 illustrates the system architecture of our target environment. A distributed storage infrastructure in the well-connected network core provides durable, persistent storage for multiple users. We do not specify the implementation of the storage infrastructure. It could be implemented as a single, large file server, but, more likely, the storage infrastructure would be a distributed, peer-to-peer system, possibly run as a federated system by multiple independent service providers. We envision services and systems like Amazon’s S3 [Ama06], OpenDHT [RGK<sup>+</sup>05], or Antiquity [WECK07] will provide the base storage service.

Experience in the research community has shown that supporting only a simple application interface can dramatically simplify the implementation of storage infrastructures. For example, DHT-based storage infrastructures have settled on a hashtable-like `put()/get()` interface [DZD<sup>+</sup>03]. We, thus, assume that the distributed storage infrastructure will provide only limited interfaces for accessing data. To support the rich APIs that file systems and applications expect, we assume the availability of middleware servers in the network. These middleware agents translate the powerful, descriptive APIs used by end hosts to the limited interfaces provided by the storage infrastructure. The middleware servers are also located in the well-connected core of the network.

The client machines that communicate with middleware servers are not in the well-connected



portion of the network. Instead client machines interact with middleware servers in a client-server fashion via weak network links at the edges of the network. We do not consider the potential benefits of additional edge-side proxies or caches in the system.

We assume that the file servers and storage infrastructure are computationally powerful enough and, by virtue of residing in the well-connected network core, sufficiently well-connected so as to have little impact on the user-perceived performance of the system. Thus, performance is primarily dependent on the time needed to transmit messages across the weak link.

### 1.4.2 Constraints

We consider two main constraints with this model. First, the end hosts by which users access data stored in the system are weakly connected. That is, the links at the edge of the network that transmit communication from clients to the middleware are of high latency or low bandwidth or both. Due to improving network connectivity, clients are mostly connected; that is, when clients are in use, they can usually communicate with the network. Unfortunately, many times that communication must traverse weak edge links.

Secondly, we assume that middleware file servers and the storage infrastructure are operating in a fundamentally untrusted environment. This constraint differentiates our work from much of the previous research in similar areas. Our system model and target deployment environment include many threats to data privacy and integrity. All messages must traverse public networks and thus could be dropped, delayed, or re-ordered. An eavesdropper can snoop on any messages in the network. Because the data ultimately resides on storage hosted by a third-party service provider, the system must guard against data exposure and leaks by administrators of the storage service. This is no mere academic concern—news reports indicate that insiders are a bigger threat to confidential data than hackers. For example, the Ponemon Institute reports that 81% of surveyed companies have had a laptop storing confidential information lost or stolen in the last year [Pon06].

### 1.4.3 Problems

This model presents a number of problems for users of remote storage systems. The high latency of the network links connecting users to the system imposes delay on every request sent to the system. This delay can make a system appear sluggish and unresponsive to users. This problem is amplified if the client implements a “chatty” protocol that requires sending several rounds of requests across the link for a single user-level request.

The low bandwidth provided by the client's network connection also presents a problem. By its fundamental nature, a storage system must transfer large collections of data between the client and server. The time to transfer data is limited by the bandwidth of the link. Consequently, a lack of bandwidth can, again, make the system appear sluggish and unresponsive to users.

Sluggish performance due to the high latency and limited bandwidth of network can frustrate users that are forced to wait for service. To reduce the wait, some remote storage systems move network operations off of the critical path of many operations. Although this approach does improve responsiveness, it can lead to conflicts and lost data because the server does not store the most current state of the data.

Because the infrastructure that is storing the data is untrusted, the client must take extra steps to protect data. Before shipping any data to the network, the client must encrypt it to ensure data privacy. The client must also provide tokens, like digital signatures, to allow the infrastructure to authenticate its requests. Encrypting data and creating digital signatures adds further latency to each request. Mistrust of the infrastructure also limits the services that the infrastructure can perform. For example, because data is encrypted, the middleware file servers are limited in how they can manipulate the data. This restricts the interfaces that can be implemented between the client and server.

In Section 1.6, we will describe how these basic problems limit the applicability of current techniques for supporting access to remote storage systems for weakly connected users.

## 1.5 Goals

In this thesis, we investigate techniques to improve access for weakly connected clients to remote storage systems like those described in Section 1.4. By improving access for weakly connected users, we hope to allow new user groups to enjoy the many benefits of remote storage systems that have made the tools so popular in corporate and academic environments.

Consider a typical user's expectations of a remote storage system first from an abstract perspective. Perhaps most obviously, a user expects the system to exhibit good performance. Users relate performance to how quickly the system responds to requests. When reading data stored in the system, the user expects the system to return the requested data promptly. When writing data to the remote storage system, the user expects the system to respond quickly. Users are frustrated when forced to endure long delays for the storage system to service requests.

While we seek to improve performance for weakly connected users, we cannot surrender

other important features of a remote storage system. For example, the storage systems must provide support for sharing data between multiple devices and users. To protect users from accessing stale data and generating conflicting updates, a remote storage system should support a strong consistency model. While the system can be optimized for the common case that data is not shared with other users or devices, it must protect users from inconsistent accesses when sharing is present. In Chapter 2 we explain why sharing is an important issue for our target user populations.

Performance optimizations, however, must not preclude other popular storage system features. For example, versioning [MRWHZ04] is a technique which promises to facilitate tasks such as back-up, disaster recovery, and intrusion detection by maintaining a history of data stored in the system. Log optimizations were designed to improve performance by eliminating the transfer of some data across the network [KS91]. A system implementing log optimizations cannot provide versioning because versioning requires that all modifications to be shipped across the network to the server.

Finally, any technique to improve performance must be compatible with techniques to ensure data privacy and integrity. To protect privacy while data is in transit or stored in the storage infrastructure, clients must encrypt all data before transmitting it from the end host machine. To promote data integrity, all communication between principals in the system must be authenticated.

To evaluate techniques for improving performance, we must define performance in more quantitative terms. We will measure the effectiveness of our work using two notions of performance based on two types of common workloads. First, consider single isolated requests, typical of interactive workloads. A high-performance storage system will serve users accessing the remote storage system interactively by providing quick, responsive service. To measure our success, we will draw on results from the Human-Computer Interaction (HCI) research community that relate response time and human perception, as summarized by Tolia et al. [TAS06]. Researchers have learned that users perceive a system to have crisp performance when response times are less than 150 ms. As response times increase to 1 second, users become increasingly aware of the lag. When interactive response time slows to more than a second, user frustration and anxiety increase while productivity decreases. We also consider workloads consisting of long sequences of events typical of batch executions of jobs. For these workloads, we measure the performance of the storage system by how long it takes to complete the sequence of commands. That is, for batch workloads, we seek to improve overall execution time.

To improve performance as defined above, remote storage systems must carefully manage communication across the weak link that connects clients to the middleware servers. In the remain-

der of this thesis, we investigate techniques that increase the effective bandwidth and decrease the effective latency of the link. These techniques reduce the amount of time that clients must wait when communicating with the storage system. Obviously, a remote storage system cannot alter the physical characteristics of a network link, but it can improve the perceived performance of the link by altering how it uses the link. For example, the system cannot impact the true one-way latency of the link, but it can reduce the latency of a message sent across that link by reducing the amount of time a message spends in queued at the client or the size of the message. In Part II, we will present several techniques that improve effective bandwidth and latency of the network link while interacting symbiotically with requirements for consistency, privacy, and integrity.

## 1.6 Current Approaches

A large body of prior research shows how to make more efficient use of network resources while accessing remote storage systems. Although we learn and draw much from this prior work, the techniques that they describe do not provide a complete solution to meet the goals we have set forth in the environment we target.

The earliest research seeking to improve the network efficiency of remote storage systems was in caching. The Andrew File System (AFS) introduced a concept called whole-file caching with write-back on close [HKM<sup>+</sup>88]. In AFS, when an application issues an `open()` request, the client fetches the whole file from the server and stores it on the local disk. Subsequent `read()` and `write()` operations, then, can be serviced without any communication over the network. Any changes made to the file are flushed (written-back) to the server when the application issues a `close()` request. Whole-file caching reduces the number of messages and roundtrips between the client and server. Caching can also reduce the amount of data that must be transferred from the server to the client. If an object cached at one client is not changed between consecutive accesses by applications at the client, the client need not re-fetch data from the server. Instead, it can simply read the data from its local cache. Although caching can dramatically reduce the amount of downstream bandwidth used by a client, it does little to curb the upstream bandwidth required to push modifications up to the server. Note, with many asymmetric networking technologies, the upstream bandwidth is even more restricted than the downstream bandwidth.

To reduce the impact of the write-back operation, researchers explored asynchronous write-back [MES95] and delayed write-back [KS91]. To support weakly connected operation, the Coda file system responds to update requests after recording the change only in the client cache.

The change is propagated to the server asynchronously when bandwidth is available. To support disconnected operation, the Coda file system writes changes to a local log instead of propagating the modifications to the server immediately and synchronously. When connectivity is re-established, the changes in the log are flushed to the server. To reduce the amount of data that must be flushed, the Coda client attempts to optimize the log by eliminating from the log changes that were later overwritten or reversed. Although asynchronous write-back and delayed write-back do optimize the use of the uplink, the state recorded at the server is stale, lagging the state at the clients. Stale server state leads to consistency problems that complicate data sharing. Furthermore, in systems that implement log optimizations, the server does not view a complete history of operations, precluding support for features like versioning.

Other remote storage systems use surrogates to overcome weak client connectivity. For example, Lee et al. [LLS99] describe a system in which clients ship operation-based updates to surrogates in the network with a strong network connection to the server. The surrogate performs the operation (with communication with the server) and sends the response to the weakly connected client. The originating client verifies that the results of the surrogate's execution matched the result of the local execution, falling back to other means if it does not. With this approach, most communication occurs between strongly connected principals. Techniques that employ proxies and surrogates provide benefits at the cost of additional infrastructure. Also, proxies are surrogates are often trusted components; if they were to be part of the untrusted infrastructure, the types of operations that they could support would be limited by the need to encrypt data and provide authentication tokens on behalf of the client. While our work compliments proxy-based approaches, we seek techniques that work in the strict client-server model embodied by our system model.

Finally, other systems have explored the potential benefits of using delta-compression on file system traffic. The Low-Bandwidth File System [MCM01] conserves bandwidth between a client and server by exploiting commonality between multiple files and versions in a file system. Before transferring an object, the sending principal first sends a description of the blocks that comprise the object. The receiving principal compares the list of component blocks against those that it already stores and requests that only the new blocks be transferred. When a file to be transferred shares many blocks with other file or versions, the amount of bandwidth required can be much lower. Delta-compression can improve the utilization of both the uplink and the downlink. However, current implementations of delta-compression require trusted servers that can arbitrarily manipulate data in plaintext form. Our target environment does not support such schemes. Also, delta-compression only benefits workloads that transfer lots of similar data. Workloads composed

of mostly unique data, like content-creation workloads, do not benefit; and thus, delta compression is not a complete solution.

## 1.7 Contributions

This thesis makes the following contributions to the study of how to improve access to remote storage systems for weakly connected clients.

**Identifying the Challenges of Weak-Connectivity:** In the next chapter, we discuss the problem of access to remote storage for weakly connected clients in detail. We also present the results of several studies that help us identify some of the individual issues in this large problem.

**Techniques for Improving Access for Weakly Connected Clients:** In Part II (Chapters 4–6), we describe and evaluate three techniques that improve access to remote storage systems for weakly connected clients by improving the effective latency and bandwidth of the weak network link.

- **Privacy-Preserving Delta-Encoding:** In Chapter 4, we describe a technique that we call privacy-preserving delta-encoding that allows weakly connected clients to enjoy that benefits of delta compression even when the server is not trusted with data in plaintext form. Principals in the system share a data structure that indexes a set of blocks containing ciphertext. To modify data, clients send the server an update which contains a set of manipulations to perform on the data structure and blocks of new data in ciphertext form. The server can apply the changes to produce a new version of the stored object. With privacy-preserving delta-encoding, a system can match the current state-of-the-art in delta-encoding for storage systems even in environments where servers are not trusted with plaintext data.
- **Intention Updates:** In Chapter 5, we describe intention updates, a write-back strategy that enables the client to control the use of the network connection by prioritizing and scheduling messages. Intention updates allow us to consider a variety of link scheduling optimizations. Intention updates empower the client to control the use of its network connection by splitting the write-back operation into two phases. In the first phase, the client sends a small message that describes the changes that are to be made. In the second phase, the data performs the bulk data upload to the server. By splitting write-back into two-phases, modifications are made visible quickly and the server reflects the current state of the data accurately while the client

can schedule the majority of the network use based on the local availability of resources. The approach allows the service to respond quickly to individual user requests while protecting users from inconsistent accesses when sharing data.

- **Two-Level API for Content-Addressable Storage:** In Chapter 6, we define a two-level API that allows content-addressable storage systems, like those that provide the backing store for scalable, wide-area remote storage systems, to support file system workloads more efficiently. By aggregating many small blocks of data typical of those created by a file system workload into larger collections, the API enables the design of storage systems that better amortize the cost of communication, repair, update, and access.

**Putting it All Together:** Finally, in Part III, we show how these techniques can be blended together in a single system. We describe a prototype implementation and evaluate how the system is able to meet the goals outlined in Section 1.5.

We note that other interesting challenges relevant to this problem domain are specifically *not* addressed in this thesis. In these area, we believe that existing work in literature can be applied. For example, we do not consider techniques to reduce cache validation latency for weakly connected clients. Instead, we refer the reader to the variable latency cache coherence mechanisms of Coda [MS94]. Similarly, we do not consider the speculative use of link bandwidth for prefetching data. A rich body of literature on the subject already exists which is compatible with the techniques we discuss in this thesis [SSS99].

## Chapter 2

# Understanding the Problem

In this chapter, we discuss several topics important for gaining a better understanding of the problem domain. In Section 2.1, we consider network characteristics that impact remote storage systems and provide an overview of several relevant networking technologies. Section 2.2 defines metrics important in our study of remote storage systems. In Section 2.3, we study the potential effectiveness of various forms of data compression in reducing the bandwidth consumed by remote storage systems. Finally, in Section 2.4, we investigate how the mobility of users affects interaction with remote storage systems.

### 2.1 Weak Networks

The general level of service that can be expected of a network link can be characterized by its latency and bandwidth. The one-way latency of a link is defined as the time between the start of a message transmission and the start of a message reception. That is, latency measures the time to send the first bit of a message across the link. We typically measure latency in milliseconds. The bandwidth of a link describes how much data it can transmit in a given time interval. The bandwidth of a link determines a message's transmission delay, the time between the receipt of the first and last bit of a message. We typically measure bandwidth in bits per second.

The bandwidth and latency of a link contribute to the response time of a request as defined in Equation 2.1. The subscripts  $c \rightarrow s$  and  $s \rightarrow c$  represent the direction of message transmission between client and server. Some networking technologies, called asymmetric technologies, have different performance characteristics on the uplink and downlink.



Network Type		Latency (ms)	Bandwidth (kbit/s)
Ideal		0	$\infty$
Ethernet	100 MB	1	100,000
	1 MB	10	1,000
Cable Modem	current	30	6144 down/384 up
	legacy	30	384 down/128 up
Wi-Fi	802.11G	10	54,000
	802.11B	10	11,000
Cell Modem	EV-DO	100	2048 down/384 up
	GSM	100	9.6
Telephone Modem		100	56

Table 2.1: **Network parameters:** Networking technologies exhibit a wide range of latency and bandwidth characteristics. The numbers presented in this table are meant as representative estimates, not exact performance measurements.

$$\begin{aligned}
 \text{ResponseTime} = & \text{Latency}_{c \rightarrow s} + \text{TransmissionDelay}_{c \rightarrow s}(\text{req}) + \\
 & \text{ProcessingTime}_s + \\
 & \text{Latency}_{s \rightarrow c} + \text{TransmissionDelay}_{s \rightarrow c}(\text{resp})
 \end{aligned} \tag{2.1}$$

An *ideal* network link transmits messages instantaneously, with no delay. Such a link can be said to have zero latency and infinite bandwidth. With ideal links, the response time for a request is solely dependent on the processing time at the server. It can be useful to simulate systems with ideal network links to determine the best possible performance of the system.

Unfortunately, true ideal links do not exist. At the very least, the speed of light determines the minimum possible latency for a network link. Real networking technologies have varying latency and bandwidth characteristics. Table 2.1 enumerates several popular networking technologies and their approximate performance characteristics. Ethernet is commonly used to connect hosts in local area networks (LANs). Its low latency and high bandwidth make it popular in well-connected environments like corporate offices and universities. Newer varieties of Ethernet boast bandwidths of 1 and even 10 GB/s.

The other technologies listed in Table 2.1 are more indicative of the type used by our target user populations. The cable modem is a popular “broadband” connectivity option for residential users. Note, it is an asymmetric technology offering more bandwidth in the downstream direction.

Unfortunately, broadband connectivity is not an option for all users. Many users in rural areas, or those looking for cheaper connectivity options, rely on dial-up access using telephone modems. The latest cable modem infrastructure promises residential users 6 Mbit/s downlink and 384 kbit/s uplink bandwidth. While such infrastructure provides relatively ample bandwidth for downloads, the uplink is even more restricted relative to the downlink (16x).

Several wireless technologies are popular among mobile users. The 802.11B Wi-Fi standard provides connectivity for users within approximately 100 feet of an access point. Although the standard claims to support bandwidth of up to 11 MB/s, congestion can dramatically decrease actual service levels [HRBSD03, XBLG02]. Newer versions of the Wi-Fi standard promise higher bandwidth. When an access point is not near, users can access the network through cellular modems. Common standards for cell phone access include GSM and EV-DO. While some standards provide less bandwidth than even telephone modems, others promise bandwidth comparable to broadband technologies.

## 2.2 Metrics

In this thesis, we will use a variety of metrics in evaluating the nature of the problem and the efficacy of the proposed techniques.

*Response time* is one of the most commonly used metrics in the study remote storage systems. The response time measures the time between when a client issues a request and receives a response. Local processing, network latency, transmission delay, queueing in the network and at the server, and processing at the server all contribute to response time. The response time metric is relevant to all types of requests. A lower response time is always better.

For requests that modify state recorded in the remote storage system, we consider two additional metrics. The *visibility latency* measures the time until an update is visible; a modification is said to be *visible* when other clients can observe that the change has occurred. For an update to be visible, the existence of that update must be recorded in some place where other clients are guaranteed to notice it. Because we limit our investigation to client/server architectures, information about a change must reach the remote storage service to be visible. The *durability latency* measures the time until an update is durable; a modification is said to be *durable* when other clients can retrieve the results of the modification. Again, because we consider client/server architectures, a change must propagate to the storage system to be durable. The visibility latency and durability latency metrics are inspired by the observations of Kim et al. [KCN02] differentiating between the

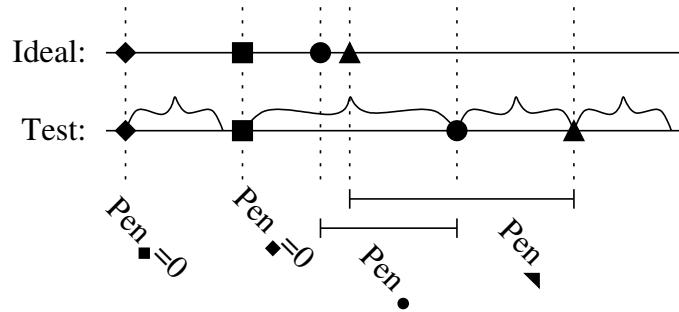


Figure 2.1: **Issue Time Penalty:** The issue time penalty is the difference between the time an event is issued in two systems. Here we compare an “ideal” system that services requests with zero latency to a realistic system. This measurement rewards systems that are responsive enough that they do not block user requests.

safety (durability) and visibility of updates. Lower visibility and durability latencies are better. Most systems do not exploit the distinction between visibility and durability; on those systems, an update’s visibility latency and durability latency would be the same.

Response time, visibility latency, and durability latency measure the absolute performance of a single system. Throughout our investigation, we will use simulations that allow us to observe the behavior of multiple storage systems to identical workloads. To compare the performance of two systems, we use the relative versions of the metrics described above. The *response time penalty* is the difference between when the client receives a response to a request in the two systems. The *visibility latency penalty* measures the difference between when an update becomes visible in the systems, and the *durability latency penalty* measures the difference between when an update becomes durable in the two systems. Lower penalties are better.

Relative metrics are cumulative. In simulations, an event is not issued until the prior events (on which the event is assumed to depend) are handled. Events are never issued faster than the trace defines. Delay caused by slow service of earlier requests accumulates and may impact the penalty attributed to later events.

Another metric for comparing two systems is the *issue time penalty*, defined to be the difference between when an event is issued on the two systems. By studying the distribution of the issue time penalty for events, we can compare how much time a client spends waiting using different protocols. Figure 2.1 illustrates the issue time penalty for a workload with four requests. To score well, a system needs to be able to serve user requests at the rate the user issues them. This metric penalizes systems that block users attempting to make a request.

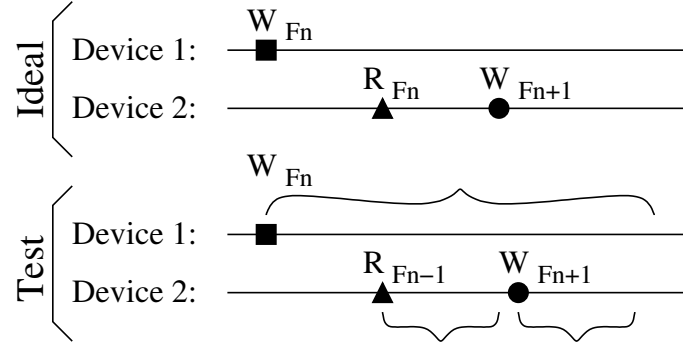


Figure 2.2: **Conflicts:** Conflicts occur when requests in a real system reach the server in an order that differs from the execution order of an ideal system. In this example, in the “ideal” system, write  $W_{F_n}$  replaces  $F_{n-1}$ ; later, another user issues read  $R_{F_n}$  to read that file and write  $W_{F_{n+1}}$  to update the file. In a system constrained by weak networks, a user issues  $W_{F_n}$  but transferring the data to the server is slow. When another user reads the file  $R_{F_{n-1}}$ , the server returns the last known version  $F_{n-1}$  resulting in a conflict. The subsequent write  $W_{F_{n+1}}$  results in another conflict because it overwrites a version,  $F_{n-1}$ , different from the ideal case. When the initial write,  $W_{F_n}$ , finally completes, it creates another conflict because again the wrong version,  $F_{n+1}$ , is overwritten.

Finally, we measure the number of conflicting accesses, or *conflicts*, allowed by a storage system. We say that an event results in a conflict if it observes different outcomes on the system under test than it would on an “ideal” system that services requests infinitely fast. Differences in outcome are typically caused when, in workloads that exhibit sharing, requests to the same object by different clients are delayed in the queue at their respective clients for different amounts of time. This causes the events to be executed in the test system in an order different than in the ideal system. Figure 2.2 illustrates a scenario in which several conflicts occur.

## 2.3 Potential for Data Compression

Many applications that must transfer large amounts of data across the network rely on various data compression techniques to increase the effective bandwidth of the network links. In this section, we present the results of a small study we conducted to investigate the potential for data compression for remote storage systems. With this study, we seek to understand the general trade-offs of various types of compression. This understanding will guide us in developing techniques to exploit data compression to improve access for weakly connected clients.

To drive this investigation, we collected a two-week trace of actual file system traffic from developer workstations. The workload was captured by a custom-built tracing library that logs, in

Trace	Files Written	Unique Files Written	Total Write Size (MB)
Developer 1	1246	454	79.1
Developer 2	528	121	8.5

Table 2.2: **Size of Traced Workloads:** To consider the potential benefits of various types of data compression, we captured file systems traces of two weeks of activity for two developers. This table shows the amount of number of files and amount of data written by each developer in the trace.

addition to traditional information like system calls and their parameters, the actual data that is written to disk. Although many file system traces exist in the literature (e.g. [Har93, RLA00]), we are not aware of any publicly available traces that record the actual data that was written to disk.

The tracing library relies on the operation of the dynamic linker/loader. Modern operating systems use a linker/loader to locate and link the implementation of shared library functions to executables dynamically at run time. The Linux dynamic linker/loader allows users to customize the path that is searched for library implementations. We use this feature to interpose our library between the application and the standard library implementation.

The tracing library provides an implementation of all system calls that affect the file system such as `open()`, `close()`, `read()`, `write()`, `rename()`, and `unlink()` as well as the `printf()` family of functions. By registering the tracing library with the linker/loader, the library can observe all system call activity related to the file system. After logging the system call, the trace library forwards the request to the standard shared library for servicing. In this manner, applications run unmodified.

We implement several optimizations reduce the size of the trace log and protect sensitive data. For example, we only log system calls that operate on 'regular' files, as identified by the `stat()` system call. This eliminates from the log interprocess communication and all activity to virtual file systems and device nodes. Additionally, the user can specify a set of files or directories to blacklist. This allows the user to protect sensitive files, such as cryptographic keys used by SSH and PGP, and files not relevant to the trace, such as files in a web browser cache.

We collected traces of file activity on two Linux workstations used by researchers for two weeks from August 1-15, 2004. Table 2.2 summarizes several key metrics of the traces. Developer 1 was primarily engaged code development using Java and Perl and document preparation using the Latex Document Preparation System. Developer 2 was primarily engaged development using Java and Ruby. While the length and size of the traces is not particularly large, the traces do enable us to compare the effectiveness of different types compression using real workloads. We consider this to

Trace	Compression		Delta-Encoding			
	Speed	Size	Whole	Block	Rabin	Rsync
Developer 1	93.4%	95.1%	4.5%	78.5%	78.8%	81.1%
Developer 2	94.8%	94.3%	0.1%	14.3%	15.3%	21.4%

Table 2.3: **Effectiveness of Compression and Delta-Encoding:** This table reports the percentage of bytes transferred in a system with no compression facilities that could be eliminated if compression were used. Compression could reduce update size in the traced workloads by nearly 95%. Delta-encoding was less effective, but could save as much as 80% of the update bandwidth.

be a valuable evaluation tool compared to the micro-benchmarks used to evaluate other research in similar areas.

For this study, we traced all activity directed at the user’s home directory. We pruned all requests to temporary files and all accesses related to web browser activity. We assert that such files should be stored on a local file system, even if a remote storage system is available. We identified temporary files as files that were created and later removed by the same process.

With these traces, we then measure the potential bandwidth savings of traditional data compression and several form of delta-encoding. To begin, we measure the amount of data that must be transferred from the client to the server to record the changes made at the client in a system with no compression. We compare that result to the amount of data that would be transferred if various forms of compression were used. Table 2.3 reports the results of these measurements, showing the percentage of total bytes transferred in the base system that could be eliminated if compression were used.

Data compression is the process of transforming data so that it can be represented in fewer bits than the original data. To measure the bandwidth savings characteristic of traditional data compression algorithms, we use the zlib compression library [IGA06]. We consider two variations of the zlib algorithms: one optimized for computation speed and one optimized for message size. Measurements reveal that simple compression can reduce the amount of data written back to the server by approximately 93–95%.

Another technique that is sometimes used to compress file system traffic is delta encoding. Using delta encoding, a storage system transfers only changes (or differences or deltas) to a file instead of the complete file. Different delta encoding algorithms can detect varying amounts of commonality between files. We will discuss delta encoding in detail in Chapter 4. Briefly, whole-file encoding recognizes commonality only when the entire file is unchanged. Block-based delta encoding divides a file into fix-sized blocks and searches for blocks that are unchanged. Rabin-

based encoding uses Rabin fingerprints in a novel manner to divide a file into variable-sized blocks and identifies blocks that are unchanged. Finally, rsync (a program for synchronizing data across devices) uses a form of delta encoding that can identify unchanged data even when it does not fall on a block boundary. The results show the potential bandwidth savings due to delta encoding varies widely. It seems, however, that the block-based, Rabin-based, and rsync schemes perform competitively. We discovered that the workload from “Developer 2” shows significantly less commonality because of several `cvs checkout` operations that wrote a large number of files. When these files were written, the file system did not contain a prior version which could contain unchanged data.

These measurements show that, for developer workloads, traditional data compression can reduce file size by roughly 95%. Data compression was effective for both workloads that we traced. The effectiveness of delta compression varied more depending on the workload. For the “Developer 1” workload, delta compression reduced file size by almost 80%; however, for the “Developer 2” workload, the technique was much less effective, reducing file size by only about 15%.

We also measured the bandwidth savings that could be achieved by combining traditional compression with delta-encoding. To combine the two approaches, we assume that the client first computes a delta-encoded update, and then compresses the data block in each delta before transmitting it across the network. Not surprisingly, compressing deltas reduces bandwidth requirements further. For example, using Rabin-based delta-encoding on the “Developer 1” workload, the potential bandwidth savings increased from 78.8% to 91.7%. Note, that the combination of delta-encoding and compression conserves less bandwidth than compression alone. We suspect this is the case because compression is performed at the block level instead of the file level. When we attempted to perform this analysis for the “Developer 2” workload, we discovered that the trace file had been corrupted and could not be recovered.

The workload type can have a significant impact on the effectiveness of different compression techniques. For example, we would expect that developer workloads, like the ones used here, would benefit from compression techniques because they handle mostly human-readable text. Other workloads that access binary files (like office productivity applications) or compressed file formats (like images) would likely benefit less from traditional compress techniques.

## 2.4 Impact of Mobility

To understand the problem domain better, we also studied the demands that our target user populations place on remote storage systems. We are especially interested in understanding

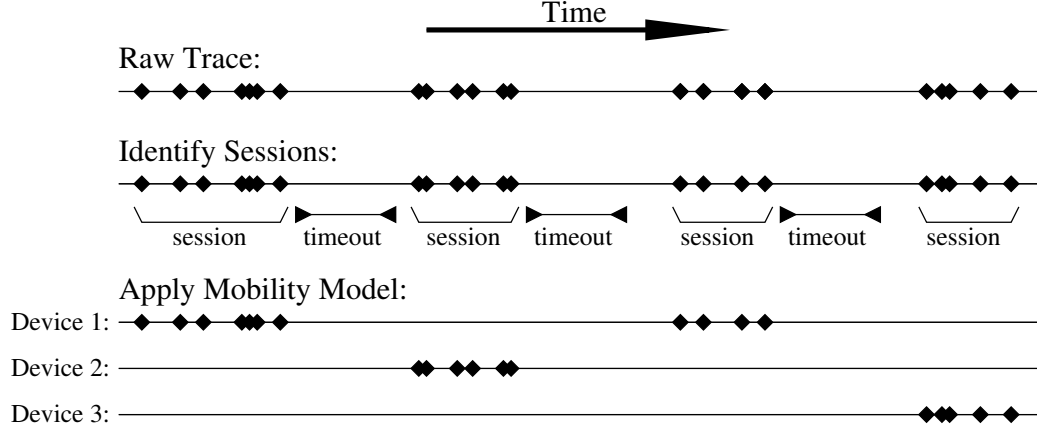


Figure 2.3: **Modelling Mobile User Workloads:** To synthesize workloads of mobile users, we augment traditional file system traces to reflect accesses from multiple locations and devices. We begin by identifying *sessions*, sequences of operations with no idle period longer than the *timeout* period. We then apply a mobility model to assign the sessions to different devices.

how those workloads differ from those of traditional users. Toward this end, we performed a study investigating the demands that mobile users make of remote storage systems.

Lacking traces of real mobile users accessing remote storage systems, we synthesize mobile user workloads by augmenting traditional traces of file system activity. We then explore, through simulation, the interaction between mobile users and remote storage systems, considering how mobility affects workload characteristics, cache management traffic, performance, durability, and consistency.

### 2.4.1 Synthesizing Mobile Workloads

Ideally, in our study, we would use traces of activity from a deployed remote storage system with real mobile users. Unfortunately, we know of no such available traces; in fact, we are unaware of even any such system on which we could collect traces. Instead, we synthesize workloads of mobile users from existing file system traces in the literature (e.g. [RLA00, Har93]). The process converts traces from fixed-location desktop machines into traces that reflect users accessing data from multiple devices.

The procedure for synthesizing workloads is based on the concept of user sessions. A *session* is a sequence of operations from a single user in which no two consecutive events are separated in time by more than a specified *timeout* period. Similar notions of a session have been used in other studies [KE02, SCJ04]. We assume that all operations in a session are performed on



the same device.

After dividing a trace into sessions, we apply a *mobility model*. We assume that each user owns a collection of one or more devices and that the user accesses data from a single device at a time. A mobility model defines how to assign the sequence of sessions to the user's collection of devices. Figure 2.3 illustrates our this approach.

By varying session identification criteria and the mobility model, this procedure creates workloads with varying characteristics. For example, by varying the session timeout period, the procedure divides the original trace into different sessions. A short timeout period (15 minutes) can be used to model a very dynamic user population that is prone switch to a different device after even short idle periods. A long timeout period (4-6 hours) models users who tend to work from a single location and device for long periods of time. A long timeout period might be used to model an employee that uses a different machine depending on whether she is working from home or the office on a given day. Section 2.4.3 shows in more detail how the timeout affects the synthesized workload.

The mobility model also helps define the resulting workloads. It can reflect users that rely on a single device even as they move among different locations, or it can model users that switch among devices freely as they move about. Section 2.4.3 shows the the mobility model plays a significant role in determining the effectiveness of caching.

One limitation of this model is that it assumes that users access data through only a single device at any one time. We know this assumption to be false, though we believe it to be true in most cases. We have spoken with some users who use multiple devices in a single session. For example, one user described using one machine with a preferred user interface for editing code and documents and another machine with more memory and a faster processor to compile or render the job. In another scenario, a user may access scheduling information on a personal organizer and a desktop machine simultaneously. Our approach to modelling mobile users does not reflect these user patterns; however, we can see the effects of rapidly switching between devices by defining a very short (60 second) timeout period.

## 2.4.2 Simulation Approach

Given this procedure for synthesizing traces, we now seek to identify the needs of mobile users accessing a remote storage system. Toward this end, we have built a simulator to model interaction between a user and a network file system. The simulator models an AFS-like protocol for

communication between clients and servers [HKM<sup>+</sup>88]. A key feature of the AFS protocol is that clients communicate with the server only on `open()` and `close()` operations. On `open()`, a client fetches the entire file and caches it locally. Subsequent accesses to the file are satisfied from the local cache, and write operations modify the file in the client cache, but not on the server. Changes are propagated to the server at the matching `close()` operation. It is said that the client performs whole-file caching with write-back on close. The write-back operation can be performed synchronously (as in AFS [HKM<sup>+</sup>88]) or asynchronously (as in Coda’s weakly connected mode [MES95]). When a file is modified, the server performs a callback to invalidate all copies cached at clients currently connected to the server. If a client loses its connection to the server, it invalidates its entire cache and must re-validate each file on its next access.

We assume an AFS-like protocol because such protocols have been shown to be well-suited for network file systems that serve regions larger than just the local area [HKM<sup>+</sup>88]. By requiring communication only on `open()` and `close()` operations, the AFS protocol reduces the number of interactions between the client and server. Also, although the AFS protocol does not match UNIX-semantics, it has been reported that most applications work atop AFS without modification [HKM<sup>+</sup>88].

Though connectivity is improving, many mobile users still rely on connections that are weak compared to the LAN links traditionally used to access the file server. Thus, it is important to model the impact of the weak link. We define communication latency between a client and server by the familiar  $\alpha + \beta \cdot n$  model where  $\alpha$  corresponds to the latency of the link separating the two machines,  $\beta$  corresponds to the link’s bandwidth, and  $n$  is the size of the message in bytes.

The workloads used to drive the simulator are synthesized from the Roselli traces [RLA00]. We use a 31-day segment of the “research” trace recorded in March 1997. This portion of the trace contains activity for 103 unique users. The session identification criteria and mobility models used are described along with the simulation results.

### 2.4.3 Workload Demands of Mobile Users

To understand the workload imposed by mobile users, we investigate the characteristics of sessions produced by our procedure, the amount of network traffic required to maintain the caches on multiple devices, and the performance, consistency, and durability observed by mobile users.

Timeout	Total Sessions	Sessions/User/Month		
		50%	75%	95%
1 min	85,003	18	170	5310
5 min	38,114	13	87	3453
15 min	4,206	6	39	226
60 min	2,373	3	26	95
240 min	1,221	3	18	44

Table 2.4: **Impact of Timeout on Sessions:** As the defined timeout period decreases, the number of sessions identified in the trace increases. Moderate timeouts of 15–60 minutes result in reasonable mobility characteristics.

### Session Characteristics

First, we examine the traces synthesized by the procedure of Section 2.4.1. Table 2.4 shows how the timeout period affects the number of sessions in the trace and the number of sessions for any single user. We measure the total number of sessions identified in the trace and the number of sessions per user. The table reports three points from the session-per-user distribution. The sessions-per-user distribution is skewed by a number of users that appear in the trace for only a short amount of time. In the segment of the trace we use, 21 users (almost 20% of all users) are active for 15 minutes or less during the entire trace.

Very short timeout periods of 1 or 5 minutes produces a very large number of sessions—5% of users have more than 100 sessions per day using a 5-minute timeout. Although very short timeouts may be occasionally appropriate to model users working on multiple devices at one time, such aggressive timeouts should not be used for generating long workloads because they overstate potential client mobility. More moderate timeout values of 15 to 60 minutes reflect potential mobility better. With a 15-minute timeout, 5% of users have roughly 7 or more sessions per day. Applying a longer timeout of 240 minutes to the trace, most users have only a few sessions per month.

Figure 2.4(a) shows the cumulative distribution function (CDF) of session duration as the timeout period varies. Focussing on the curves corresponding to moderate timeout values, we see that most sessions are less than a few tens of minutes in length. A few sessions last thousands of minutes. Assuming that real users do not continuously initiate traffic for hours on end, these long sessions indicate that the original traces contain some traffic from applications that run periodically or unattended. Unfortunately, given the information in the traces, we are unable to identify these operations that are initiated without user interaction.

Figure 2.4(b) shows the CDF of the number of operations per session. An operation is

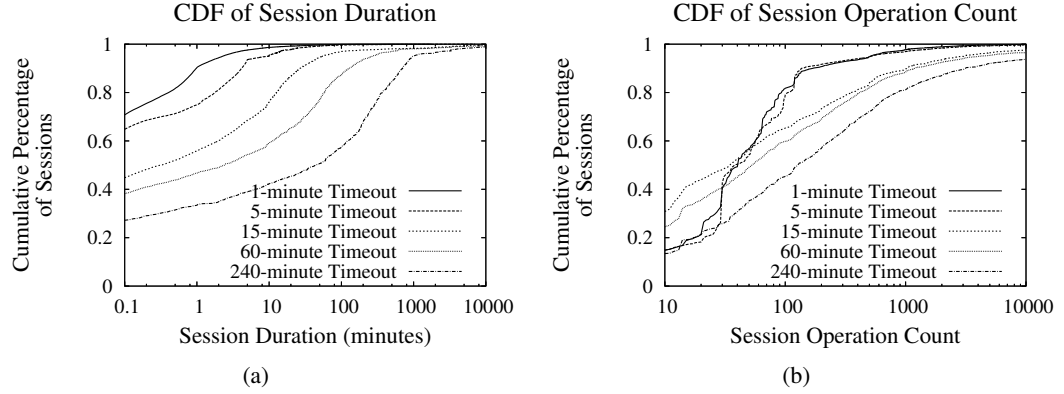


Figure 2.4: **Session Characteristics:** (a) The cumulative distribution function of session duration as the session timeout varies. (b) The cumulative distribution function of session operation count as the session timeout varies.

a system call such as `open()`, `close()`, `read()`, or `write()`. The operation count is a rough indication of the load during a session. Most sessions contain only a few operations. Even with a 4-hour session timeout, roughly 80% of sessions contain less than 1000 operations. With a 15-minute session timeout, 90% of sessions contain less than 1000 operations. Because most sessions contain few operations, systems that attempt to predict future access based on user requests will likely have difficulty adapting before the user moves to a different location or device. Interestingly, we found little correlation between the duration of the session and the operation count of the session.

### Cache Management Traffic

Next, we consider how moving between multiple devices impacts cache management traffic. Seminal studies (e.g. [OCH<sup>+</sup>85, NWO88a, BHK<sup>+</sup>91]) have shown that caching can greatly reduce the amount of data that must be transferred from the server to a client in traditional workloads. In a caching system, the client stores a copy of a file locally after retrieving it from the server. When an application accesses that file in the future, the client can satisfy the request locally, without contacting the server. Effective caching can benefit mobile users by reducing the amount of traffic sent across the network connection which may have little bandwidth or expensive per-byte usage fees.

To prevent users from accessing stale data, however, the system must ensure that data in the cache is up-to-date. If the data was changed by another user, the client must re-fetch the data from the server before access. Because many traditional workloads exhibit little sharing [KS91],

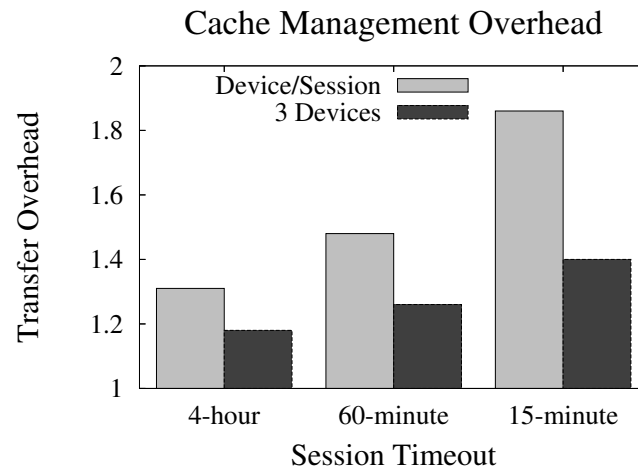


Figure 2.5: **Impact of Mobility on Cache Management Traffic:** The amount a cache management traffic needed to ensure that all devices access up-to-date varies with the defined timeout period. The amount of traffic is normalized by the amount of data transferred when all traffic for a given user originates from a single device.

clients must re-fetch data from the server only occasionally, and the reduction in traffic is substantial.

The contents of a device’s cache can be invalidated not only by another user modifying shared data, but also by a single user modifying data on a different device. When a user makes changes on one device, data cached at other devices must be invalidated and re-fetched upon next access.

We measure the impact of mobility on cache management traffic for two different mobility models. In the first, each user depends on a large number of devices, using new, unique device for each session. This model represents a worst-case scenario requiring data to be transferred to each new device as it is used. In the second model, each user cycles through a set of three devices, changing devices for each session. We measure the number of bytes transferred from the server to the client in each scenario. We report the amount of cache management traffic normalized by the amount of traffic that would be generated if each user performed all accesses from a single device.

Figure 2.5 shows the results of these simulations. In the worst-case, when users switch to a new device for each session, the overhead of cache management traffic varies between 31% and 86%, depending on the assumed session timeout. Using a more realistic mobility model, the cache management overhead decreases, but can still be significant. With a 60-minute timeout, the cache management traffic is 26% more than in the base case; with a 15-minute timeout, the overhead is 40%.

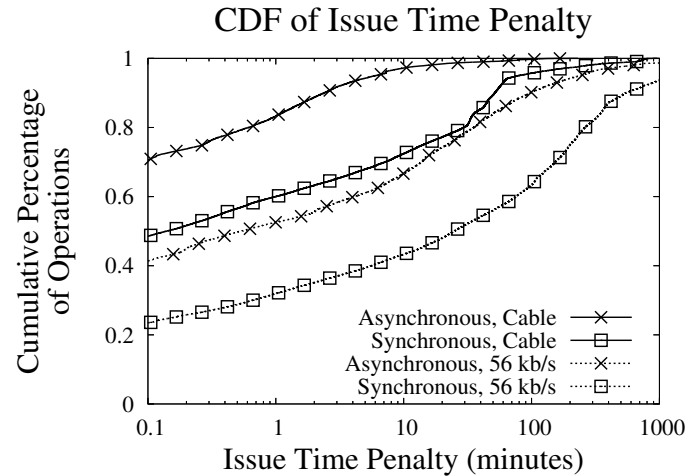


Figure 2.6: **Impact of Weak Connectivity on Performance:** The issue time penalty, reflecting user-perceived slowdown, of clients accessing remote storage using synchronous and asynchronous write-back.

### Performance, Durability, and Consistency

Next, we consider the performance, durability, and consistency seen by mobile users across a range of network technologies. We simulate four types of network connectivity: a cellular modem with 10 kb/s bandwidth, a telephone modem with 56 kb/s bandwidth, a cable modem with 128 kb/s upstream and 384 kb/s downstream bandwidth, and a 1 Mb/s connection. Workloads were generated using a 15-minute timeout period and assume that each user rotates between three devices.

We measure performance using the issue time penalty. The first event in each session is issued at the same time on the ideal and test system; thus the first operation in a session always has a issue time penalty of zero. A slow operation can only delay other operations *within* a session.

In addition to measuring the performance of an AFS-like system, we also measure the performance of a system that implements Coda's mechanisms to support weakly connected clients [MES95]. The designers of Coda recognized that write-back can be very expensive, especially for weakly connected clients, because it requires that clients transfer modified files across the network to the server. To reduce the issue time penalty due to write-back, when using Coda in weakly connected mode, data is uploaded to the server asynchronously. After issuing a write request, the user can continue working without delay.

Figure 2.6 plots the CDF of issue time penalty for clients implementing synchronous

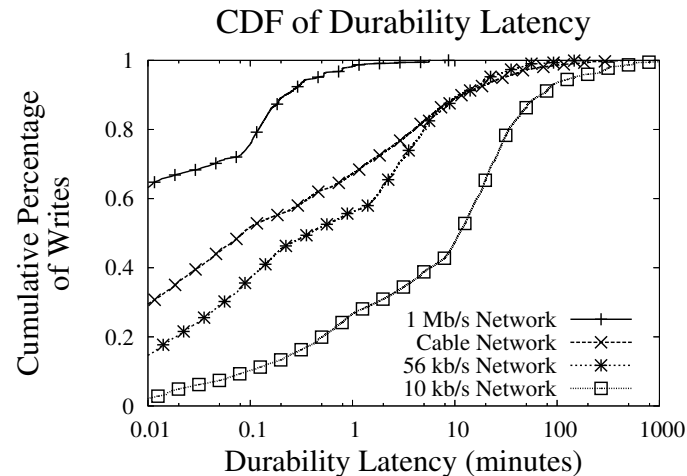


Figure 2.7: **Impact of Weak Connectivity on Durability:** The durability latency measures the time until data is stored durably at the server using asynchronous write-back.

(AFS-like) and asynchronous (Coda-like) write-back. For clarity of presentation, we present results for only two types of networks; the trends, however, also applied to faster and slower networks. For clients implementing synchronous write-back, the issue time penalty is several minutes when using a cable modem and several tens of minutes for telephone modem users. Asynchronous write-back can improve responsiveness, cutting issue time penalty by as much as a factor of 10 in many cases. The cumulative impact of slow networks is especially striking given that we have already seen that most sessions contain relatively few operations (Figure 2.4(b)).

Although asynchronous write-back can reduce the issue time penalty seen by users, it introduces other problem: data which the user thinks has been saved to the server may actually be queued in the client cache waiting for upload. Asynchronous write-back reduces data durability and raises the possibility that different users view different versions of the data.

Figure 2.7 plots a CDF of latency to durability. The graph shows that many writes require more than the session timeout period to write data back to the server, especially at lower-bandwidths. If a user attempts to access data before it has been written back to the server, the server will return stale data. If the user notices that the version returned is old, it may look as though the changes just made were lost. Worse, if the user does not notice, new changes may conflict with old changes, requiring manual intervention to repair.

The effects of slow write-back can be exacerbated if a user shuts down a device shortly after completing a session to conserve limited battery power or to reduce charges from pay-per-

Timeout	Network	Files	Conflicts	
			Open	Close
15 min	10 kb/s	746	1.74%	1.71%
	56 kb/s	110	0.84%	0.91%
	cable	40	0.32%	0.39%
	1 Mb/s	9	0.016%	0.017%
60 min	10 kb/s	915	1.44%	1.36%
	56 kb/s	107	0.61%	0.65%
	cable	39	0.13%	0.16%
	1 Mb/s	11	0.016%	0.017%

(a) Synchronous Write-back

Timeout	Network	Files	Conflicts	
			Open	Close
15 min	10 kb/s	350	1.20%	0.59%
	56 kb/s	60	0.39%	0.26%
	cable	32	0.12%	0.083%
	1 Mb/s	13	0.011%	0.0067%
60 min	10 kb/s	514	0.81%	0.47%
	56 kb/s	56	0.17%	0.12%
	cable	30	0.091%	0.053%
	1 Mb/s	14	0.011%	0.0066%

(b) Asynchronous Write-back

Table 2.5: **Effect of Network Bandwidth on Conflict Rate:** Mobile users that rely on low-bandwidth network connections must cope with the long delays when accessing and storing data in network file systems. The latency can lead to inconsistent accesses. These tables show the number of files affected by this problem and the fraction of `open()` and `close()` operations that produce results different than those indicated by the original trace.

minute network plans. Modifications waiting to be written back to the server may be trapped on the device indefinitely when it is powered down and stowed away.

To understand the severity of this problem, we compare simulation results on finite-capacity links against “ideal” networks and record the number of conflicts. We consider an `open()` operation to be in conflict if the version of the data accessed differs between tests using ideal and real networks; we consider a `close()` to be in conflict if the version that is replaced differs. For this experiment, we assume that devices remain on-line to propagate changes to the server, even after the user has completed a session.

Table 2.5 reports the results of these measurements as the network bandwidth and write-back strategy varies. The tables report the number of files that are accessed in an inconsistent manner as well as the percentage of `open()` and `close()` operations that are in conflict. The results



show that if clients have broadband connectivity or better, the conflict rate is less than 0.4%. This mirrors results reported in the literature [BHK<sup>+</sup>91]. However, as the bandwidth of the networks decrease, the percentage of conflicts increase markedly. For users that must rely on technologies like cellular modems for connectivity, the conflict rate approaches 2% of accesses. Further, recall that the severity of this problem could be even higher if users shut down devices before they have propagated changes to the storage infrastructure.

#### 2.4.4 Discussion

The results of the previous section highlight several challenges for mobile users accessing network file systems. These challenges point to opportunities for innovation in remote file systems research.

The most significant results of this study concern the cost to mobile users of sharing data. For mobile users, data is shared not, as normally considered, between multiple users, but between multiple devices from a single user. In the traditional network file systems literature, conclusions on the importance of supporting sharing have been mixed. Measurements of the Coda distributed file system showed write-sharing of non-system files to be rare [KS91]. On the other hand, Baker et al. concluded from studying the Sprite file system that “shared access to modified data occurs often enough that a system should provide cache consistency” [BHK<sup>+</sup>91]. Our simulation results suggest that support for data sharing is important for controlling several different costs for mobile users.

One cost of increased sharing is increased traffic between server and client to maintain client caches. Section 2.4.3 showed that even users with only a few devices could face a 40% increase in cache maintenance traffic. Depending on how a user connects to the network, additional traffic leads to proportionally higher wait times or network usage fees. To reduce cache management traffic, file systems targeting mobile users may wish to employ compression techniques that exploit commonality between file versions stored in caches and on the server [MCM01]. Alternatively, a file system might attempt to maintain caches during periods when the network is otherwise idle (assuming that financial considerations do not restrict communication for speculative purposes). Finally, in some usage scenarios, systems might be able to use additional local bandwidth for peer-to-peer file transfers to reduce wide-area network usage.

Another problem with increased sharing is ensuring that users see a consistent view of data. In Section 2.4.3, we saw that low-bandwidth connectivity can render synchronous data write-back infeasible; but, with asynchronous write-back, modifications may not be propagated back to

the server until several session timeout periods have elapsed. Slow write-back increases the delay until changes are visible at other devices and can lead to inconsistent accesses. In fact, combining the results of Table 2.5(b) and Figure 2.4(b), we could expect weakly connected users to make at least one inconsistent access in half of all sessions. One way to address these challenges is to extend the observations that motivated Fluid Replication [KCN02]. Kim et al. recognized that safety (durability) and visibility of data, though often managed in a unified manner, can be decoupled. The designers of Fluid Replication used WayStations to ensure that data was made safe promptly. Our simulations, however, indicate that mobile users may be better served by systems that make changes visible quickly, even if at a slight cost to durability. This suggests that systems supporting weakly connected, mobile users may seek ways to improve visibility, for example, by prioritizing notification of modifications over the actual transfer of data. Of course, techniques for improving visibility need not preclude the use of WayStations, or similar mechanisms, to promote durability.

## Chapter 3

# Related Work

The work explored in this thesis was inspired and shaped by a number of previous designs presented in the research literature.

**Centralized, Remote File Systems:** The earliest systems to provide remote storage for users were centralized file servers. A centralized file server allows remote access to data stored on the server. A user storing data on the file server can access data from any machine that can access the server.

One of the earliest and most successful centralized file systems was the Network File System (NFS) [SGK<sup>+</sup>85]. NFS provides transparent access to remote file systems using a stateless protocol. Any machine can act as a server by exporting a subtree of its file system. Because the protocol is stateless, clients must contact the server on each read and write operation. To reduce the performance impact of repeated communication with the server, NFS adopted a set of heuristics to guide client caching. For example, clients were permitted to cache directory inodes for up to 30 seconds. Unfortunately, these heuristics often led to confusing, inconsistent results for users sharing data through NFS. Subsequent versions of the NFS protocol have addressed caching and consistency issues more methodically [SCR<sup>+</sup>03].

The Andrew File System (AFS) [HKM<sup>+</sup>88] used a stateful protocol to reduce the load on servers, improving scalability and performance. During normal operation, an AFS client interacts with the server only during open and close operations. During the open operation, a client retrieves and caches locally the whole file. By performing whole file caching, the client can perform subsequent read and write operations locally without contacting the server. When the application closes a file, the client pushes all changes back to the server. To ensure that client caches do not contain stale data, the file server records the location of all file replicas. When a file is modified, a server issues

a *callback* that invalidates other cached copies to prevent conflicted updates. Whole file caching results in consistency semantics that differ from traditional UNIX semantics; the authors assert that most applications function properly with these modified semantics.

**Distributed File Systems** Although centralized file servers provide convenient remote access to storage, the scalability and availability of any centralized design is necessarily limited. Even considering the improvements introduced by AFS to reduce the load on servers, as more users are added to the system, the centralized file server eventually becomes a performance bottleneck. Also, though a file server may be specialized hardware acting in a dedicated role and administered by trained staff, eventually the server will fail, preventing users from accessing their data. To mitigate the scalability and availability problems of centralized solutions, researchers have explored a number of designs for distributed storage systems.

Coda [SKK<sup>+</sup>90] is an extension of AFS that strives to improve data availability using two techniques: server replication and disconnected operation. Server replication stores data at multiple servers so that clients can continue access data even if some of the servers fail or are partitioned. Coda uses a variant of version vectors to maintain consistent replicas across replicated servers. Clients use a read-one, write-all approach to communicating with replicated servers. Disconnected operation allows clients to operate autonomously while disconnected from all servers by relying on data cached on the local disk. The system takes an optimistic approach to replication, allowing simultaneous changes at multiple clients. Upon reconnection, clients perform *reintegration* whereby modifications are pushed to servers and data is checked for conflicts.

The xFS [SDH<sup>+</sup>96] project attempts to remove the performance-limiting central server with a serverless network file system design. Tasks typically performed by the file server, including maintaining cache coherence and servicing requests, are distributed across machines in the network. Participating machines manage the metadata for a subset of files stored in the system and each machine maintains a global map that is used to locate data. The actual transfer of data can occur directly between clients; it need not route through the metadata server for that file. xFS was designed for environments where machines were connected with fast (LAN) network that trusted other machines to enforce security.

Bayou [PST<sup>+</sup>97] is a weakly consistent storage system that operates without centralized resources. Each file, or *database* in Bayou's parlance, is replicated on several servers throughout the network. Clients may read or update any replica. The read-any, write-any property of the system improves data availability. Replicas are periodically reconciled using an anti-entropy proto-

col. The system scales well because reconciliation occurs directly between replicas. Anti-entropy provides eventual consistency and naturally operates across weak and intermittent network connections. Bayou, however, relies on a primary commit scheme to determine when updates become stable [TTP<sup>+</sup>95]. Relying on a single master replica to determine update commit order allows clients to access data that may later be deemed in conflict and rolled back. Although Bayou's approach improves the availability of databases, it can make it hard for clients to determine that they are accessing valid and current copies of the data.

The Sprite [NWO88b] network filesystem allows different portions of the namespace to reside on different servers. Clients consult prefix tables to resolve filenames to the host machine. Sprite implemented an advanced caching scheme to provide high performance and strong consistency. Clients used a delayed-write policy that allows data to remain in client caches uncommitted for up to 60 seconds. The server, however, tracks files that may have been written recently and ensures that other clients receive consistent data, even if the server must recall that data from a client cache.

The Network-Attached Secure Disks (NASD) [GNA<sup>+</sup>97] system is designed to improve scalability by eliminating the file server from the critical path of most requests. The central file server, or metadata manager, tracks the location and state of files stored in the system, but it does not actually store the data. Disk drives attached directly to the network store the data. All metadata requests are serviced by the metadata manager, but requests to read or write data are submitted directly to the disk responsible for that file, after obtaining an appropriate capability from the server. One can view the NASD approach as a hybrid design that distributes data transfer among a set of disks that are managed by a centralized metadata manager.

Petal [LT96] distributed virtual disks represents one of the earliest truly decentralized designs for storage systems. Each participating machine uses the Paxos algorithm to maintain global state that maps virtual addresses to physical machines. Clients use the map to locate data and contact the specific machines directly. Although storage can be added to Petal incrementally, the scalability of Petal is ultimately limited to moderate sized systems by its use of the Paxos algorithm. Petal uses chained-declustering to distribute requests across machines for availability.

**Wide-area, Distributed Storage Systems:** As file systems continue to evolve towards increasingly distributed architectures, systems based on structured peer-to-peer overlay networks have received increased attention. These systems can be widely geographically distributed and scale gracefully. Communication occurs directly between peers without consulting a centralized resource.

First-generation peer-to-peer storage systems have demonstrated widely heterogeneous functionality. For example, CFS [DKK<sup>+</sup>01] and Venti [QD02] provide archival storage and thus do not support mutable data. PAST [RD01] allows clients to replace objects stored in the system; incremental updates are not supported.

Several systems do provide more traditional file system-like interfaces. Ivy [MMGC02] records all changes to the file system in logs that are distributed throughout the network. Far-Site [ABC<sup>+</sup>02] and OceanStore [KBC<sup>+</sup>00] rely on small sets of peers executing Byzantine agreement protocols to manage updates to objects stored in the system. Pangaea [SKKM02] aggressively replicates data whenever it is accessed. It implements optimistic concurrency with eventual consistency among replicas.

Other peer-to-peer storage systems focus on maintaining the availability and durability of data. Archival systems such as Carbonite [CDH<sup>+</sup>06], Glacier [HMD05], and Total Recall [BTC<sup>+</sup>04] track redundancy levels and initiate repair when replicas are lost. Many systems use a combination of replication and erasure codes.

Finally, several companies have recently made storage services available to the general public. These systems, including Amazon's Simple Storage Service [Ama06] and Apple's .mac service [App06], appear to be based on the same peer-to-peer paradigms as recent research systems.

**Log-Based Storage Systems:** Several previous storage systems have demonstrated the utility and benefits of log-based designs. The Log-Structured File System [RO92] is a local file system that writes updates to disk as if it were an append-only log, improving write bandwidth. It relies on large client caches to ensure good read performance.

The Zebra File System [HO93] extended the use of logs to network file systems. Zebra stripes data across a set of servers. Rather than striping data on a per-file basis, however, Zebra combines all writes into a single log and stripes the logs across multiple servers.

The Secure Untrusted Data Repository (SUNDR) [LKMS04] project showed how to exploit a log to store data securely on untrusted servers. In SUNDR, each update refers to the previous state of the log in a cryptographically secure manner and clients check the chain of references before reading data from the log. The scheme supports forks consistency, which the authors prove is the highest level of security that can be attained in the assumed environment.

**Update Encoding:** Researchers have previously explored how operation representation can reduce data transfer costs. The rsync [TM96] tool is a general-purpose file transfer utility that sup-

presses the transfer of blocks of a file if it can determine that those blocks already exist at the destination. The Xdelta [Mac00] project provides a delta encoding algorithm that extends upon the rsync algorithm and a suite of utilities to manage delta encoded storage.

More recently, the Low-Bandwidth File System (LBFS) [MCM01] used Rabin fingerprints to divide file into blocks for the purpose of eliminating the transfer of blocks already stored at the server. LBFS assumed that the server was trusted to manipulate data stored in plaintext form.

Operation-based update propagation [LLS99], or operation shipping, captures changes above the file system layer to reduce bandwidth requirements even more than delta encoding. After logging the user-level commands that produce file modifications, the commands are shipped to an identical surrogate client that is strongly connected to the server where they are re-executed. If the re-execution produces identical files, the changes are committed to the server; otherwise, the system reverts to shipping whole files.

**Link Scheduling:** Although many different filesystems projects have proposed various compromises among performance, consistency, and durability, we know of no work that has considered systematically the range of options. Kim et al. [KCN02] recognized that consistency, which they call visibility, and durability, which they call safety, can be managed separately. With this observation, they developed fluid replication to provide safety and visibility for data accessed by mobile clients. Caches in the local-area, called WayStations, replicate changes made by clients to provide safety without traversing wide-area links. Decoupling update notification and propagation allows other clients to view changes within predictable time bounds.

To support weakly connected clients, Coda introduce trickle integration [MES95]. Trickle integration is similar to write-back caching. Updates are logged and performed locally ensuring low response latency. Changes are written back to servers as network connectivity allows. Coda also uses an aging policy which delays write-back, allowing more opportunities for log optimization and further reducing the bandwidth required to write back changes. The LITTLE WORK projected implemented a technique similar to trickle integration directly on top of AFS [HH95].

PRACTI replication [DGN<sup>+</sup>06] aims to develop algorithms to support simultaneously partial replication, arbitrary consistency, and topology independence. A key feature of their approach is separating invalidations from message bodies. This allows invalidations to propagate across the network faster.

**Other Techniques:** The CAP conjecture, first offered by Eric Brewer [Bre00], states that it is not possible for a web service to provide simultaneously consistency, availability, and partition-tolerance. In designing systems, engineers must sacrifice at least one of these properties, and the features of the resulting system depend heavily on the choice of which property the designers choose to relax. We defer to the CAP conjecture in analyzing existing systems and developing new mechanisms. Our target application requires strong consistency and availability; consequently, the approaches we consider in this thesis necessarily assume a partition-free network.

Finally, we exploit several algorithms developed by others. The ideas behind self-verifying data were made popular by the Secure Read-Only File System (SFS-RO) [FKM00]. The hash-chaining technique for creating self-verifying data structures is due to Merkle [Mer88]. The fast fingerprinting algorithm that we use for delta-encoding updates was developed by Rabin [Rab81]. The data structure that we use to support delta-encoded updates is inspired by a similar data structure developed in the Exodus Database project [CDRS86].



## **Part II**

# **Techniques for Improving Access for Weakly Connected Clients**

## Chapter 4

# Bandwidth-Efficient, Privacy-Preserving Updates for Legacy Applications

In this chapter, we present a data structure that allows clients to update objects stored in a remote storage system in a bandwidth-efficient manner without sacrificing privacy. To benefit from the properties of the data structure, a client expresses an update as a sequence of operations that insert and delete data blocks in the data structure. Before shipping the update the server, the client encrypts all data blocks rendering them opaque to the server. To apply an update, the server executes the sequence of operations on the data structure in the storage system.

Unfortunately, the existing body of applications cannot benefit from this approach directly because legacy applications generally use the POSIX file system API to access data. So that all applications can enjoy bandwidth-efficient, privacy-preserving updates, we describe a technique to translate updates from legacy applications into the format used by our data structure.

This translation process is essentially a form of delta-encoding. Current approaches to delta-encoding assume that the server can be trusted to manipulate data in plaintext form arbitrarily. Such schemes are not suitable in our target environment where servers are not trusted to maintain data privacy. To broaden the applicability of delta-encoding to systems like those we target, we employ a client-side solution that exploits commonality between consecutive versions of a file to compute efficient operation-based updates.

With simulations, we have demonstrated that the technique can reduce update bandwidth, and correspondingly client-perceived latency of write and read operations, by up to 80%, depending on the workload, compared to techniques used in current systems. We have implemented the tech-

nique in the OceanStore prototype [REG<sup>+</sup>03] as well as the Moxie prototype described in Chapter 7.

The literature includes several projects that use delta-encoding to decrease the bandwidth consumed between a client and a file server [Mac00, MCM01]. Recently, the Low-Bandwidth File System (LBFS) brought much attention to the approach [MCM01]. The constraints of our target environment, however, render those previous techniques inappropriate. For example, previous techniques assumed a trusted file server that could perform arbitrary transformations on data stored in plaintext. In our system model, however, sensitive data must be encrypted to ensure privacy because communications channels and other machines are untrusted. Because the data is encrypted, the servers cannot transform the data in an arbitrary manner. Furthermore, previous schemes require multiple rounds of communication between the client and server [MCM01]. To improve interactive response time, we seek techniques that limit the number of network round trips.

## 4.1 A Data Structure for Bandwidth-Efficient, Privacy-Preserving Updates

In this section, we present a data structure that supports bandwidth-efficient, privacy-preserving updates. We first describe a simple data structure that provides sufficient support for legacy applications using the technique describe later in Section 4.2. We then describe a simple extension that provides additional flexibility for non-legacy applications.

### 4.1.1 Requirements

The requirements for the data structure depend on the general approach to the problem. We assume that clients modifying an object in the storage system will describe the changes as a sequence of block operations. For operations that write new data to the object, the client includes with the operation a data block. To ensure data privacy, the client may optionally encrypt the data in the block. When a server receives an update, it applies the sequence of operations, producing a new version of the data structure. When the client reads data from the server, the server returns a portion of the data structure, allowing the client to extract the requested data.

To support this general approach, the data structure used by the storage system to manage objects must meet several requirements.

- **Data Blocks are Opaque:** Data blocks must be considered opaque and indivisible by the storage system. They may not be repartitioned at the server, as in previous systems [CDRS86].

---

Operations to Support Bandwidth-Efficient Updates

---

append(block)  
truncate(length)  
insert(offset, block)  
delete(offset, length)  
read(offset, length)

---

Table 4.1: **Required Operations:** To support all of the techniques presented in this chapter, a data structure must support the operations listed above.

The only information that the block must reveal to the server is the number of bytes of user data that the block contains. If a client requests any portion of a block, the server should return the whole block to allow the client to invert any cipher or compression mechanism that was applied when the block was stored.

- **Data Blocks are Variable-Sized:** The data structure must index blocks of unequal size efficiently. Clients may store blocks of any size. This requirement renders inode-like schemes inappropriate.
- **Flexible Operations:** To allow clients to represent updates in the most bandwidth-efficient manner, the data structure must support a variety of operations. Table 4.1 lists the operations required for the techniques described in this chapter. Different system constraints or goals could lead to a different set of required operations.
- **Incremental Updates:** The required operations should be incremental. That is, if only a small portion of the object's data is modified, a correspondingly small part of the data structure should change. Incremental operations help to limit the amount of new data that must be stored with each update.
- **Data Structure is Self-Verifying:** Because we intend to use this data structure in a untrusted, content-addressable storage system, we require the data structure to be self-verifying. This requirement primarily affects how the data structure represents pointers to other elements of the data structure. Systems that do not require self-verifiability can represent pointers in other manners.

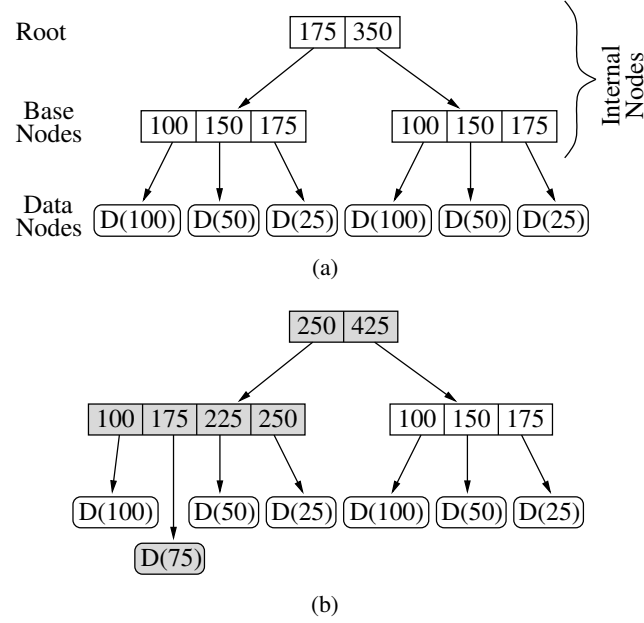


Figure 4.1: **A Data Structure to Support Bandwidth-Efficient, Privacy-Preserving Updates:** A B-tree that indexes data by *relative* offsets enables bandwidth-efficient updates. (a) The tree maintains an index over six blocks of data, shown at the leaves. The amount of user data stored in each data node is indicated in the figure. The root shows that the first 175 bytes of data can be found via the left subtree; data between offset 175 and 350 can be accessed via the right subtree. Note, both base internal nodes contain the same counts because the data structure uses relative indexing. (b) Maintaining relative offsets enables efficient incremental insert operations. Shown is the result of inserting 75 bytes at offset 100 in the data structure shown in (a). Blocks modified during the operation are shaded.

#### 4.1.2 A Basic Solution

The data structure presented in this section supports modifications only at block boundaries. This is sufficient to execute the updates created using the technique of Section 4.2. An extension that allows modifications at arbitrary offsets is given in the next section.

Our solution was inspired by the large-object data structure in the EXODUS database system [CDRS86]. The data structure, a variation of the B<sup>+</sup>-tree, is a balanced tree with all data stored at the leaves. We use the term *balanced* to mean that the number of children is balanced, not the amount of data in each child.

The tree is composed of two types of nodes. At the base of the tree are *data nodes*. A data node contains an opaque, indivisible, and uninterpreted block of user data.

Other nodes, called *internal nodes*, combine to form an index over the data nodes. The

top-most internal node is called the *root*; internal nodes that point directly to data nodes are called *base internal nodes*. Internal nodes store up to  $2n + 1$  (*count*, *pointer*) tuples, where  $n$  is the *degree* of the tree. Each *pointer* is a self-verifying reference to another node in the tree. All internal nodes except the root store between  $n$  and  $2n + 1$  non-null tuples at all times; the root may contain fewer tuples. The *count* field is an integer that records the number of bytes accessible from the current node via children referenced from the given tuple and all other tuples to the left of the current tuple. By convention,  $count[-1] = 0$ . The key feature of this structure is that the *count* field records the *relative*, rather than *absolute*, offset of data within the structure. The *pointer* field is a secure hash that identifies and verifies the child node. The *pointer* fields form a Merkle [Mer88] tree over the structure, making it self-verifiable. Figure 4.1(a) shows an example of this data structure over a file divided into 6 blocks.

We describe in detail the algorithm used to insert a block in the data structure. Other operations are implemented in a similar fashion.

The insert operation allows a client to insert a block in an object. It does not overwrite any data; rather, the offset of all data after the insertion point is shifted by the size of the inserted block. Let *Data* be the block of opaque user data to be inserted, and let *InsertPoint* be the offset at which to insert the block. Recall that, for the current discussion, the insertion point must lie on an existing block boundary. Throughout the discussion, refer to Figure 4.1(b) which illustrates the result of an insert operation on the data object of Figure 4.1(a).

1. The algorithm must not descend into a node that cannot accommodate a new child. If the root is full, create a new internal node to be the root, assign the old root to be its only child, and split that child. The split procedure works in a fashion similar to the traditional B-tree split operation. Let  $Node \leftarrow root$ . Proceed to Step 2.
2. *Node* contains an array of tuples describing the range of data that may be reached from each child node. Using this state, find the child that is the parent of the insertion point; that is find  $i$  such that  $count(i - 1) < InsertPoint \leq count(i)$ . If *Node* is a base internal node, then, by our assumption that all modifications occur at a block boundary,  $InsertPoint = count(i)$ . Let  $count(j) \leftarrow count(j) + Data.size$  for all  $j \geq i$ . Also, mark the child at  $pointer(i)$  as modified. If *Node* is a base internal node, skip to Step 4; else proceed to Step 3.
3. Prepare to descend to the child. If the child referenced by  $pointer(i)$  is full, split that child and adjust  $i$  accordingly to point to the new child that is the parent of the insertion point. Let

$Node \leftarrow pointer(i)$ . To track the relative offsets, let  $InsertPoint \leftarrow InsertPoint - count(i - 1)$ .  
Recurse to Step 2.

4. Insert a new tuple at offset  $i$  with  $count(i) = count(i - 1) + Data.size$  and  $pointer(i) = HASH(Data)$ .  
The operation is now complete.

Although the new data block has been inserted, the ability to verify the data structure has been lost. To restore the verifiability, after all actions of the update have been applied, perform a depth-first traversal of the data structure, recomputing the secure hash of all modified blocks and storing the result in the parent. The traversal need not descend into any children that were not modified. Thus the amount of data that must be hashed after any update is proportional to the amount of modified. The algorithm could be extended to maintain verifiability after each modification. However, most updates contain multiple operations, and we can save computation by restoring verifiability only after all modifications have been completed.

Note, the algorithm, as described here, never descends into a full internal node. That is, before descending into a child that is full, the parent node splits the child node. This may result in unnecessary split operations. We rationalize, however, that if a node is nearly full, it will be split soon, and thus splitting slightly before it is strictly required is not particularly onerous. Opportunistic splitting also simplifies the algorithms. If needed, algorithms that split internal nodes only when strictly necessary can be implemented.

### 4.1.3 Supporting Arbitrary Insertions

The solution presented in the previous section supports points of modification only at existing block boundaries. While this functionality is sufficient to support the technique presented later in Section 4.2, a simple extension enables even more bandwidth-efficient updates for non-legacy applications.

The extension allows the data structure to support insertion and deletion at *arbitrary* offsets. In all base internal nodes, we add to the tuples an additional *offset* field. The new field is defined to point into the middle of a block of encrypted data stored in the data node. The extension is illustrated in Figure 4.2. In the example, before the update, the data structure holds two chunks of data, one of 500 bytes and one of 50 bytes. The first  $count[0] = 150$  bytes can be found starting at offset  $offset[0] = 350$  of the encrypted data stored in the first data node. The first 350 bytes in that block could be referenced by a tuple that is not shown or could be unreferenced in the current version.

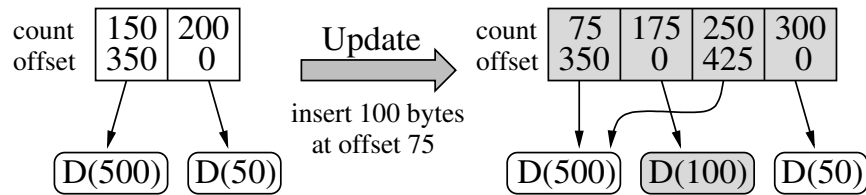


Figure 4.2: **Data Structure Extension to Support Arbitrary Insertions:** By maintaining an “offset” which points into the middle of a block of ciphertext, the data structure can support the insertion and deletion of data at arbitrary offsets. This extension might enable more efficient updates.

The figure also shows the result of updating the data object. After inserting 100 bytes at offset 75, the first 75 bytes of data can be found starting at offset 350 of the first data node. The bytes in the range 175 to 250 can also be found in the same data node, starting at offset 425. The algorithms to modify this data structure are similar to those used to modify the data structure in Section 4.1.2. The primary difference is the condition that a node checks in deciding whether it must split a child node. When inserting data in the tree, a base internal node may need to add two tuples to point to new child nodes. Thus, assuming the implementation splits nodes opportunistically, as described in Section 4.1.2, when descending into a child, the parent must split the child if it is almost full. That is, a parent must split the child if it already stores  $2n$  or  $2n + 1$  tuples.

Although this extension does allow clients more flexibility in how they represent updates, it comes with additional complications, especially related to security and performance. For example, the data nodes may store data that has been deleted from the current version of the data object. By manipulating the blocks directly, a malicious attacker could reconstruct information about the change history of the file. Of course, the attacker would still need a key to decrypt any encrypted data. This dead data is also a performance liability. Retrieving the data object from the infrastructure may require retrieving extra data that is no longer relevant to the current version of the document.

Despite the complications, the extension opens several possibilities. For example, with a carefully defined data format, a non-legacy application with a stale copy of the data object could submit an update without retrieving the current version and merging the changes. Alternatively, an application could dynamically trade-off computation and bandwidth depending on the current level of connectivity. In low-bandwidth conditions, it could submit the smallest update possible. When connectivity is restored, the application could retrieve the data object, defragment the data, and rewrite it to the infrastructure in the more efficient manner. Finally, an application could be written to exploit the asymmetric nature of many networking technologies. For example, a typical cable modem provides 3–16 times the amount of downstream bandwidth as upstream bandwidth.



An application could submit small updates that can be uploaded to the infrastructure quickly even if they result in less efficient storage in the data structure knowing that extra downstream bandwidth will enable quick retrieval of the document.

## 4.2 Extracting Bandwidth-Efficient Updates for Legacy Applications

In Section 2.3, we showed that there often exists significant similarity between different versions of the same file. In this section, we describe how clients can exploit that commonality to create bandwidth-efficient updates for storage systems that use the data structure of the previous section for storing objects. This technique is a client-side approach designed to benefit legacy applications accessing unstructured data.

Briefly, to exploit similarity between versions of a file, we compare the new version of an object to its previous version, extract the changes made, and encode those changes in an update. An update is similar an edit script [Mye86], describing how to transform the old version of the file to the new version. The update is then sent to the middleware servers to modify the state stored in the system. Refer to Figure 4.3 as the details of this approach are described below.

Commonly, edit scripts are computed using a utility such as `diff`. Such tools are not appropriate, however, for delta encoding in file systems. Many of the problems stem from `diff` and similar tools being byte-oriented utilities that identify changes at a very fine granularity. The computational cost of computing fine-granularity edit scripts on frequent write operations is infeasibly expensive. Secondly, to execute edit scripts created by tools like `diff`, the executor must be able to manipulate text arbitrarily. Because middleware file servers in our target environment are trusted with user data only in ciphertext form, the condition cannot be satisfied. Finally, even if the servers could edit the ciphertext arbitrarily, encrypting edit scripts that describe changes at a fine granularity would reduce the potential bandwidth savings because the padding and initialization vectors necessary for encryption could dwarf the actual data changes.

Rather than computing fine-granularity edit scripts, we divide a file into blocks and consider changes at that coarser granularity. At the center of our approach is the *file summary*. A file summary is a condensed representation of the contents of a file. File summaries are similar in concept, if not in use, to file recipes used in the CASPER file system [TKS<sup>+</sup>03]. To create a file summary, a file is first divided into smaller pieces, or *chunks*. Each chunk is named by a cryptographically secure hash of the plaintext of the chunk’s data. The file summary records the list of chunks that comprise the file. The file summary also records the length of each chunk and its offset

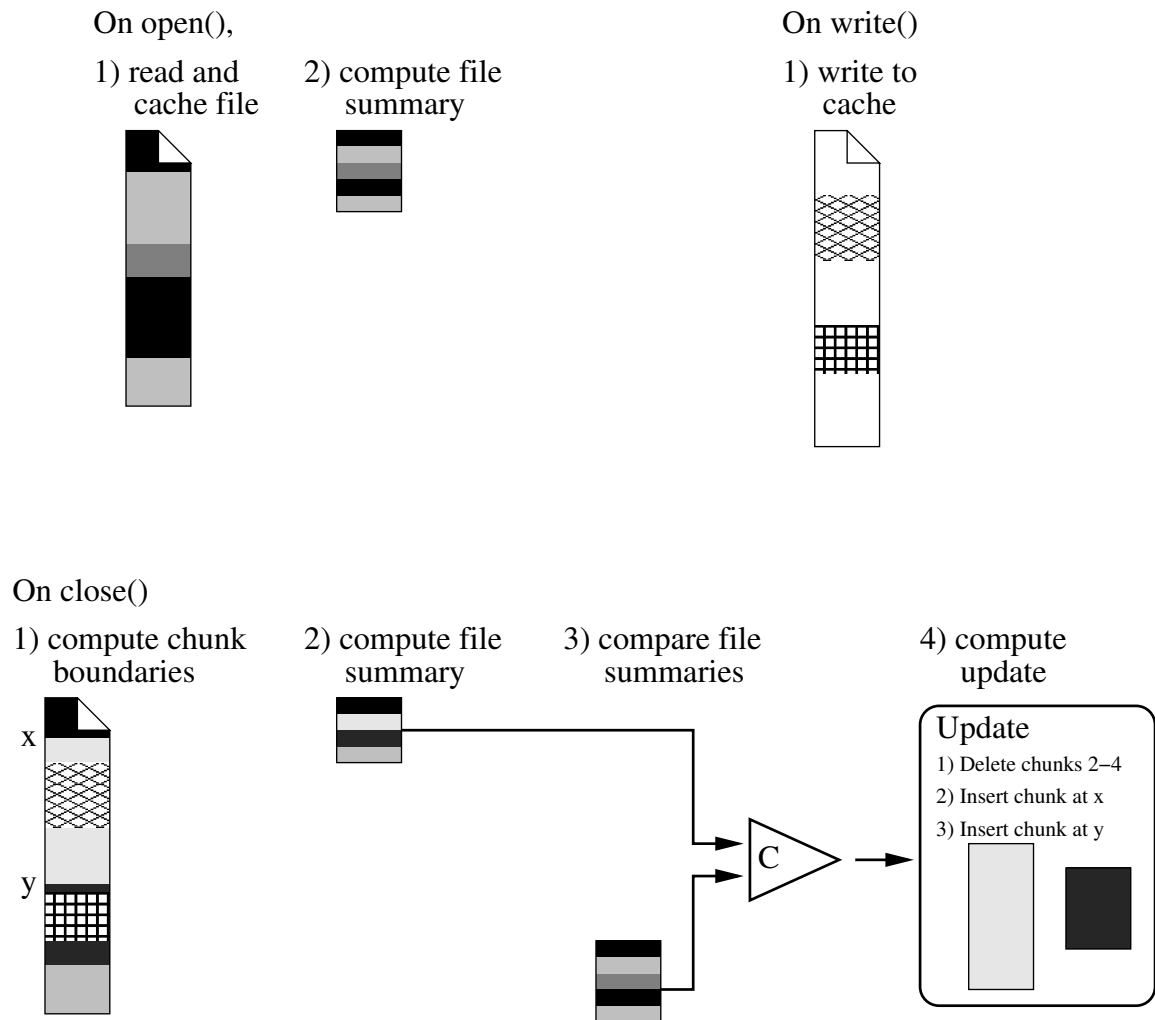


Figure 4.3: **Client-Side Delta Encoding:** When an application accesses a file via the `open()` operation, the client machine reads the file, if necessary, and caches it locally. It computes the chunk boundaries for the object and constructs a file summary. In this example, the fingerprinting algorithm divides the file into five chunks; the file summary has one entry for each chunk. On `write()` operations, the client simply modifies the file in the local cache. The figure indicates that the application has written to two regions of the file. On `close()`, the client computes the chunk boundaries for the new version of the file and constructs a new file summary. Here, the new version contains four chunks. The first and last chunk remain unchanged from the previous version. The client then compares the new file summary against the previous file summary and computes an update to submit to the server. File offsets 'x' and 'y' referenced in the update are indicated on the chunked copy.

in the file.

To divide a file into chunks, we use the Rabin fingerprinting algorithm as in LBFS [MCM01]. A fingerprinting algorithm, in general, computes a hash over a piece of data; Rabin's fingerprinting algorithm, in particular, can compute hashes efficiently over a sliding window of data. We use Rabin fingerprints to compute the fingerprint over data at each offset of the file. If the fingerprint at a given offset falls in a defined range, we mark that offset to be a breakpoint, defining a chunk boundary. This approach tends to identify the same boundaries even as modifications may shift data offsets or change the size of the file, and it divides a file into chunks such that the number of chunks that differ between consecutive versions of a file is roughly proportional to the amount of data modified between the versions. To avoid pathological cases that produce chunks that are unreasonably large or small, we define a minimum and maximum chunk size, as in LBFS. Boundaries are ignored if the resulting chunk would be too small; a boundary is inserted if the chunk reaches the maximum size.

When data is written to a file, the object's file summary changes. It is probably not reasonable to, however, update the file summary after every write operation. Though the Rabin fingerprinting algorithm can compute chunk boundaries (and thus file summaries) quickly compared to other fingerprinting techniques, computing a new file summary after every write would require the client to scan portions of the file around the changes (if not the whole file) to update file summary. Further, the client needs an updated file summary only when it computes an update to send to the storage system. Experience with the Andrew File System [HKM<sup>+</sup>88] has taught us that most applications work correctly when changes are propagated to the file server only when the application closes the file. Thus, to amortize the cost of computing file summaries we buffer multiple write operations, computing a new file summary only when the file is closed. More precisely, a client creates a new file summary on every file `close()`, if the file was originally opened for writing.

Clients use the file summaries to construct updates to send to the middleware servers. To exploit the commonality that is typical between consecutive versions of the same file, the client compares the newly created file summary to the file summary of the previous version. The result of the comparison is an edit script that describes how to insert and delete chunks from the previous version of the file to reconstruct the new version. An update consists of this edit script along with all chunks containing new or modified data. The client then forwards the update to the middleware server.

This procedure tends to create efficient updates; that is, the size of the update is roughly proportional to the amount of data that actually changed. It does this by translating the set of

int data_length	byte[ ] init_vector	byte[ ] ciphertext
--------------------	------------------------	-----------------------

Figure 4.4: **Contents of a Data Block that Supports Encryption:** Before shipping a chunk to the server, the client packages it in a block that contains the length in bytes of actual user data (not including padding or initialization vectors), the byte array containing the initialization vector, and the ciphertext.

traditional file system operations into a more flexible set of operations. Unmodified applications still use the standard file system interface with truncate, append, and overwrite operations. With only these operations, however, applications are often forced to overwrite the tail of a file, even if only a small amount of data is changed. A naive translation of file system operations to updates would also result in overwriting the tail of a file. The proposed process for extracting operations recasts the modification as a series of insert and delete operations that more succinctly describe the changes made to the file.

The client translates each chunk in the update to the opaque data block that is stored in the data structure by the server. The contents of a data block that supports encryption are shown in Figure 4.4. The `data_length` field records that amount of actual application data stored in the block, neglecting any encryption overhead and is used by the server when operating on the data structure. The other contents are interpreted by only the client. Any additional state that the client requires to interpret the data blocks, like encryption algorithms or key names, should be stored in the object's metadata, not in each data block. Because data blocks are opaque to the server, clients may perform additional data formatting as they see fit. For example, to conserve additional bandwidth, an implementation of the client might compress data in a chunk before encrypting it.

Note that fingerprinting and creation of file summaries must be performed over the plaintext of the data. If the data were to be encrypted (using the standard cipher block chaining mode) before it was divided into chunks, later chunks would depend upon earlier chunks. This would eliminate the possibility of detecting file commonality after the first modification point in the file and consequently increase the size of updates created by the client.

A server applying a delta-encoded update will produce a new version of a file that matches the version written at the client only if it applies the update to the same version of the file used in the file summary comparison at the client. Applying the update against a different version could corrupt data. To ensure consistency, each update is predicated on the condition that the current version of the data object managed at the server is the same version used to construct the update. If

the condition does not hold when the server processes the update, the client will receive an update failure response. The client can then elect to fetch the new version of the data object and construct a new update or overwrite the entire file. Of course, not all applications or updates require this level of consistency. For example, an application may wish to append data to a file regardless of the its current state. In that case, the update could be predicated on the null predicate or the true predicate.

This technique for extracting update operations from legacy applications is compatible with a range of consistency models. Using an optimistic approach, a client would compute the file summary and update based on the file version cached at the local machine, without contacting the server. If the predicate is not satisfied by the version stored at the server, the client would receive an update failure response indicating that it should re-compute the update based on the current version of the file. Such a scheme would force all updates to fall in a sequential order. To allow concurrent modifications to different segments of the same file by different clients, a storage system could support more flexible predicates, such as range predicates that require only a portion of the file to be unchanged.

After committing an update to the infrastructure, a file summary can be considered soft state. In the common case, the client should cache the file summary for use during the next write operation. However, a client may discard a file summary at any time to pare the amount of state that it must record. It can simply recreate the file summary the next time an application opens a file for writing. Certainly, this approach incurs a significant computational cost, but for many clients, computational capacity is abundant compared to scarce network bandwidth. Alternatively, the client may choose to neglect altogether the file summary of the previous version, instead sending the entire file in the next update. This approach may be appropriate for small files. We do not consider these optimization further.

The approach described above increases the effective bandwidth of the network link by reducing the size of the update that must be transferred across the network. This affects a corresponding improvement in client-perceived update latency. Combined with caching, this technique can also improve the client-perceived read latency.

Caching can reduce the latency to read data, even if the cache contains stale data. If a client has a previous version of a file cached, to reconstruct the current version of the file, the client needs only to retrieve the blocks that were added to the data structure after the the cached version. Because the process that creates the updates tends to preserve unmodified chunks, a client with a recent version of a data object cached needs to retrieve a number of new chunks proportional to the amount of data that has actually changed.

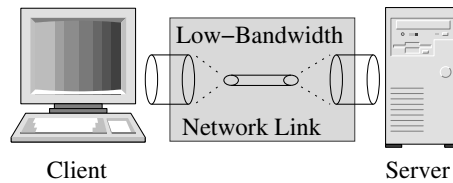


Figure 4.5: **A Simulator for Evaluating Bandwidth-Efficient Updates:** Our evaluation is based on a simple simulator in which the client and server directly via an emulated low-bandwidth network link.

## 4.3 Evaluation

We have implemented the data structure presented in this chapter (supporting arbitrary insertions) in the OceanStore prototype [REG<sup>+</sup>03] and the Moxie prototype (described later in Chapter 7). We have implemented the client-side technique for extracting operation-based updates from legacy applications in the Moxie prototype. However, for experimental isolation and control, we evaluate these schemes using simulation based on the client-server model. A discussion and evaluation of these techniques in the larger Moxie system is included in Chapter 7. Note, some systems that use modified client-server architectures may benefit more from these mechanisms than the measurements here indicate. For example, the OceanStore design implements the server as a set of replicas called the “inner ring” that use Byzantine agreement algorithms to modify state. In this design, a client sends an update to a single members of the inner ring. That replica then broadcasts the update to the other replicas. In OceanStore, any savings in bandwidth between the client and inner ring would be magnified in the communication among the replicas.

### 4.3.1 Simulation Overview

We developed a simple simulator to drive our evaluation. In the simulations, the client executes traces of file system activity, computes file summaries and updates, and submits requests to the server. The server applies updates from clients and serves blocks to the client in response to read requests. The two components communicate using TCP through an emulated restricted network connection. This configuration is shown in Figure 4.5.

The simulator is written in Java using the event-driven architecture toolkit developed as part of the Bamboo project [RGRK04]. The client and server components run simultaneously on a single Dell Dimension 8300 desktop with one 3 GHz Pentium 4 processor and 1 GB of RAM running Gentoo Linux 1.4.

Technology	Latency (ms)	Bandwidth (kb/s)
Cellular modem	100	10
Telephone Modem	100	56
Cable Modem	30	384 down / 128 up
LAN Connection	10	1000

Table 4.2: **Network Characteristics:** The simulated network connection is controlled to perform like several popular networking technologies.

To emulate low-bandwidth network connections, we impose a delivery delay on all messages at the receiver. We assume a simple model in which the bandwidth and latency are well-defined and constant between the client and server. We compute the delay using the familiar  $\alpha + \beta \cdot n$  formula, where  $\alpha$  corresponds to latency,  $\beta$  corresponds to bandwidth, and  $n$  is the size of the current message in bytes. Since the actual network connection between the two components is the loopback network device, we ignore any latency the connection imposes.

Table 4.2 shows the bandwidth and latency parameters used in the evaluation. They correspond roughly to a cellular modem, a telephone modem, a cable modem, and a LAN connection.

We compare the bandwidth savings achieved with using our approach for computing updates against two alternative techniques. The *Whole* approach treats the entire file as a single chunk. If any single byte of the file is changed, all data must be shipped to the server for update. This is analogous to the approach employed in PAST [RD01]. The *Block* approach divides a file in to fix-sized blocks, much as a traditional file system divides a file into blocks for storage on disk. In this scheme, the server represents the data as a standard  $B^+$ -tree. When a file is modified, only those blocks that have changed need to be submitted to the server. In simulations, each block is 4096 bytes. This is similar to the approach used in CFS [DKK<sup>+</sup>01].

When simulating the *FileSummary* approach, chunks are identified using Rabin fingerprints with a minimum chunk size of 4 kB and a maximum chunk size of 16 kB. We use the SHA-1 algorithm for secure hashing and the algorithm best known for its implementation in the UNIX diff utility [Mye86] for comparing file summaries. All user data is encrypted using the Rijndael/AES cipher with 128-bit keys.

### 4.3.2 Workloads

We drive the simulator with synthetic micro-benchmarks and macro-benchmarks created from traces of actual file system activity.

Name	Files	Total Write Size (kB)	Commonality			
			Whole	Block	Rabin	Rsync
Word	1	700	0%	0.6%	82.4%	93.6%
Bamboo	26	435	0%	10.9%	20.4%	75.7%
Mbox	1	100	0%	81.8%	78.4%	82.1%

Table 4.3: **Micro-benchmark Characteristics:** The micro-benchmarks vary in size and in commonality between consecutive versions using the different chunking strategies.

**Micro-benchmark Workloads:** Table 4.3 summarizes the three micro-benchmark workloads used in our evaluation. We describe each workload in turn.

The *Word* workload is composed of consecutive versions of a document created using Microsoft Word 2000. The 11-page document includes several figures and graphs. To create the second version, we performed a global search and replace operation, changing 15 instances covering most pages of the document. This workload is representative of an office productivity workload that works with large binary files.

The *Bamboo* workload is consecutive source code releases from February 9 and 13, 2004 of the Bamboo DHT and event-driven programming toolkit <sup>1</sup>. The February 13th release modified 26 files ranging in size between 1.5 kB and 70 kB. This workload is representative of a developer workload in which most files and changes are small and hand-edited.

The *Mbox* workload is composed of consecutive versions of a user’s email folder stored in mbox format. The user removes deleted messages from the folder and saves new messages to the folder. After the modifications, the size of the folder increases by approximately 25%. In the mbox format, new messages are always appended to the end of a mail folder.

**Macro-benchmark Workload:** The macro-benchmark workload is created by replaying the traces of developer activity described in Section 2.3. We extracted from the traces all activity directed at the user’s home directory. We pruned all requests to temporary files and all accesses related to web browser activity. We posit that such accesses would be handled by local file systems, even if remote storage systems were available. We identified temporary files as files that were created and later removed by the same process. We replay the trace as fast as possible, with no user think time.

---

<sup>1</sup>Current and previous Bamboo releases are available at <http://www.bamboo-dht.org/>.



### 4.3.3 Workload Commonality

The effectiveness of this approach depends on how much similarity can be identified between consecutive versions of files. Table 4.3 reports the percentage of commonality found in the micro-benchmark workloads using different chunking techniques. We define commonality to be the ratio of bytes found in chunks from the previous version to the total bytes in the new version. The commonality of the macro-benchmark workload is reported in Table 2.3.

Of course, the Whole approach is unable to identify any commonality in any micro-benchmark workload in which all files change. It does, however, identify a small fraction of commonality in the macro-benchmark workloads when the file that is written to disk matches identically the file already stored on disk.

The effectiveness of the Block approach is sensitive to the offset in the file of the first modification. If the point of first modification is early in the file, as in the Word workload, the approach is able to detect only very little commonality. If the offset of first modification is later in the file, the approach can perform very well. The commonality results from the micro-benchmark shows that there exists many workloads on which the Block approach does not perform well. The macro-benchmark results, however, shows that on typical developer workloads the Block approach works reasonably well compared to other approaches.

The Rabin approach used to create file summaries performs well across the range of workloads. It can identify commonality throughout a file, even if modifications shift the offset of shared chunks. Because of the relatively large minimum chunk size, the scheme fails to identify all available commonality when files contain many small changes, as in the Bamboo micro-benchmark.

We also compute the amount of commonality that could be found by a scheme unencumbered by the restrictions of our target environment. The *Rsync* results report the amount of commonality identified using the rsync [TM96] binary diff algorithm with a very small minimum block size of 64 bytes. This approach is able to identify more commonality than other schemes by searching for shared data at a much finer granularity.

### 4.3.4 Simulation Results

The primary benefit of the technique presented in this chapter is the reduction in bandwidth required to ship updates to the server by exploiting commonality between versions of a file. To detect this similarity, however, a client must incur other costs. We begin by studying these additional costs.

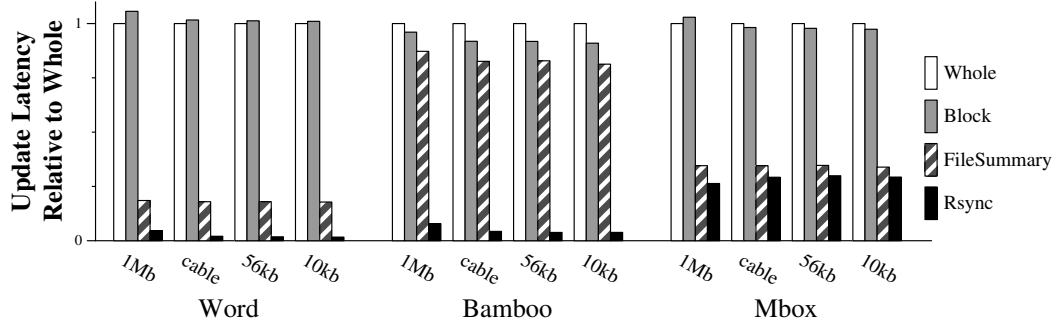


Figure 4.6: **Micro-benchmark Update Latency:** The client-perceived write latency depends on the amount of commonality detected in the workload and the characteristics of the network.

The client must pay the computational cost of computing chunk boundaries and translating file summaries into updates. Using unoptimized Java code, the client can detect chunks using Rabin fingerprints at a rate of 5 MB of data per second. To avoid redundant computation, the client can cache previously computed file summaries. The overhead of storing file summaries for the three micro-benchmarks ranges from 4.3 to 6.0 megabytes per gigabyte of stored data. The total cost of creating the update is dependent on the length of the file and the number of new chunks. On the Word workload, for example, it takes 120 ms to create the update; of this time, however, about 50 ms is spent encrypting the data.

We measured the performance improvement attained by our proposed scheme on the various benchmark workloads. In addition to comparing the performance to the Whole and Block approaches used in current systems, we also compare the experimental results against an ideal, lower-bound latency approximated by use of the rsync utility. The *Rsync* latency is a computed quantity assuming that operation latency depends only on the speed of network transmission. The size of the update is computed using a binary diff algorithm based on the rsync [TM96] as in Table 4.3 and assumes that the server can perform arbitrary transformations on encrypted data. Furthermore, it is assumed that computation is infinitely fast and data can be stored as ciphertext with no storage overhead. We consider this quantity a near-optimal latency.

**Micro-benchmark Results:** First, we measure the impact of our technique on write latency on the different micro-benchmark workloads and under various network parameters. Figure 4.6 shows the client-perceived latency of writing the various workloads to the server. We measure the time to update the file(s) stored on the server to the second version in the workload. The results are the mean of three trials. The variance was negligible in all cases.

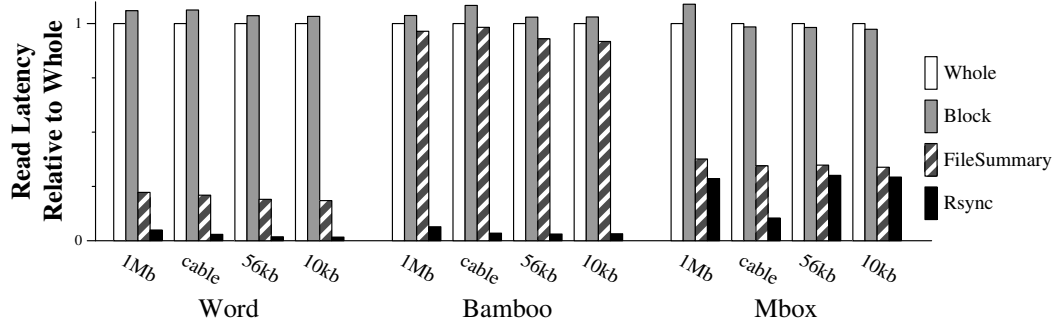


Figure 4.7: **Micro-benchmark Read Latency:** The client-perceived read latency also depends on the commonality in the workload and the network characteristics. The results differ from the update latency because clients must retrieve the internal nodes of the data structure.

The performance does, indeed, reflect the amount of commonality that can be identified by the different schemes (as reported in Table 4.3). By recognizing more commonality, the FileSummary scheme performs better than the Whole and Block approaches.

In general, none of the schemes approaches the approximated ideal (Rsync) latency. One could interpret the overhead as the cost of ensuring privacy. In estimating the minimum update latency, we compute updates at the byte granularity and ship them to the server in plaintext form. With the other approaches, we consider changes at a more coarse granularity, limiting the amount of commonality that can be identified between consecutive versions of files. Although it does increase bandwidth requirements, managing changes at the chunk granularity improves the efficiency of the data structure for indexing the chunks and reduces the overhead of padding and initialization vectors for encryption. It is possible that unique data characteristics in some workloads may allow for the identification of an “ideal” block size that minimizes these overheads. For general purpose workloads, however, we have seen no particularly acute sensitivity to block size. The minimum and maximum block sizes used in our simulations mirror those used in the LBFS [MCM01] work that inspired our approach.

Next, we measure the impact of the technique on read latency. We assume that the client has a copy of the first version of the workload in the local cache. Upon requesting the file, the client discovers that a newer version has been created and that it must retrieve chunks to reconstruct the new version locally. This situation would arise for a user that accesses data from multiple machines or devices.

Figure 4.7 presents the results of this test. As expected, identifying unmodified data allows clients to update local copies of the file by retrieving less data from the infrastructure. Note that

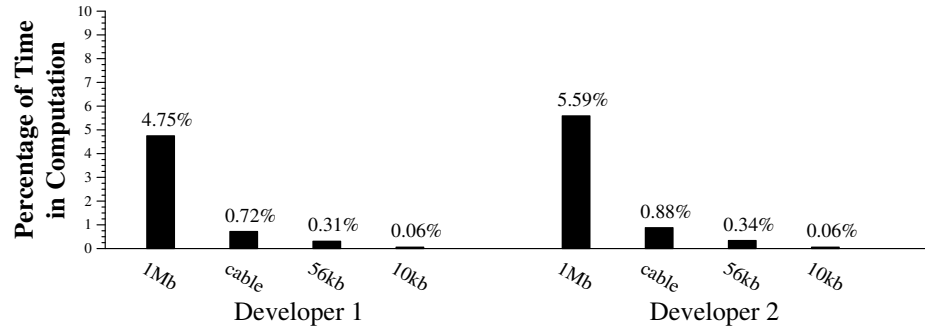


Figure 4.8: **Client-Side Computational Overhead:** The graphs show the relative cost of additional client-side computation for various network speeds. For fast connections, the computation overhead can be as much as 5%. For slower networks, the overhead is negligible.

when reading data from the infrastructure, the Block and FileSummary schemes pay a penalty for retrieving the internal nodes of the data structure. The Block approach actually performs worse than the Whole method on some workloads because of this overhead coupled with the lack of detected commonality. The FileSummary scheme, however, is able to detect sufficient commonality to overcome this penalty and perform better than the Whole approach.

**Macro-benchmark Results:** With the macro-benchmark workload, we have enough data to begin to evaluate the computational overhead required by our scheme. As before, we measured total client latency for each update. However we also profiled the time required for the client to identify data changes, create an update, and encrypt the update. These actions represent the computation required on the client side. On the server side, we measured how long it takes to apply an update. The remaining client latency is due to network delays. We aggregated this data over both developer traces to consider total computation time required by both client and server as a percentage of total client latency. These results are presented in Figure 4.8. We use Rabin fingerprints to compute chunk boundaries.

The computational overhead decreases as the network becomes more constrained; for less powerful connections, network delays dominate the client latency. The results for both developers are remarkably similar, with the worst performance at around 5% for both when using the 1 Mb link. As the link becomes weaker, the network time quickly begins to dominate. Keep in mind that all of these computations are done in unoptimized Java and could be reduced further by optimization in Java, C, or even cryptographic hardware.

We also compared the cost of the different phases of additional computation. Figure 4.9

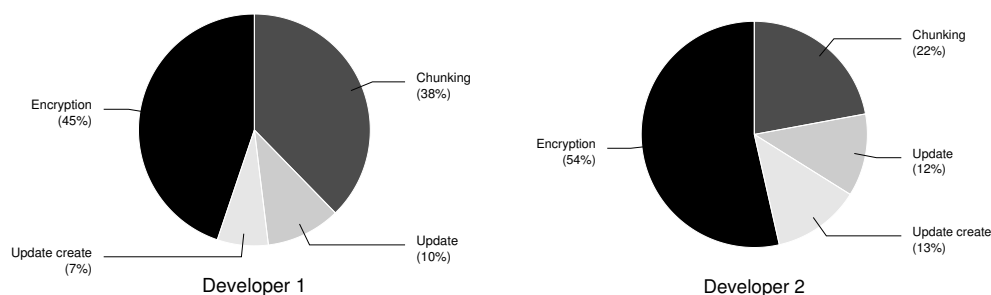


Figure 4.9: **Client-Side Computational Profile:** The charts show the relative cost of different phases of computation. For both traces, data encryption is the most expensive phase.

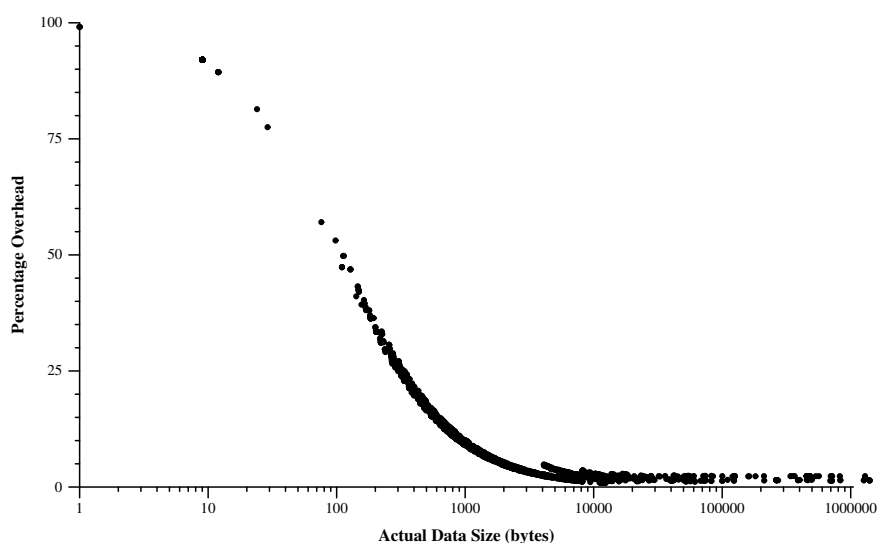


Figure 4.10: **Storage Overhead:** The storage overhead of the data structure is significant for small files. For larger files, the overhead is much smaller.

shows the profile for both developer traces. These figures are again aggregated over the entirety of both two week traces. The update creation on the client side and update application on the server side tend to be the least computationally intensive tasks. Indeed they are both manipulations of efficient data structures. The chunking of data (identify chunk boundaries) is a significant computational cost, but as we saw before, it saves us time in network transmission. Encryption tends to require the most amount of time, but is unavoidable if we wish to maintain privacy.

Finally, we analyzed the storage overhead of our data structure. After each update, we had the server report the total size of the data structure and the size of the actual, stored data. The storage overhead is due to the data structure and encryption: the system must storage the index maintained

in the data structure and the initialization vectors and padding to support encryption. Using these figures, we computed the percentage storage overhead as:

$$1 - \frac{\text{actual data size}}{\text{total data structure size}}$$

In Figure 4.10, we scatter plot this overhead against the size of the data. These data points come from both developers traces. The overhead of the data structure is significant for small file. As file size increases, the overhead decreases. For files greater than 1 kB in size, the overhead of the data structure is less than 10%. For files bigger than 10 kB, the overhead is below 5%.

## Chapter 5

# Intention Updates

Operation-based updates can reduce the amount of bandwidth used to communicate changes between client and server when commonality can be detected between consecutive versions of a file. Still, communication may require significant bandwidth in many cases. For example, many applications, like media applications, produce large files that contain data unlike any other files stored in the system. Also, the initial version of a file must always be transferred in its entirety. Finally, measurements have shown that not all modifications produce files similar to their predecessor (Tables 2.3 and 4.3). In cases such as these, the system must transfer large amounts of data that cannot be optimized with delta-encoding.

In this chapter, we describe a technique that decreases the effective latency of large transfers when a client writes data to the server. The mechanism, which we call *intention updates*, separates the management of metadata and data resulting in a two-phase update operation that promotes fast update visibility and enables flexible scheduling of upstream bandwidth. In the first phase, a client sends a small *intent* to inform the server about the modification. Later, the client consummates the update by transferring the modified data to the server. With two-phase updates, changes can be propagated to the server quickly enabling the server to track the state of data accurately and ensuring that modifications are visible to other clients in a timely manner.

### 5.1 Background: Write-back

The intention update technique optimizes the process that clients use to propagate modifications to the server. Before considering the details of intention updates, it is important that we first understand the traditional approaches to data write-back. Refer to Figure 5.1 during the following

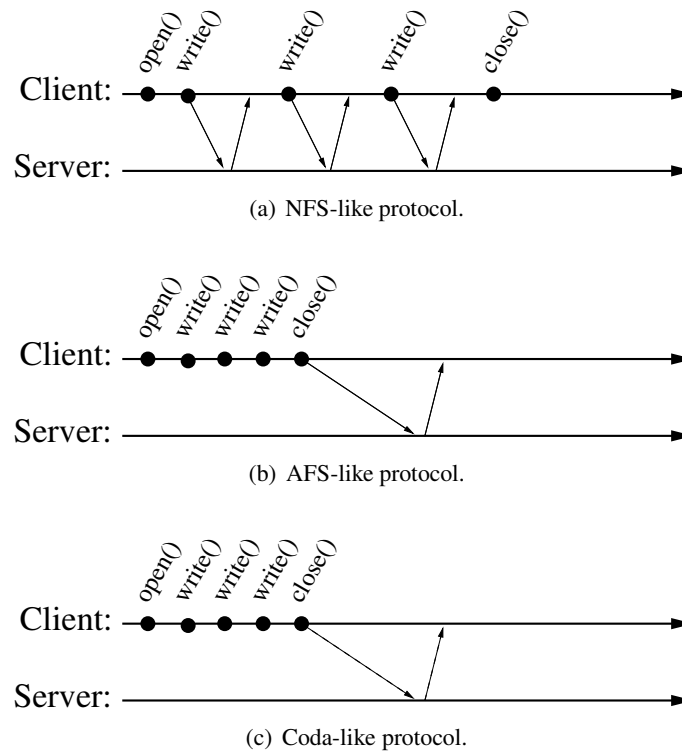


Figure 5.1: **Popular Write-back Strategies:** (a) Early remote file systems like NFS transmit data to the server on each `write()` operation. (b) AFS implements write-on-close. Clients buffer changes locally until the application closes the file; on `close()`, the client blocks until data can be transferred to the server. (c) Coda uses a write-on-close approach similar to AFS. To support weakly connected clients, Coda transfers data to the server asynchronously without blocking the application.

discussion.

Early remote file systems like NFS [SGK<sup>+</sup>85] implemented protocols that closely mirrored traditional file system APIs. Consequently, the approach to data write-back is closely related to file system's `write()` procedure. When an application accessing a file stored in NFS (version 2) invokes the file system's `write()` procedure, the client machine issues an NFS `write()` request to the file server. The NFS protocol stipulates that data must be stored durably, typically on disk, before the server can send a response. In this design, a client must issue many synchronous (blocking) requests when writing data to a file. Exacerbating the cost of write-back, NFS defines the maximum amount of data that can be transferred in a single request to be 8 KB. Although the NFS design is slow, if users disable data caching, it does preserve UNIX file system semantics which require that all clients read the most recent value written. (Recent versions of NFS do support techniques to reduce the cost of write-back; however, these versions are not as widely supported as the version 2



protocol.)

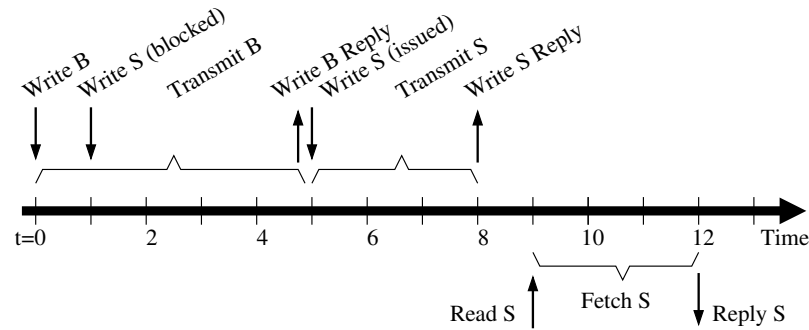
Attempting to support operation across the wide area, the designers of the Andrew File System (AFS) [HKM<sup>+</sup>88] developed an approach called write-back-on-close. In AFS, when an application opens a file, the client fetches the whole file from the server and caches it on the local disk. All `write()` operations update only the locally cached copy; the client does not propagate the changes to the server. Only when the application closes the file are all changes propagated to the server. With write-back-on-close, all changes are pushed to the server via a single message when the file is closed. The client blocks until the server receives the changes and responds to the request. Although the client must block to send only a single message across the network, the transmission time can be long if the message is large. AFS does not implement UNIX file system semantics, but the designers discovered that most applications work properly over AFS without modification.

The Coda project extended AFS to support disconnected [KS91] and weakly connected [MES95] clients. Owing to its heritage in AFS, Coda also implements write-back-on-close. To support weakly connected clients, Coda performs write-back asynchronously. That is, when an application closes a file, the client initiates a message to push the changes to the server but responds to the application immediately. As in AFS, the client sends only a single message to the server, but Coda gives the applications no guarantee as to when the data will be stored durably at the server. Coda calls its form of asynchronous write-back *trickle reintegration*. Clearly, Coda does not provide UNIX file system semantics.

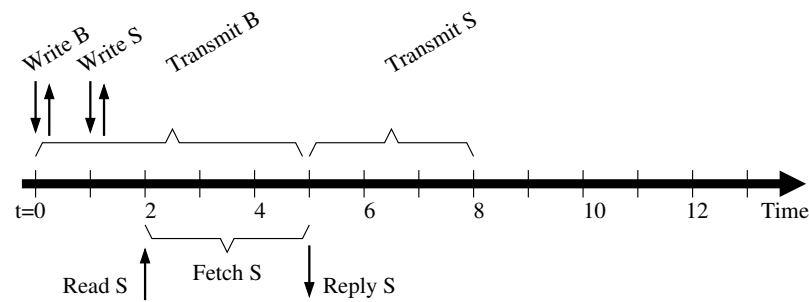
Different write-back strategies can have a profound effect on how users, especially weakly connected users, interact with a remote storage system. Consider the following scenario to understand how the use of remote storage system can be affected by write-back strategy. Assume a user saves a large file and then, after some small amount of think time, saves a smaller file. After the user receives a response to both requests, she informs a colleague of changes to the smaller file. The colleague then fetches the small file to view the changes.

Figure 5.2 illustrates how AFS's synchronous write-back and Coda's asynchronous write-back strategy respond to this scenario. We do not consider the response of NFS because its synchronous message on each `write()` operation make it unsuitable for operation across the wide area.

Figure 5.2(a) illustrates the behavior of a system that uses an AFS-like protocol with synchronous write-back. Though the user decides to write the small file at time  $t = 1$ , the operation is blocked until the changes to the large file are transmitted across the network and the first operation completes. The user is then delayed again as the changes to the small file are propagated to the



(a) AFS-like protocol.



(b) Coda-like protocol.

Figure 5.2: **An Example Demonstrating Different Write-back Strategies:** A user writes a big file, **B**, and a small file, **S**, and then shares the small file with another user. (a) In traditional file systems, like AFS, write operations block while changes are propagated to the server. (b) With Coda's trickle reintegration, write operations complete immediately and data is transferred to the server asynchronously in the background.

server. The blocking operations become more disruptive as a client's connectivity deteriorates. Poor responsiveness leads to user frustration and is a common reason that weakly connected users do not use remote file systems.

Figure 5.2(b) illustrates the behavior of a system using a Coda-like protocol with asynchronous write-back. Although trickle reintegration does improve user-perceived responsiveness, a key problem is that the state recorded at the server lags the user's view of the data. Stale server state impacts users sharing data with themselves or other users. For example, a user, believing that data is stored at the server after a write request completes, may notify a colleague of changes that have not yet propagated to the server. The co-worker may then retrieve a stale copy of the file, believing that it contains the most recent changes. In a similar scenario, if a user switches to a different device before changes are propagated to the server, they might be puzzled by changes that apparently disappeared.

Finally, note that powering down a device or terminating a network connection can further increase the likelihood that stale state at the server leads to conflicts. Such action may delay propagation of changes indefinitely. Thus, changes that would normally be propagated to the server in several seconds or minutes may be buffered at the client's device for several hours or more.

## 5.2 Using Intent to Decrease Latency

Intention updates optimize the write-back process to provide users with fast response times while maintaining accurate state at the server and enabling flexible scheduling of bandwidth usage.

The approach is motivated by the observation that, for each file, a file system maintains two distinct types of state: user data and metadata. User data is that data written by users via applications. Metadata records information about the state of user data, such as its size, the identity of the last user to modify the data, or the identity of a lock holder. As a fraction of bytes stored, a file system's contents are dominated by user data. User data is generally opaque to and uninterpreted by the file server. Metadata, on the other hand, consumes a small fraction of storage resources. File servers interpret and update metadata to record the state of user data and respond to client requests.

Typically, remote file systems update user data and metadata in a unified manner. That is, to propagate a change to the server, a client sends a single message. Upon receipt of the message, the server updates both metadata and user data to reflect the change. Most popular remote file systems like NFS [SGK<sup>+</sup>85], AFS [HKM<sup>+</sup>88], and Coda [SKK<sup>+</sup>90] implement unified updates.

It is not necessary, however, to unify the management of metadata and user data together in this manner. Intention updates decouple the management of metadata and user data by defining a two-phase process for propagating changes from client to server. This allows clients to communicate the existence of an update separately from the update itself. In the first phase, a client sends to the server an *intent* which notifies the server of a change requested by a user. An intent provides the server with enough information to update file system metadata to reflect a modification. The intent also includes a description of the user data associated with the update; this description may, for example, take the form of a cryptographically secure hash of the file after the modification. Upon receipt of an intent, a server evaluates whether the change summarized by the intent is permitted (by validating the request, checking file permissions, etc.), updates metadata to reflect the modification including recording the description of the user data provided by the client, and responds to the client indicating whether the intent was accepted or rejected. When the client receives a response to an

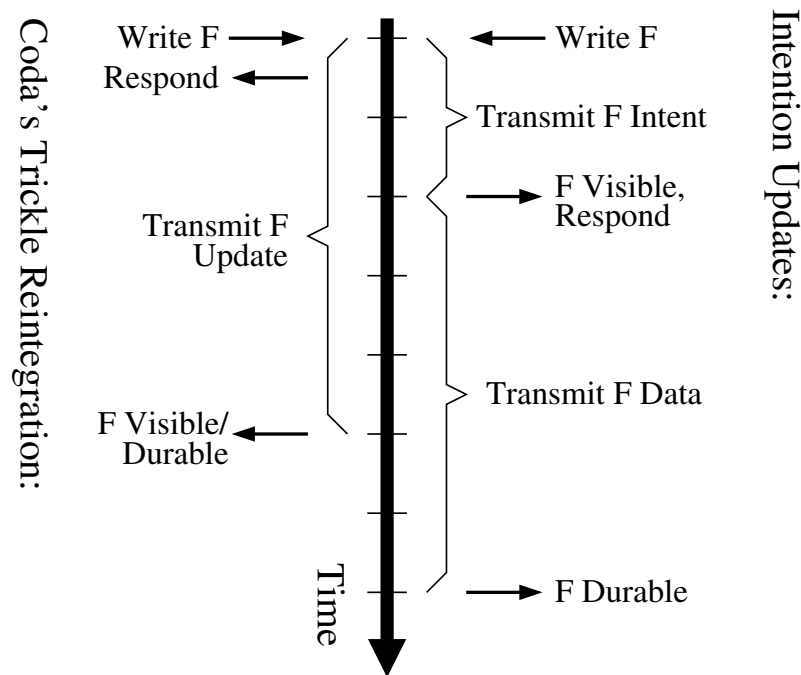


Figure 5.3: **A Comparison of Coda's Trickle Reintegration and Intention Updates:** With trickle reintegration, a response to the update request signals neither visibility or durability which are both tied to the transfer of the update message. Using intention updates, a response guarantees update visibility; durability is not assured until the completion of the data transfer phase.

intent, it responds to the corresponding application request. We will describe below the guarantees this decision provides clients.

After receiving the response to an intent, the client enters the second phase of the protocol. In the *data transfer* phase, the client transfers modified user data to the server for storage. When the server receives a data transfer message, it matches the message to a previous intent, checks that the user data in the message matches the description of the data in the intent, and stores the user data to disk.

A user may submit further update requests before the client has completed the data transfer phase of previous updates. To ensure fast response, the client prioritizes intent messages over data transfer messages. In this way, the client uses excess bandwidth to propagate user data to the server asynchronously.

Figure 5.3 compares intention updates with Coda's trickle reintegration, highlighting the differences in update visibility and durability. The distinction between update visibility and durability is detailed by Kim et al. [CN01]. In our model, an update is said to be *visible* if other clients in

the system can see the change. An update is said to be *durable* (or *safe*) when it is stored to disk at the server. With trickle reintegration, visibility and durability are entangled and depend on when the unified update reaches the server. With intention updates, visibility and durability are decoupled. An update is made visible to other clients when the server receives an intent; it is made durable when the server receives the corresponding bulk transfer message. With trickle integration, the application receives an immediate response to each request because it is deemed too expensive to block until the update is visible and durable. With intention updates, the client does not respond to the application until it receives a response to the intent. With this approach, a user knows that, when an operation completes, the changes are visible to all other clients. We will consider the trade-offs between response time, visibility, and durability quantitatively in Sections 5.5 and 5.6.

Improved update visibility, and indeed many other benefits of intentional updates, are possible because changes to metadata can typically be described much more concisely than changes to user data. Consequently, intents are small messages, especially relative to the subsequent bulk transfer messages. (For ease of illustration, figures in this paper show intents to be nearly as large as data transfer messages; in actuality, intents are much smaller.) Intention updates exploit the small size of intents to respond quickly to user requests while maintaining timely, accurate state at the server. It should be noted that the total amount of data transferred to update a file using intention updates is actually greater than the amount of data transferred using traditional unified updates because the messages must contain some state to enable the server to match data transfer messages to their associated intents. Intention updates are able to realize benefits over traditional updates despite this bandwidth penalty by prioritizing metadata traffic over bulk data transfer.

Let us return to the example of the previous section to understand how intention updates can improve file system access for weakly connected users. Figure 5.4 illustrates how the system would respond to the example scenario. After the user saves the small file, it is visible to all other clients in the system. Thus, when the colleague attempts to retrieve the changes in the small file, the system has enough information to recognize that the most recent version of the file is not available at the server. Such recognition may trigger one of several responses. In this example, the storage system blocks the colleague's request until it can ensure that the user is served the current state of the file.

This example reveals an additional opportunity for improving access. When a user attempts to read the file with changes that are visible but not yet durable, she must wait while the originating client completes the data transfer phase for the large file. However, since the server maintains a consistent record of modifications, the data transfer phases of multiple operations can

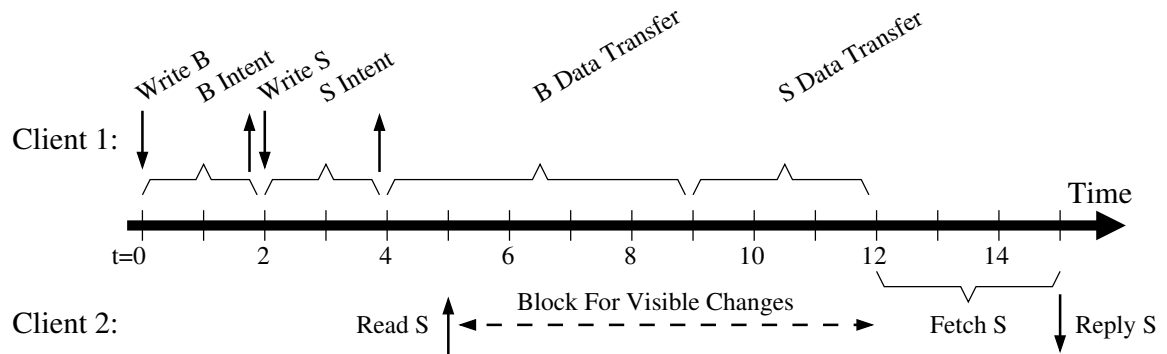


Figure 5.4: **Behavior of Intention Updates to Example Scenario:** Notice of writes by Client 1 are propagated to the server quickly, and the client does not receive a response until the update is visible. When Client 2 attempts to retrieve the changes to file *S*, the system blocks the request until data corresponding to the visible changes are available at the server, ensuring the Client 2 fetches the latest copy of the data.

be reordered. This can allow clients to reduce the time to durability for data that is considered more important (either because it is more valuable or because another user is requesting the data). We describe this and other variations in Section 5.3. The example also suggests that intention updates may introduce new failure modes for clients. In this example, the colleague must block for data to become available, which could take an arbitrarily long time if the originating client goes off-line. We discuss the issue of failure modes further in Section 5.4.

Finally, to help place intention updates in better context, it is useful to compare them to other popular techniques that clients use to manage communication with remote storage systems. In some systems, clients use locks or leases to gain exclusive access to an object stored in a remote storage system. A client holding a lock for an object is assured that no other user will be able to access the object. This promise allows lock holders to buffer changes locally or write changes back to the server lazily. Data need not be propagated to the server until the object is unlocked. Leases are similar to locks, but they grant exclusive access to a resource for only a time-bounded period of time. Although a client can use a lock to ensure that changes are made atomically, to prevent deadlock and starvation, all system components must follow locking (and unlocking) protocols. Furthermore, servers must be able to identify when lock holders die so that they can recover the lock.

Casting intention updates in terms of locks, one could consider an intent to be the combination of a lock request, a set of operations, and an unlock request. All operations in an intent are applied atomically to the object. With intention updates, however, the server does not require all

of an object's data before unlocking the object; rather, the server requires a description of all of an object's data. The client can continue to trickle data to the server even after the intent has unlocked the object for access by other clients. Because the unlock operation is intrinsic to the intent, an intent can never deadlock. Although the server is relieved of the responsibility of detecting failed lock holders and revoking locks, client failures can also cause complications in systems using intention updates. If the client that submitted the last intent fails before transferring the corresponding data to the server, other clients will receive error responses on subsequent accesses. To restore the system to regular operation, a client should initiate recovery by writing a new version to the file. Note that the server simply stores state provided by the clients and need not be aware when failure and recovery occur.

### 5.3 Intention Update Variations

The basic intention update mechanism can be varied to meet the requirements of other environments.

**Protocol variations:** As described in the previous section, a client initiates the data transfer phase only after receiving a response to the intent. This approach saves bandwidth when updates are rejected due to conflict. The intent response acts as an “early reject” message, reporting operation outcome before the client has committed to the relatively expensive data transfer phase. This variation benefits clients for whom bandwidth is limited or financially expensive. The cost of conserving bandwidth is in the latency to data durability. If, however, bandwidth is plentiful or cheap, a client may wish to initiate data transfer immediately after submitting the intent, before receiving the intent response. Such an approach should reduce the latency to data durability slightly at the cost of additional bandwidth when updates are rejected.

Finally, in very low-bandwidth or disconnected scenarios, or for applications that are very sensitive to delay, asynchronous intent updates may be appropriate. Asynchronous intention updates combine the approaches of standard intention updates and trickle reintegration. Clients propagate changes to the infrastructure in two phases, but both intents and data transfers are transmitted asynchronously. This variation enables applications to proceed after the update is recorded locally, but makes updates visible at the server more promptly with the small, high-priority intents. Of course, asynchronous intention updates do expose users sharing data to the possibility of accessing stale state from the server, however, the window of vulnerability should be shorter than with trickle rein-

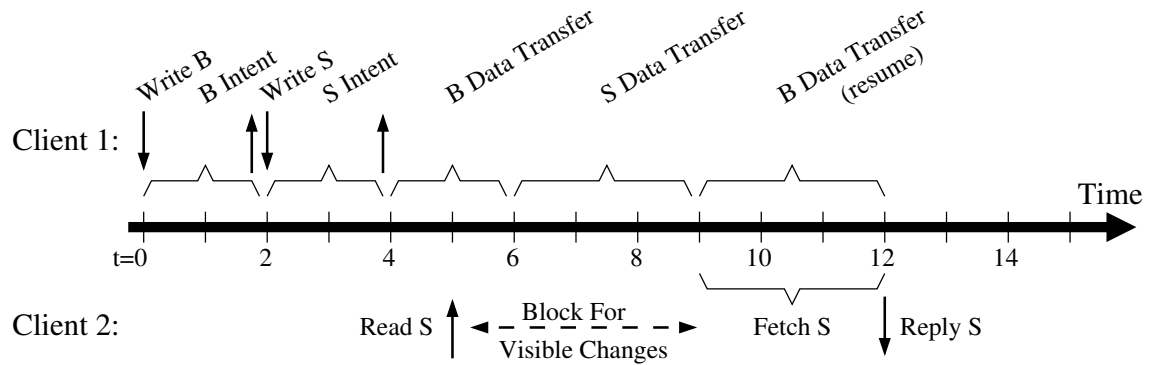


Figure 5.5: **Example Demonstrating the Benefits of Reordering and Fragmenting Data Transfers:** When a client requests data that is visible at the server but not yet durable, the server can suggest to the originating client that the requested data be transferred with higher priority. The originating client can execute data transfers in any order and even pausing in the middle of a transfer. Compare this example to Figure 5.4 to see how reordering and fragmenting transfers can facilitate sharing.

tegration. We will consider the trade-offs between response time, visibility, and durability in our evaluation.

**Re-ordering data transfers:** A client makes modifications visible by submitting an intent to the server and, thus, must submit intents in an order compatible with the system's consistency model. The client can, however, freely reorder the consummating data transfer messages. By reordering data transfers, a client can reduce the time to durability for data that is considered more “valuable”, by some metric. For example, a user may consider some files to be more financially valuable and wish to ensure that they are propagated to the server as quickly as possible. Alternatively, a versioning application may place greater value on the latest version of a file and, so, when multiple modifications are buffered at the client, elect to transfer the most recent version first. To support this type of re-ordering, a client must manage data transfer messages of multiple priorities.

As alluded to earlier, re-ordering data transfers can also facilitate data sharing. The server can reduce the amount of time that one client blocks waiting for changes from another client by requesting the data transfer message that includes the visible changes to be sent immediately. Re-ordering data transfer to support sharing requires extensions to both the client and the server. The server must recognize when a read request is blocked waiting on data that is visible but not durable. It must, then, determine the client responsible for the latest visible update and send a message to that client requesting the intent be consummated immediately. The client must be extended to handle



and respond to such requests by the server. Figure 5.5 illustrates how reordering data transfers can benefit clients. We will show how re-ordering data transfers can reduce time to durability for valuable data in Section 5.6.

**Varying intent detail:** An implementation can vary the level of detail in an intent, allowing a system to trade message size against descriptiveness. Varying the amount of information in an intent allows the system to optimize for a client’s available bandwidth and workload characteristics. Minimally, an intent must identify the object that is being updated and a description of the modification. Probably the most concise method for describing the effect of the update is a cryptographically secure one-way hash of the resulting object. An intent of this minimal form serves to make visible the existence of an update to a server but provides little additional information. Given such a terse intent, a server could provide little information about the object until it receives the consummating data transfer.

A more descriptive intent can contain other details such as the device on which the update was initiated and the complete updated file system metadata for the object. An even more verbose intent might include the byte ranges that are invalidated or even a complete file recipe [TKS<sup>+</sup>03] of the resulting file. Detailed intents allow the server to provide more complete responses to future requests, like metadata requests and consistency checks, even before receiving the associated data transfer. If the server can provide to a client a complete file recipe, the client could fetch the data directly from the originating client. Such functionality could be especially beneficial for clients co-located in a region of good connectivity that are separated from the server by weak network links.

**Fragmenting data transfers:** Just as data transfers can be re-ordered, they can also be subdivided into multiple smaller transfers. Data does not need to be delivered to the server atomically. Fragmenting data transfers can help support operation over connections with very low-bandwidth or frequent interruptions. A client could use fragmentation to propagate changes selectively; for example, it could send some changes from several versions to allow the server to construct the most recent version.

**Capping uncommitted data:** With a burst of activity, a client using intention updates can end up with a large number of modifications buffered locally. By monitoring the propagation of data transfers, a client can track the amount of data on a device that is not durable and throttle system

performance to limit risk due to loss of the device.

## 5.4 Implementation Issues

Several issues arise when implementing intention updates in real systems.

### 5.4.1 Architectural Requirements

To support intention updates, a system must modify both client and server code. To use intention updates, the client, for example, requires a networking API that supports messages of multiple priorities. Minimally, the network must differentiate between “high” priority intents and “low” priority data transfer messages. Clients can take advantage of additional priority levels to control the order of data transfer messages (as described below). The client also requires a persistent cache to record modifications until they are successfully transferred to the server. Such functionality could be provided by a component similar to Coda’s client modify log [MES95].

On the server side, intention updates require significant new functionality. Intents represent a new class of events that servers must handle. The server must maintain additional state to determine if the data that is actually described by the metadata is available in the file system or is still in transit to the server from the client. The server must also record state from each intent to enable it to match later data transfer messages.

Intention updates also require careful consideration of security issues. To secure any remote file system, the server must authorize the requests that it processes, especially those requests that modify state. Assume a system that uses public key cryptography for authentication. An intent changes state at the server and, thus, must be signed by the client to allow authorization. Assuming the intent is accepted, the subsequent data transfer message also updates state at the server. The data transfer message, however, need not be signed. If the intent includes a secure, one-way, collision-resistant hash of the data to be included in the transfer message, the server can confirm the integrity of the data transfer message without a signature. If, however, an intention update implementation represents the data summary differently, it may require the data transfer message to be signed. Because intention updates are designed to benefit weakly connected clients, implementations should consider the use of elliptic curve cryptography for its smaller signature size [Mil85].

### 5.4.2 Failure Modes

While intents do enable clients to propagate information about updates to the server quickly, they also, unfortunately, introduce several new failure modes.

In a system that uses traditional unified updates, if the server knows of the existence of a new version of an object, then it also has the data corresponding to the new version. This is not the case with intention updates. After a server has processed an intent and before the accompanying data transfer is completed, the server knows that a new version exists, but it cannot satisfy requests to read that new version. Taking this situation to the extreme, a server may record an intent that is never consummated by a data transfer. Clients may fail to send data transfer message for malicious or benign reasons.

Several different responses to such failures are possible. If the failure is transient (that is, clients are functioning correctly and the data transfer will be completed eventually), a client can take appropriate action depending on the requirements of the application. It could abort the read request, returning an error to the application. If the server supports versioning, the client could read an older version. If the latest intent includes sufficient information to determine which portions of the object are unchanged from prior versions, the system could return parts of the object that are known to be valid. Or, if the server can describe the data unambiguously (for example, by providing a secure hash of the data that has yet to be transferred), the reading client could fetch the data from the writer directly.

The system must also provide some means to recover from non-transient failures. To allow progress after a permanent failure, the system must allow a client to overwrite the unconsummated version. The version used in recovery could be a previous version of the object or a new version created by the recovering client.

## 5.5 Analytical Evaluation

We begin our evaluation of intention updates analytically by applying methods from queueing theory. Queueing theory studies the stochastic flow of commodities or events through a channel or service of finite-capacity. The correspondence to our system is as follows: events are intents or data transfer messages and the finite-capacity channel is the network. We assume that the cost of processing a message is equal to its network transmission latency; we assume that processing at the server is negligible.

We will argue that intention updates can be modeled effectively as a fixed priority queueing system that implements preemptive resume. We will analyze the behavior of the system in equilibrium and use approximation methods to understand transient response to bursty workloads.

### 5.5.1 Preliminaries

We begin by describing the parameters that control a queueing system. Additional details can be found in a traditional queueing theory book (e.g. [Kle75]).

Central to the study of queueing theory is the rate at which events enter and leave the system. The average arrival rate of events to the system is denoted by  $\lambda$ . The average rate at which events are serviced is denoted by  $\mu$ . It is often easier to consider the average amount of time required to service a single event,  $\bar{x} = 1/\mu$ .

The fraction of time that a server spends processing events is the *utilization factor*,  $\rho = \lambda/\mu$ . To ensure that a server can process events in finite time, a system must enforce that  $\rho < 1$ .

The key quantities of interest in the study of queueing systems are the number of events in the queue, the average amount of time an event spends in the queue waiting for service, and the average amount of time an event spends in the whole system. These measurements are known as the queue length ( $N$ ), the queue waiting time ( $W$ ), and the system time ( $T$ ). From these definitions, it is clear that  $T = W + \bar{x}$ .

Key to our proposed method of intention updates is that intents are processed preferentially over data transfers. We can model this preferential service as a priority queueing system. To analyze a priority queueing system, we must extend the system model.

We employ a basic priority queueing system in which events are assigned to one of  $P$  different priority classes designated with subscript  $p$  ( $p = 1, 2, \dots, P$ ). Events of class  $p = 1$  are the lowest priority events and events of class  $p = P$  are the highest priority events. Events of higher priority are serviced before events of lower priority. Events of equal priority are serviced in a first-come, first-served (FCFS) manner.

The average arrival rate for events of priority  $p$  is denoted  $\lambda_p$  and  $\sum_{i=1}^P \lambda_i = \lambda$ . Analogously, the average service time for events of priority  $p$  is denoted  $\bar{x}_p$ . The average service time for all events is the weighted average of the service times of different event classes  $\bar{x} = \sum_{i=1}^P \lambda_i \bar{x}_i / \lambda$ . Finally, the fraction of time a server spends processing events of priority  $p$  is  $\rho_p = \lambda_p \bar{x}_p$ , and the total utilization factor is computed as  $\rho = \sum_{i=1}^P \rho_i$ .

### 5.5.2 Preemptive Priority Queueing

Much of queueing theory studies the steady-state or equilibrium behavior the systems, and this too is where will begin our analysis. Frequently, we must simplify the system model to make analysis of queueing systems tractable. One common simplification, which we adopt, is to assume that the intervals between consecutive events (the interarrival times) are exponentially distributed.

With this assumption, we can use the results of the M/G/1 queueing model. This shorthand indicates that the system has an exponential (**M**emoryless) arrival distribution, **G**eneral service distribution, and a single (**1**) server.

First, consider an M/G/1 queue that process requests FCFS. In our application context, this corresponds to a system that implements traditional, unified updates. In such a system, the average amount of time an event spends in the system is given by

$$T = \bar{x} + \frac{\lambda \overline{x^2}}{2(1 - \rho)}$$

where  $\overline{x^2}$  is the second moment of the service time distribution [Kle75].

Next, let us consider how to extend the M/G/1 model to reflect the use of intention updates. To model intention updates we adopt a fixed priority queueing system that implements *preemptive resume*. In a preemptive priority queueing system, when an event enters the system and finds an event of lower priority in service, the newly arriving, higher-priority event evicts the low-priority event from the server and begins service immediately. In the preemptive resume model, a preempted event does not lose credit for the portion of its service that has already been performed; when it resumes service, it picks up where it left off.

This model reflects our application well. In our application, low-priority messages are generated during the data transfer phase. Data transfer messages transfer blocks between the client and the server. A client can preempt the data transfer phase at any block boundary to send a high-priority intent message. After processing the intent, the client does not need to re-start the data transfer phase. It only needs to continue the transfer from where it was interrupted.

Figure 5.6 illustrates a simple model of our proposed system, showing how it relates to the priority queueing with preemptive resume model. Each user-level request is converted into two network-level requests of different sizes. The different types of requests are, at least logically, queued separately. The requests in the intent queue require less service and are given priority over the larger events in the data queue that require longer to service.

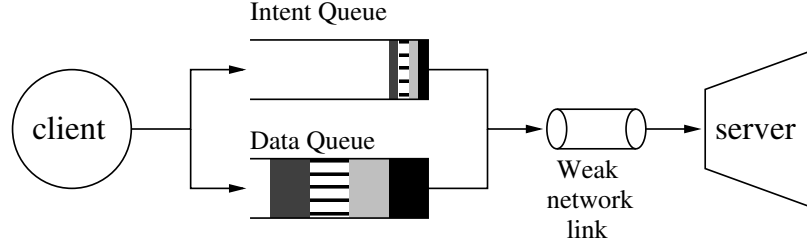


Figure 5.6: **Modeling Intention Updates:** Updates are divided into two events of different priority and different size. The service time is assumed to be the transmission delay due to the weak network connection between the client and the server.

For an M/G/1 priority queue that services events of  $P$  classes using a preemptive resume strategy, the average system time of an event of priority  $p$  is given by

$$T_p = \frac{\bar{x}_p(1 - \sigma_p) + \sum_{i=p}^P \frac{\lambda_i \bar{x}_i^2}{2}}{(1 - \sigma_p)(1 - \sigma_{p+1})}$$

where  $\bar{x}_p^2$  is the second moment of the service time distribution for events of priority  $p$  and  $\sigma_p = \sum_{i=p}^P \rho_i$  [Kle76]. Recall that  $p = 1$  is the lowest priority, and  $p = P$  is the highest priority.

Our application uses only two event classes. We use the subscript  $D$  to refer to quantities related to data transfer events. Data transfer messages are low-priority events with  $p = 1$ . We use the subscript  $I$  to identify quantities related to intents events. Intent messages are high-priority events with  $p = 2$ . Note, because each application-level update is divided into an intent and data transfer message,  $\lambda_I = \lambda_D$ . Using this notation, the average system time for a high-priority intent is given by

$$T_I = \frac{\bar{x}_I(1 - \rho_I) + \frac{\lambda_I \bar{x}_I^2}{2}}{(1 - \rho_I)}$$

and the average system time for low-priority data transfer events is given by

$$T_D = \frac{\bar{x}_D(1 - \rho_I - \rho_D) + \frac{\lambda_I \bar{x}_I^2}{2} + \frac{\lambda_D \bar{x}_D^2}{2}}{(1 - \rho_I - \rho_D)(1 - \rho_I)}$$

Figure 5.7 illustrates how these results apply to our system. We assume a client with 128 kbit/s of upload bandwidth processing updates with average size of 10 KB. When using intention updates, the average intent size is 1 KB while the average transfer size remains 10 KB—intents are purely overhead. The service time curves follow a normal distribution and are computed from the

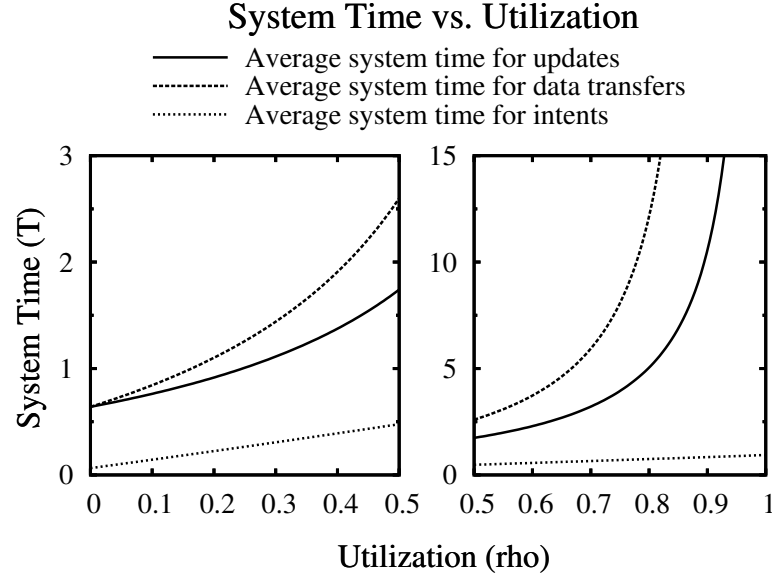


Figure 5.7: **Analysis of Average System Time:** The client has 128 kbit/s of upload bandwidth and submits updates with average size of 10 KB. When using intention updates, the intent is 1 KB. Message sizes are normally distributed with  $\sigma^2 = 1$ . The plot has been split to reveal detail at both upper and lower limits of utilization.

upload bandwidth and message size with variance  $\sigma^2 = 1$ . The graph shows the average of the total system time as the load (utilization of the link) increases. The  $x$ -axis reflects the utilization of the base system, using traditional unified updates; because intents represent an additional load to the system, the behavior of the intent-based system becomes unstable before  $\rho$  reaches 1.

Most generally, Figure 5.7 shows that as the system becomes more loaded, the system time increases. The increase is especially dramatic as the system approaches full utilization. With intention updates, intents are serviced quickly, even as the load increases. This ensures that the server is notified of changes to objects quickly. The cost of prompt notification is delay in uploading changes to the server, reflected in the difference between the service time curves for unified updates and data transfers.

### 5.5.3 Fluid Approximation

The assumption of exponentially distributed interarrival times is reasonable for many real-world phenomena (like phone calls arriving at a switch board or alpha particles emitted from a radioactive material). It is not, however, a particularly compelling model for file system workloads.

Instead, studies of file system usage in production environments have shown that workloads are typically bursty [BHK<sup>+</sup>91].

Even if we could describe these workloads mathematically, a steady-state analysis like we performed in Section 5.5.2 would be impenetrably complex and unlikely to yield any significant insight. To understand the system response to more realistic workload models, we rely on approximation techniques from the queueing theory literature. Specifically, we will extend the fluid approximation for simple queueing systems to model systems with multiple event classes of different priorities. Although the fluid approximation is a well-known tool in queueing theory texts and literature, we are not aware of any prior work that uses the fluid approximation to analyze systems with events of multiple priorities the way we describe here. Using this approach, we can study the system's transient response to various workloads.

To model bursty workloads, we allow the request arrival rate to vary with time,  $\lambda(t)$ . (The capacity of the server may also vary with time, such as when network conditions change, but fluctuations in the service rate are typically less than the arrival rate. In this analysis we will assume the server has a fixed maximum capacity of  $\mu$ .) From steady state analysis, we know that in the limit the utilization factor must not exceed unity. In this transient analysis, however, we are most interested in the system response during periods of heavy load where the utilization factor  $\rho(t) = \lambda(t)/\mu$  exceeds unity.

Let us begin by understanding the fluid approximation for queues. The fluid approximation [Kle76] is most useful for understanding the response of a system in overload. It estimates the discrete, discontinuous flow of events in the system as a continuous flow without stochastic variation. Consider, first, the discrete case. Let

$$\begin{aligned}\alpha(t) &= \text{Number of request arrivals in } (0, t) \\ \delta(t) &= \text{Number of request departures in } (0, t)\end{aligned}$$

The number of events in the system, then, is given by

$$N(t) = \alpha(t) - \delta(t)$$

The fluid approximation replaces the discontinuous stochastic processes  $\alpha(t)$  and  $\delta(t)$



with continuous deterministic processes  $\overline{\alpha(t)}$  and  $\overline{\delta(t)}$  where

$$\begin{aligned}\overline{\alpha(t)} &= \int_{\tau=0}^t \lambda(\tau) d\tau \\ \overline{\delta(t)} &= \int_{\tau=0}^t \mu(\tau) d\tau\end{aligned}$$

and the time-varying function for the amount of service performed is given by

$$\mu(t) = \begin{cases} \mu & \text{for } N(t) > 0 \\ \lambda(t) & \text{for } N(t) = 0 \end{cases}$$

Analogously, the number of customers in the system, is given by

$$N(t) = \overline{\alpha(t)} - \overline{\delta(t)}$$

Figure 5.8 illustrates the application of the fluid approximation. Figure 5.8(a) plots the flow rate for events in the system; it shows that the client demand exceeds the system's service capacity during the period  $3 < t < 6$ . The fluid approximation estimates the queue length to be the difference between the cumulative number of requests that have entered the system and the cumulative number of requests that have left the system. Figure 5.8(b) plots the cumulative number of arriving and departing requests. The length of the queue is represented by the difference between the two curves, which is also plotted in Figure 5.8(c). Notice that the queue begins to form when the instantaneous utilization factor,  $\rho(t) = \lambda(t)/\mu$ , exceeds unity. The length of the queue continues to grow until  $\lambda(t)$  drops back below  $\mu$ , at which point the queue is drained. The rate at which the system empties the queue is determined by the amount of excess capacity  $\mu - \lambda(t)$ .

It is important that we understand the limitations of the fluid approximation. The fluid approximation accurately predicts queue lengths for systems in overload. The weakness of the approximation is that it neglects the formation of queues due to variability in event arrival *before* overload. However, if we assume that the arrival rate is either above or significantly below the service rate at all times—that is, client behavior can be approximated as a process that is either “on” or “off”—then queueing due to stochastic variations in arrival rate is minimal, and this approximation models the system accurately.

Now let us extend the fluid approximation to handle multiple event classes. We will again assume that each event is assigned to one of  $P$  different priority classes and each event class has an

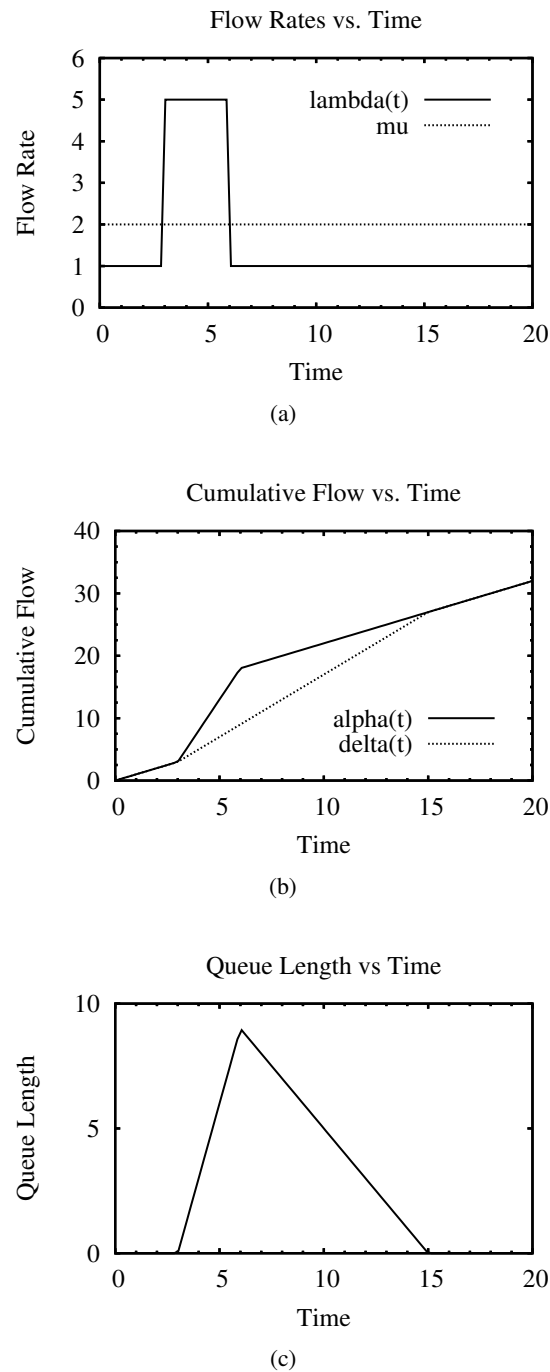


Figure 5.8: **Fluid Approximation Example:** An example illustrating the utility of the fluid approximation for queues. (a) During overload, a time-varying arrival rate can exceed the service rate. (b) To compute the queue length, we first compute the cumulative number of event arrivals and departures. The length of the queue is represented by the area between the curves. (c) A queue begins to form when the instantaneous utilization factor exceed unity and is at its longest when utilization factor drops back below the service rate.

accompanying service time distribution. To accommodate these features, rather than tracking the flow of requests in the system, we track the flow of *work units* in the system. Let  $u_p(t)$  be the arrival rate of work units of priority  $p$  and  $w_p(t)$  be the rate that work units of priority  $p$  are serviced.

Let  $U_p(t)$  represent the cumulative amount of work of priority  $p$  that has entered the system and  $W_p(t)$  be the cumulative amount of work of priority  $p$  that has been serviced. That is,

$$\begin{aligned} U_p(t) &= \text{Number of work units} \\ &\quad \text{of priority } p \text{ arriving in } (0, t) \\ W_p(t) &= \text{Number of work units} \\ &\quad \text{of priority } p \text{ serviced in } (0, t) \end{aligned}$$

As before, we replace the discontinuous stochastic processes  $U_p(t)$  and  $W_p(t)$  with continuous deterministic processes  $\overline{U_p(t)}$  and  $\overline{W_p(t)}$  where

$$\begin{aligned} \overline{U_p(t)} &= \int_{\tau=0}^t u_p(\tau) d\tau \\ \overline{W_p(t)} &= \int_{\tau=0}^t w_p(\tau) d\tau \end{aligned}$$

The number of work units of priority  $p$  waiting in the queue is given by

$$N_p(t) = \overline{U_p(t)} - \overline{W_p(t)}$$

To apply this result to evaluate a queueing system, we must define the rate that work units arrive and depart. The arrival rate of work depends on the rate that requests enter the system and the amount of work required to service each request. In our particular domain, the amount of work needed to service a request could be considered a property of the application, while the request rate could be considered a function of how much processor time is allocated to the application. Let  $\lambda_p(t)$  be the rate that events of priority  $p$  enter the system, and assume that each event of priority  $p$  requires  $\omega_p$  work units to service. Then  $u_p(t) = \omega_p \lambda_p(t)$ .

The rate that work of varying priority departs the system depends on how the system allocates its resources to servicing requests of different priority. Assume a single server can process  $\mu(t)$  work units per unit time. A server services work of priority  $p$  at a rate of  $\mu_p(t)$ . Of course,  $\sum \mu_p(t) \leq \mu(t)$  for all  $t$ . By this definition,  $w_p(t) = \mu_p(t)$ . A system implementing preemptive

priority resume would allocate service to different priority classes as defined below.

$$\mu_p(t) = \begin{cases} \mu(t) - \sum_{k=p+1}^P \mu_k(t) & \text{for } N_p(t) > 0 \\ u_p(t) - \sum_{k=p+1}^P \mu_k(t) & \text{for } N_p(t) = 0 \end{cases}$$

This definition states that service will be allocated to events of the highest priority first. The amount of service allocated to a given priority class will not exceed the work load imposed by that class. If the system has excess capacity after servicing events of the highest priority, the remainder of the capacity may be allocated to requests in lower priority classes. Of course, the amount of work allocated to service requests of any event class must be non-negative.

We can now estimate the behavior of our system modeled as in Figure 5.6. Let  $\omega_2$  be the number of work units required to service the high-priority intent of the first phase of our schedule. Let  $\omega_1$  be the number of work units required to complete the low-priority data transfer of the second phase of our schedule. The weak network link can transmit  $\mu$  work units per unit time. Because each application-level request is converted into two network-level requests,  $\lambda_1(t) = \lambda_2(t)$ .

Consider the example in Figure 5.9. In this example, the work contributed by an intent is  $\omega_2 = 1$ ; the work required to service each data transfer is  $\omega_1 = 4$ . The system can service  $\mu = 2$  work units per unit time. Figure 5.9(a) plots the instantaneous request rate in the system showing bursts of activity at  $3 < t < 5$  and  $13 < t < 15$ . Figure 5.9(b) shows the cumulative flow of work in the system, as well as the total amount of work and the amount of work that has been serviced. As in the example of Figure 5.8, the queue lengths in the system can be extracted by computing the difference between the curves. Figure 5.9(c) shows the queue lengths corresponding to intents and data transfers.

Figure 5.10 shows the analysis of the queue length without using intention updates. In this analysis, we have assumed that each user-level request requires  $\omega = 4$  units of service. A comparison of this analysis with Figure 5.9(c) is thus conservative since a request requires  $\omega_1 + \omega_2 = 5$  units of service when using intention updates. That is, we assume that the cost of processing intents is purely overhead compared to the base system. In comparing the graphs of queue lengths, note the time that the client receives a response to the last request. With the traditional approach, the response to the last request in the first burst is sent at time  $t = 23$  (computed by extrapolating the first downward sloping segment of the curve to the x-axis). The response to the last request in the second burst is sent when the queue is drained at time  $t = 35$ . With intention updates, the response to the last

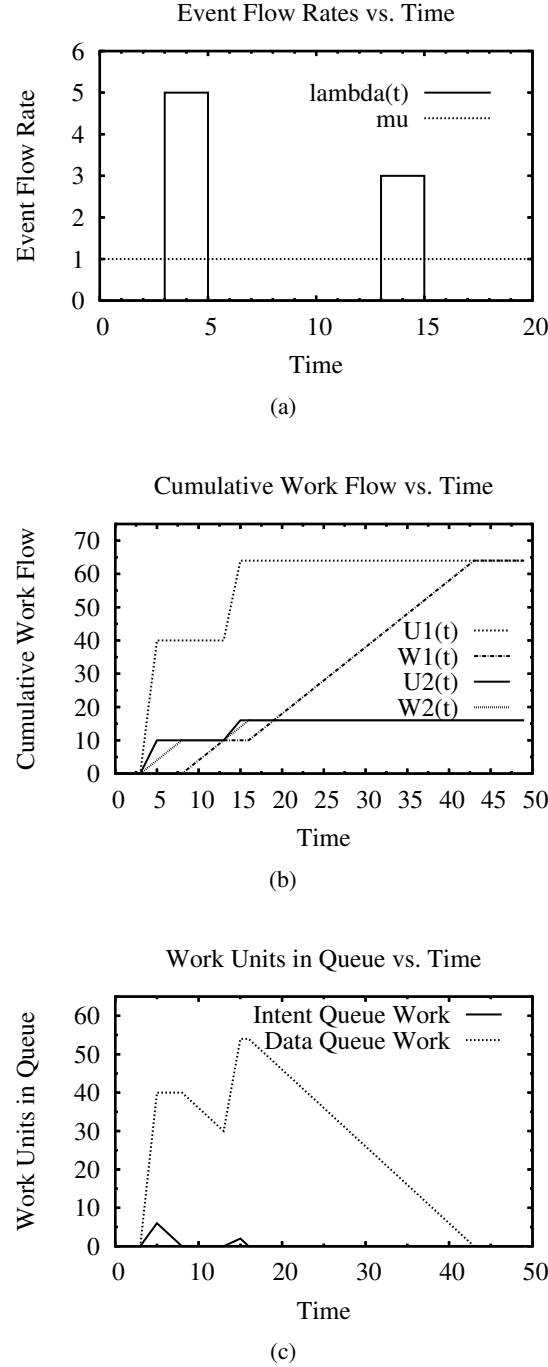


Figure 5.9: **Approximating Queue Length with Intention Updates:** (a) The request stream exhibits bursts of activity at  $3 < t < 5$  and  $13 < t < 15$ . (b) Each request is converted into two network-level requests that each require a different amount of work to process. The graph plots the cumulative amount of work in the system of each type and the amount of work serviced. (c) With the first burst of activity, queues containing both types of requests begin to form. The intent queue is drained, and the client receives a response to the last request of the first burst, at time  $t = 8$ . The system then works to transfer data until it is interrupted by the next burst. The client receives a response to the last request of the second burst at time  $t = 16$ . Data transfer is finally completed at time  $t = 43$ .

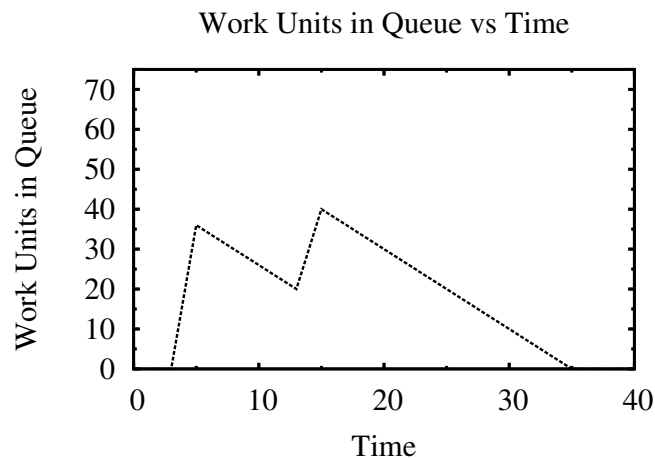


Figure 5.10: **Approximating Queue Length with Unified Updates:** An analysis of the queue length of the system shown in Figure 5.9 assuming traditional, unified updates. The queue is drained, and the client receives a response to the last request, at time  $t = 35$ .

request of the first burst is sent when the *intent* queue is drained at time  $t = 8$ . The response for the last request of the second burst is sent at time  $t = 16$ . By requiring the client to block only until the intent is processed, response time is kept low. Of course, with intention updates, the improvement in response time comes at the expense of durability. The modifications are not completely durable until the data queue is drained at time  $t = 43$ . Until that time, some data is stored only at the client and is susceptible to loss.

## 5.6 Evaluation via Simulation

Next, we consider how simulation can improve our understanding of the behavior and consequences of intention updates. With simulation, we can compare the behavior of intention updates against other protocols currently in use. Also, we can test with realistic workloads from traces that have been widely studied in the literature.

### 5.6.1 Simulation Methodology

Figure 5.11 illustrates the simulator architecture. A trace generator feeds an identical stream of requests to a “reference” system and a “test” system. The trace generator may output synthetic traces or real traces from the literature (e.g. [Har93, RLA00]). Each subsystem consists of one or more clients communicating with a single remote server. Each client has a dedicated

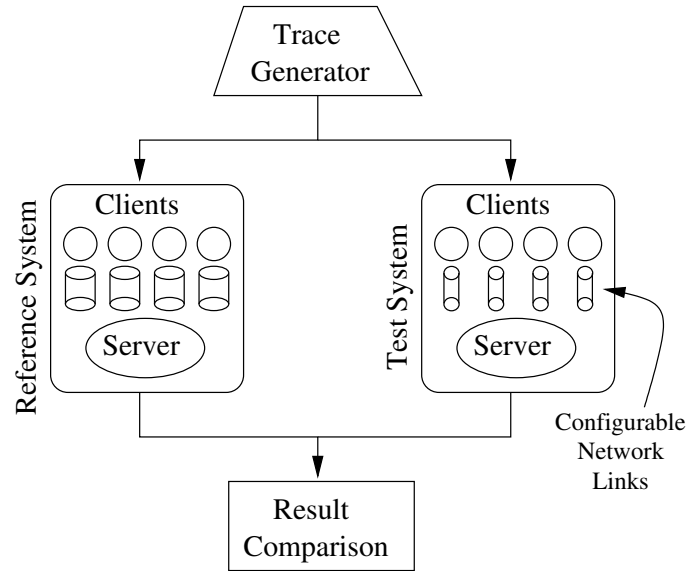


Figure 5.11: **Simulator Architecture:** The simulator executes an identical stream of requests on two systems in parallel. Typically, the “reference” system is configured to simulate ideal network links (with no latency and infinite bandwidth), while the “test” system models more realistic connections. Servers process requests without adding delay. After the event is executed in both system, the simulator compares the outcomes and records the results.

connection to the server. Message delay is computed using the familiar  $\alpha + \beta \cdot n$  model, where  $\alpha$  corresponds to link’s latency,  $\beta$  corresponds to the link’s bandwidth, and  $n$  is the size of the message in bytes. We typically configure the “reference” system to use “ideal” network links with infinite bandwidth and no latency. Other network models we use are shown in Table 5.1. In both subsystems, we assume that the server is infinitely fast; thus, a request’s service time is dependent only on the network transmission delay. After an event is executed in both subsystems, the simulator compares the results and records the outcome.

We record the absolute metrics of response time, durability latency, and visibility latency.

Network Type	Latency (ms)	Bandwidth (kbit/s)
Ideal	0	$\infty$
Telephone Modem	100	56
Cable Modem	30	384 down/ 128 up

Table 5.1: **Network parameters:** Message delay is computed based on a link’s latency and bandwidth. The parameters corresponding to different networking technologies are shown in the table.

We also record, to compare the two systems, relative metrics including response time penalty, durability latency penalty, and visibility latency penalty. See Section 2.2 to review how these metrics are defined.

### 5.6.2 Modeling Client Protocols

In addition to modeling the behavior of intention updates, we model the protocols of several well-known remote file systems. The *NFS* model exemplifies a stateless remote file system like NFS [SGK<sup>+</sup>85] where `open()` and `close()` operations are performed locally. Our NFS model does not perform any caching and thus transfers data between the client and server on all `read()` and `write()` operations. The *AFS-poll* model is based on the initial version of AFS [HKM<sup>+</sup>88]. It implements whole-file caching; modifications are buffered locally and propagated back to the server on `close()`. To ensure consistency, the client polls the server on each `open()` operation. The *AFS-cb* model is similar to the AFS-poll model, but it uses callbacks to maintain consistency. While callbacks were originally designed to improve scalability by reducing the number of requests that must be handled by a server, they also benefit weakly connected clients by reducing the number of messages that must traverse the weak link. Finally, the *Coda-wc* model implements Coda's strategy for supporting weakly connected clients [MES95]. Built atop AFS, Coda also uses whole-file caching with write-back on close. To support weakly connected clients, Coda introduced trickle integration which propagates data to the server asynchronously. All of these models use a unified message to communicate metadata and data changes.

In evaluating the simulation results, we focus on how different protocols respond to store operations because that is where the protocols differ most markedly. The file system operation that actually transfers the data to the server differs between the protocols. For NFS, we focus on `write()` operations; for models based on whole-file caching, we monitor the `close()` operation.

We can predict much about client behavior even before running any simulations. To predict the performance of different protocols, we calculate the expected latency to access and store data in the remote storage system based on the network messages required by the protocol. Our model uses the parameters defined in Table 5.2. Let  $\alpha_u$  and  $\alpha_d$  signify the latency of the uplink and downlink, respectively; let  $\beta_u$  and  $\beta_d$  be the bandwidth of the uplink and downlink; and assume a message that transfers data is of size  $M_t$  while other small protocol messages (such as consistency check, intent, and acknowledgement messages) are of size  $M_s$  bytes. From these definitions, we can describe the read and write latency for each protocol.



Parameter	Definition
$\alpha_u$	latency of the uplink
$\alpha_d$	latency of the downlink
$\beta_u$	bandwidth of the uplink
$\beta_d$	bandwidth of the downlink
$M_t$	size of message that transfers data
$M_s$	size of “small” message that does not transfer data

Table 5.2: **Parameters for Modeling Remote File System Performance:** To estimate the performance of different remote storage system protocols, we estimate the latency to access and store data in the system based on the network messages required by the protocol. Our analysis depends on the parameters define in this table.

Client Protocol	Access Latency
NFS	$\alpha_u + M_s/\beta_u + \alpha_d + M_t/\beta_d$
AFS-poll (uncached)	$\alpha_u + M_s/\beta_u + \alpha_d + M_t/\beta_d$
AFS-poll (cached)	$\alpha_u + M_s/\beta_u + \alpha_d + M_s/\beta_d$
AFS-cb Coda (uncached) Intent	$\alpha_u + M_s/\beta_u + \alpha_d + M_t/\beta_d$
AFS-cb Coda (cached) Intent	0

Table 5.3: **Computed Read Latencies:** The cost of accessing data depends on the protocol used by the client and whether the client already has a valid copy in the cache. For NFS, the computations apply to the `read()` operation; for other protocols based on whole-file caching, the computations apply to the `open()` operation. The cost of accessing data depends primarily on the transmission delay of the reply message, which depends on whether the server must supply the client with the requested data or merely validate the copy in the client’s cache.

Client Protocol	Store Visibility	Store Durability	Store Response
NFS	$\alpha_u + M_t/\beta_u$	$\text{Vis}_{NFS}$	$\text{Vis}_{NFS} + \alpha_d + M_s/\beta_d$
AFS-poll	$\alpha_u + M_t/\beta_u$	$\text{Vis}_{AFS-poll}$	$\text{Vis}_{AFS-poll} + \alpha_d + M_s/\beta_d$
AFS-cb	$\alpha_u + M_t/\beta_u$	$\text{Vis}_{AFS-cb}$	$\text{Vis}_{AFS-cb} + \alpha_d + M_s/\beta_d$
Coda-wc	$\alpha_u + M_t/\beta_u$	$\text{Vis}_{Coda-wc}$	0
Intents	$\alpha_u + M_s/\beta_u$	$\text{Resp}_{intent} + \alpha_u + M_t/\beta_u$	$\text{Vis}_{intent} + \alpha_d + M_s/\beta_d$

Table 5.4: **Computed Write Latencies:** We analyze the cost of store operations using three metrics: visibility, durability, and response time. For NFS, the computations apply to the `write()` operations; for AFS-based protocols, the computations apply to the `close()` operation. For traditional approaches that use unified updates, the visibility latency and durability latency are equal. Intention updates reduce visibility latency and response time by sending only a small intent to the server to make changes visible. The costs shown for  $\text{Vis}_{Coda-wc}$ ,  $\text{Dur}_{Coda-wc}$ , and  $\text{Dur}_{intent}$  represent minimum possible costs; costs can increase if other requests stall low-priority write-back traffic.

Tables 5.3 shows the read latency for the various protocols, depending on whether the client has the document cached, in terms of the parameters of Table 5.2. Most protocols require the client to contact the server when accessing data. When the client communicates with the server, it must wait for the latency and transmission delay of both the upstream request and the downstream reply. The upstream and downstream latency is a property of the link, not the protocol, and thus all protocols pay the same cost ( $\alpha_u + \alpha_d$ ). The transmission delay depends on the size of the message being sent. In our model, in which all protocol messages have the same size ( $M_s$ ), the transmission delay for the request is the same for all protocols ( $M_s/\beta_u$ ). The size of the reply message, however, depends on whether the server needs to supply the client with data or merely validate the copy in the client’s cache. Only the latter is needed in the AFS-poll model when the client has the data cached, and thus the transmission delay of the reply is  $M_s/\beta_d$ . For all other scenarios we model, the server must supply the client with data and the transmission delay of the reply is  $M_t/\beta_d$ . Of course, for callback-based protocols (AFS-cb, Coda, and Intent), once the client has the data cached, it can access the data again without contacting the server until it receives a callback for the object. Details of the access costs are shown in Table 5.3.

For operations that write data to the server, we calculate three key points for each operation: when the application receives a response, when the data is visible, and when the data is durable. Table 5.4 shows the results of those computations. For the traditional protocols that use unified updates (NFS, AFS-poll, AFS-cb, and Coda-wc), the update becomes visible when the server receives the write request. All of these protocols send the new data in the write request, and thus, the message latency is  $\alpha_u + M_t/\beta_u$ . Because the data is included in the request, the durability latency is

equal to the visibility latency. In most cases, the client blocks until it receives a response, a small protocol message, from the server. In the Coda-wc protocol, however, the client does not block at all on write requests. The behavior of a system using intents differs greatly. In an intent-based system, updates are visible after the server receives the intent message. We consider an intent a protocol message, and thus it is much smaller ( $M_s$ ) than a message that includes the new data ( $M_t$ ). By virtue of the small size of an intent, the visibility latency is lower for systems using intention updates. After submitting an intent, the client blocks until it receives a response to the intent. That is, the client waits until the server responds with a small protocol message. The update becomes durable when the server receives the data transfer message. At the earliest, the client can send the data transfer message after it receives the intent response. The update is durable after an additional time of  $\alpha_u + M_t/\beta_u$ . The durability latency can increase, however, if the data transfer message is preempted by other messages.

From these computations, we can predict the general trends that the simulator will reveal. For example, given that  $M_s \ll M_t$ , we expect clients using intention updates to be more responsive than clients using other protocols. Also, because of the high cost of accessing data in the NFS and AFS-poll protocols, we expect these protocols to exhibit significant cumulative delay.

### 5.6.3 Simulating Synthetic Workloads

We begin our simulation studies by driving the simulator with a synthetic trace generator. Using a synthetic workloads helps us identify the salient characteristics that distinguish the different protocols.

The *bursty* traffic generator models clients that alternate between idle periods and periods of activity, as shown in Figure 5.12. The *cycle time* determines the amount of time between consecutive periods of activity, and the *bursty period* dictates the length of the busy period. The *average interarrival time* describes the event arrival rate averaged over the entire cycle. The event arrival time is scaled by the relative length of the bursty period to ensure the specified average interarrival time. Additional parameters control other aspects of the generated workload like the size of the client's file set, the file size distribution, and the read/write ratio.

First, consider the response to a bursty workload with files of uniform size. Figure 5.13 shows the simulation results to a bursty workload with 10-minute cycle time, 30-second bursty period, and 10-second average interarrival time. Each client accesses a file set of 1000 4KB files; 20% of accesses write data. Clients in the reference system use ideal links; clients in the test system

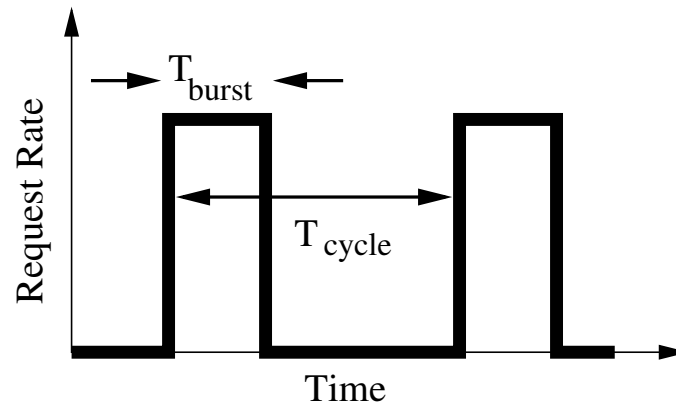


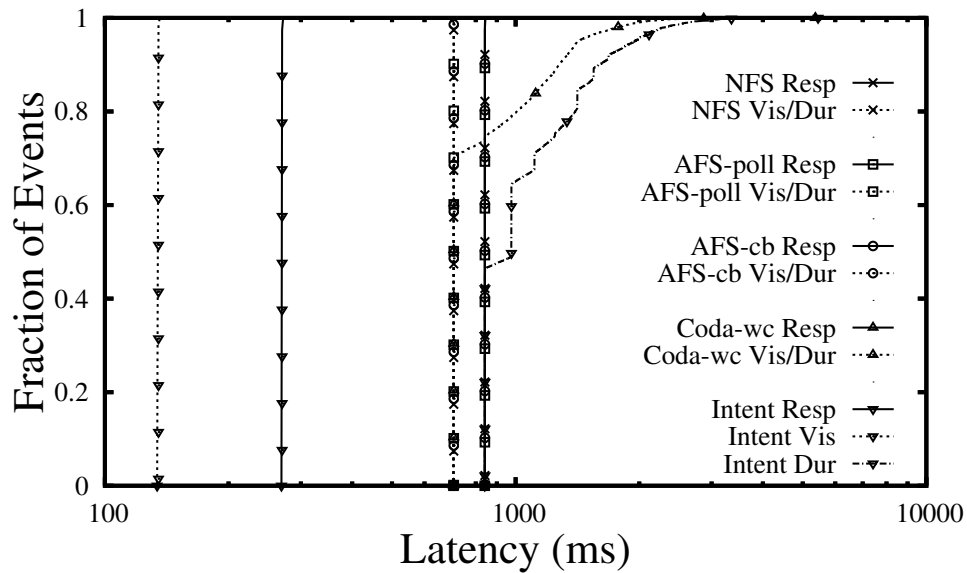
Figure 5.12: **The Bursty Workload:** The bursty workload is a synthetic workload that can be useful for studying the response of storage systems in overload. In this workload, clients alternate between idle periods and periods of activity. Parameters control the intensity of the bursty periods and characteristics of the file set.

rely on telephone modems. This is a relatively light workload that does not saturate the modem connections.

Figure 5.13(a) shows the cumulative distribution function (CDF) of response time, visibility latency, and durability latency of store operations for the various protocols in the test system. Because the files are of uniform size, most latencies are constant, as shown by the vertical lines. The curves cross the x-axis at points predicted by the computations of Table 5.4. The curves corresponding to Coda-wc's visibility/durability latency and intention update's durability latency exhibit a tail with some operations taking longer than the computed minimum. These latencies are all dependent upon the delivery of asynchronous messages. During bursts of activity, these asynchronous messages can queue up behind one another at the client. The tail becomes more pronounced as the intensity of the bursty period increases.

Although the latency of individual operations is regular and predictable, the cumulative impact of the latency leads to more stochastic behavior. Figure 5.13(b) shows the CDF of the response time penalty, visibility latency penalty, and durability latency penalty of store operations, comparing the behavior of the test and ideal systems. Recall that the penalty measurements are cumulative; one slow event can increase the penalty of later operations. The backlog may persist indefinitely, but idle periods provide an opportunity for the client to empty its queue of waiting events, "catching-up" to the reference system. First, we note that many of the curves contain vertical segments intersecting the x-axis in the same place as in Figure 5.13(a). This indicates that most operations are able to complete before subsequent operations arrive and reinforces that claim that

### CDF of Response, Visibility, and Durability Latency



### CDF of Response, Visibility, and Durability Penalty

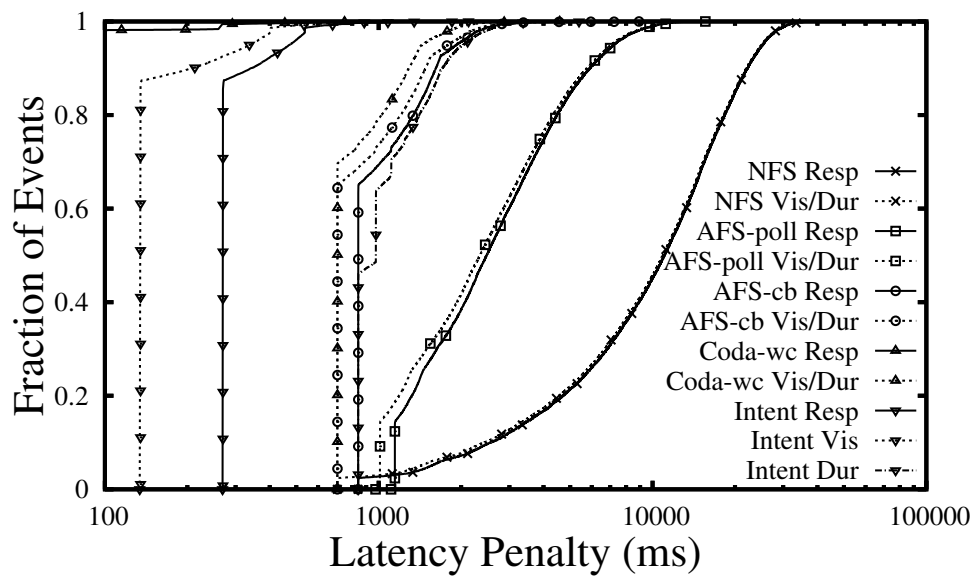


Figure 5.13: **CDF of Store Operation Performance with Uniform File Size:** The synthetic bursty workload is created with 10-minute cycle time, 30-second bursty period, and 10-second average interarrival time. Each client has a file set of 1000 4KB files, and 20% of accesses are writes. The reference system uses ideal links; the test system uses telephone modems.

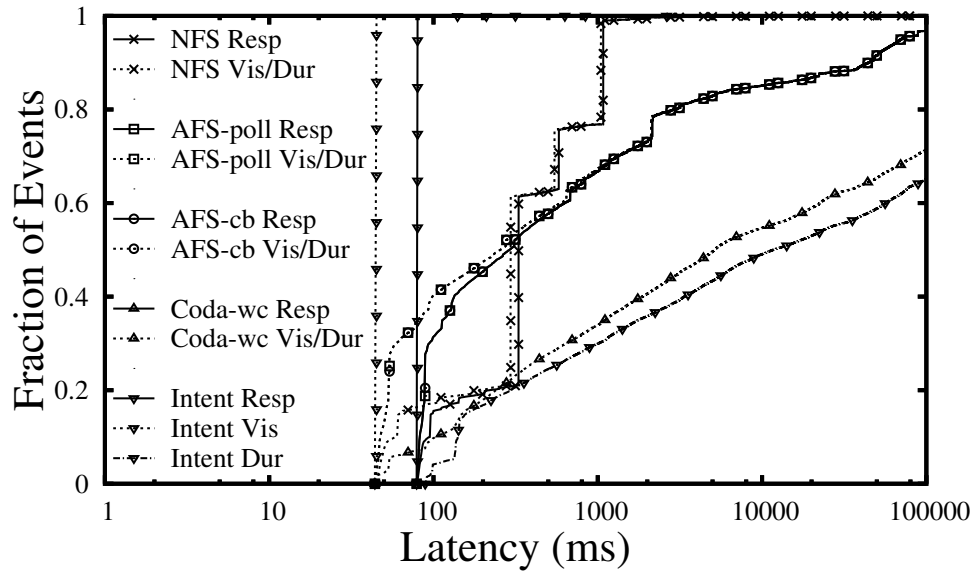
this is a relatively light workload. The x-intercepts of the AFS-cb curves are shifted slightly because AFS-cb must poll the server on every `open()` operation. Next, we observe that all of the curves do exhibit a tail. These features show that, even with light workloads like this one, queueing due to slow networks can significantly increase the amount of time it takes the user to complete a sequence of operations. The response time penalty of Coda-wc is negligible; a few operations are delayed to fetch uncached data. The Intent model is the next most responsive; this model avoids delays by requiring the client to send only small messages in the critical path of the operation. For the same reason, the Intent model provides the minimum visibility latency penalty (with little impact on the response time penalty). The durability of all models that use callbacks to maintain consistency is roughly comparable. The high cost of accessing data dramatically affects the latency penalty of the NFS and AFS-poll protocols. Some requests are serviced 10-40 seconds later than they would be on a well-connected client. Finally, though the Intent model must send more total data across the network, the durability latency penalty is not significantly greater than that of other protocols.

#### 5.6.4 Simulating Traced Workloads

Understanding the basic features that the simulations reveal, we now model the response of different protocols to real-world workloads. Specifically, we drive the simulator with the Roselli traces [RLA00]. We use the workload recorded in the March, 1997 segment of the “research” trace. This portion of the trace contains activity from 23 unique users. The “reference” subsystem still models ideal network links; clients in the “test” system, however, now use faster networks modeled on cable modem technology.

Figure 5.14(a) shows the CDFs for response time, visibility latency, and durability latency for the different client protocols. Each curve combines the results of all users in the trace. The graph shows many of the same trends as Figure 5.13(a). Each curve intersects the x-axis at the point predicted by Table 5.4. Many of the curves share a common point of intersection because the trace contains some accesses to files that are empty or of very small size. As predicted by the model, when  $M_t = M_s$ , many of the protocols behave in a congruent manner. Because the size of the intent message is fixed, the response time and visibility latency of the Intent model are fixed. The latency for other models increases as the size of the file, and consequently the size of the message increase. For AFS-poll and AFS-cb, more than 10% of store operations take more than 10 seconds to complete. The durability latency of the Coda-wc and Intent models are comparable with some writes taking more than several minutes to reach the server. The performance of the NFS

### CDF of Response, Visibility, and Durability Latency



### CDF of Response, Visibility, and Durability Penalty

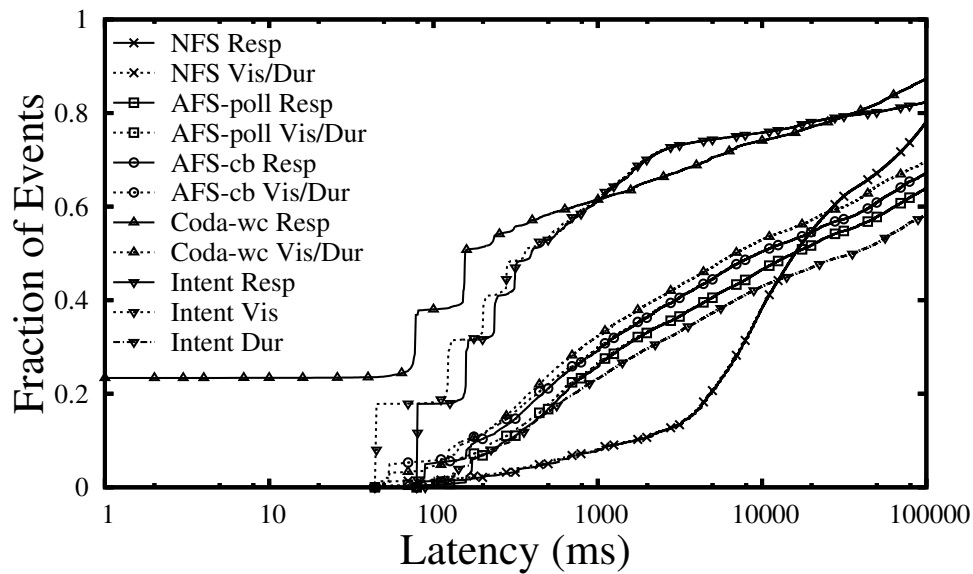


Figure 5.14: **CDF of Store Operation Performance with Workload from Roselli Traces:** The plots show the response of different protocols to the workload recorded in the March, 1997 segment of the Roselli research trace. The reference system uses ideal links; the test system uses cable modems.

model appears to be competitive with the AFS-based models, but recall that NFS issues a `write()` operation for each block written by the application. Although individual operations are relatively fast, this approach has a pernicious effect on overall performance as we will see below.

Figure 5.14(b) shows the CDF of the response time penalty, visibility latency penalty, and durability latency penalty of store operations, comparing the behavior of the test and ideal systems. Again, recall that the penalty measurements are cumulative. The Coda-wc and Intent models provide the best response time. By using small intent messages to make changes visible at the server, the Intent model has a low visibility latency penalty too. The Intent model is able to provide this improved service with negligible impact on durability latency penalty—the time to durability is similar for all AFS-based protocols. Because it contacts the server on every write operation, the NFS models lag in performance for most operations.

## 5.7 Simulating Variations

We now evaluate two of the key variations described in Section 5.3.

**Asynchronous intents:** In that section, we argued that some applications might benefit from the improved response time of asynchronous intents. Figure 5.15 shows the potential performance improvements. In this experiment, we use the workload generated from the Roselli traces assuming clients rely on cable modem connections. We simulate both synchronous intents and asynchronous intents. We measure the response time penalty, visibility latency penalty, and durability penalty compared to clients working over ideal links.

With asynchronous intents, about 25% of all store operations have no response time penalty—the user receives a response at the same time in both the ideal and test systems. For the remaining operations, the response time penalty is non-zero but is less than the response time penalty of synchronous updates by roughly the round-trip latency of the link. The response time penalty is non-zero because of delay accumulated from prior operations. However, when the operations are finally issued, they complete with no delay.

As we would expect, the visibility latency penalty and durability latency penalty are roughly equivalent whether intents are performed synchronously or asynchronously. The window of vulnerability when the client believes the data to be visible and when it actually becomes visible is less than 100 ms.



## CDF of Response, Visibility, and Durability Penalty

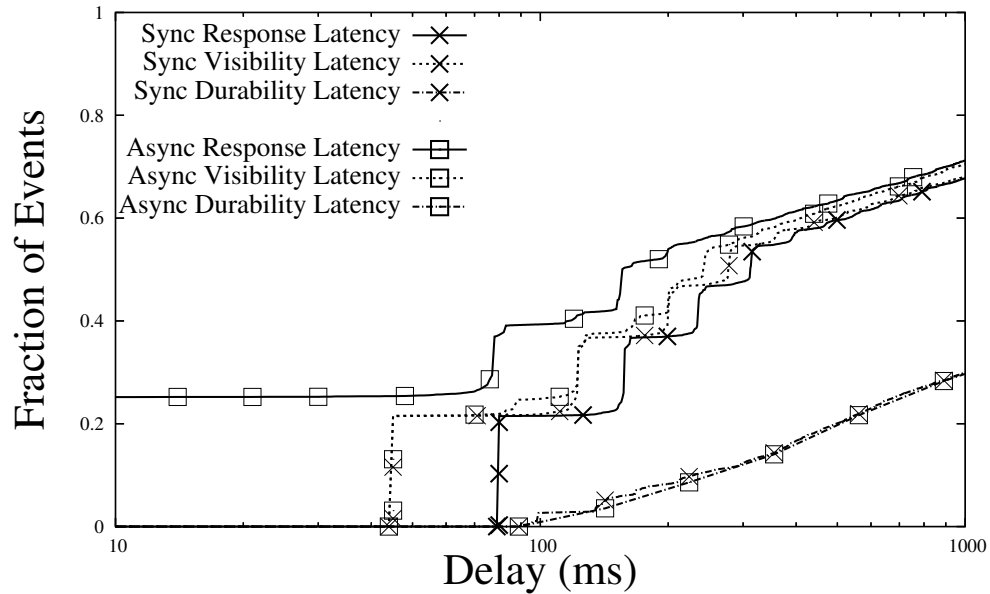


Figure 5.15: **Performance of Asynchronous Intents:** Simulating clients from the Roselli traces accessing remote storage over cable modems, we compare the behavior of synchronous and asynchronous intents. Asynchronous intents reduce the client-perceived response time. The visibility latency penalty and durability latency penalty are roughly equivalent.

**Re-ordering data transfers:** Also in Section 5.7, we explained how intention updates enable clients to alter the order that data was transmitted to the server without risk of inconsistent access. A client might re-order data transfer to provide better durability for certain types of data or to share data with other users more quickly.

Figure 5.16 shows how re-ordering data transfers can improve performance for clients sharing data. In this experiment, a user is accessing data from two devices simultaneously. The user initiates most accesses from the primary device, but, occasionally, after writing data with the primary device, the user accesses the data on the second device (for example, to check cross-platform compatibility). In the base case, data is propagated from the primary device in the order that it is created, and the server blocks the accesses of the secondary device until data is available. In the optimized case, when the secondary device requests data that is not yet available at the server, the server sends a hint to the primary device requesting that it increase the priority of the transfer of the requested data. We record the read latency distribution of accesses from the secondary device. The results show that re-ordering data transfer messages can dramatically reduce the latency to prop-

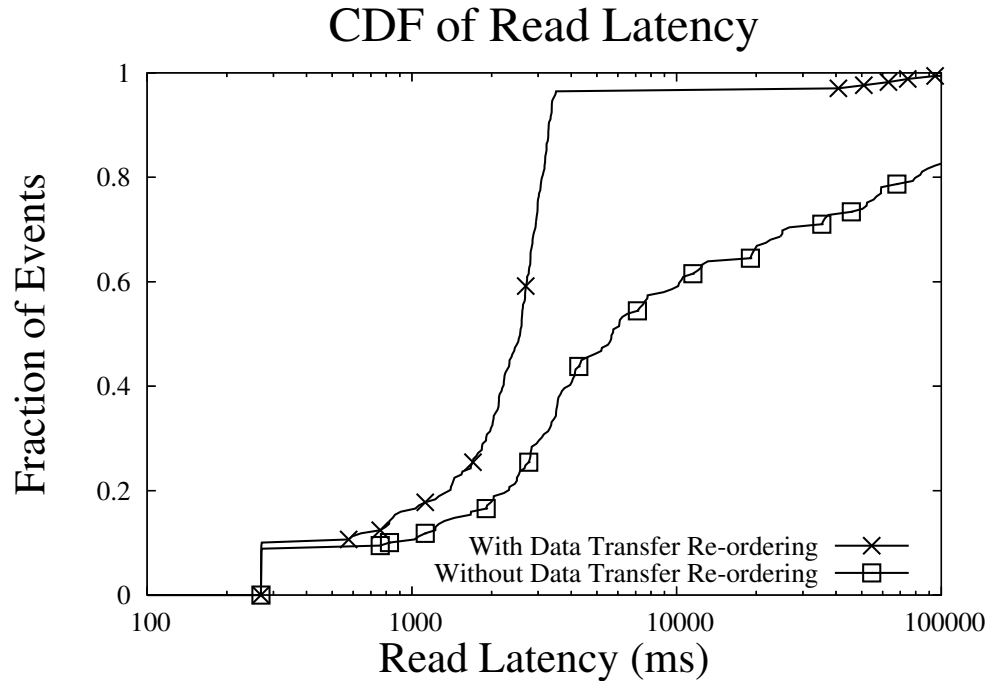


Figure 5.16: **Re-ordering Data Transfers:** The user accesses a file set of 1000 files with size distributed as in the “large” workload. We simulate a bursty workload with 10-minute cycle time, 30-second bursty period, and 5-second average interarrival time. 20% of access are writes. The secondary device accesses every 100th file touched by the primary device. Both devices use 56 kb/s network connections. The graph plots the latency of accesses from the secondary device.

agate critical data to the server by ensuring that it is not queued behind other, less critical data. The long-tail of the optimized case is due to large files—modifications to large files can be slow to propagate, even when granted priority.

## 5.8 Discussion

In this study, we have explored how decoupling the update of metadata and user data can improve the client-perceived responsiveness of a remote storage system.

We have seen through simulation of file system traces that bursts of activity are common in real workloads. Further, we have seen that waiting for communication across a weak network during bursty periods can delay requests, triggering the formation of queues of requests at the server and increasing the time that it takes to service requests. As the queue length grows and the average queue waiting time increases, client performance drops. To users, it appears that the system is

sluggish; we have seen that it can take many minutes longer to complete some sequences of bursty operations on a weak link than on a high-speed link.

The simulations also validated, however, the contention that we can create new protocols to improve the responsiveness of remote storage systems. Specifically, we explored how a protocol that emphasizes update visibility and response time can reduce the amount of time users spend waiting for requests to be serviced. In fact, simulations reveal that such protocols can reduce the time needed to complete a session by as much as hundreds of seconds. These results corroborated the results from our queueing theory analysis in Sections 5.5. The simulation reinforced the intuition that small, high-priority messages can improve client-perceived performance by transmitting messages from the client to the server quickly and keeping the state stored at the server current. We were especially encouraged to see that intention updates could achieve these benefits at such a small cost to data durability.

Finally, in simulating some variations of intention updates, we observe that the ability to re-order data transfer messages could be a big benefit to weakly connected clients. With the guarantee that the server can protect clients from accessing stale data, writers can re-order transfers to push critical data to the server first. This feature can improve the durability of critical data and facilitate sharing among weakly connected clients. We also note that the effect of asynchronous intention updates is insignificant when using broadband networking technologies.

## Chapter 6

# Extent-Based Content-Addressable Storage

In Section 1.4, we assumed that the interaction between the middleware file server and the storage infrastructure had no impact on the client-perceived performance of the system. Of course, this assumption is not trivially true. If a large network distance separates the middleware file servers from a centralized storage provider or some members of a distributed storage infrastructure, then interaction between the file servers and the storage infrastructure will impose additional latency on requests.

In our experiences with large, distributed, content-addressable storage systems, like the one that motivated our work, we observed that the latency of interacting with the storage infrastructure is even greater than the network latency would suggest. Attempting to reduce the latency of reading and writing data to the system, we developed a new API for interacting with content-addressable storage systems. In this chapter, we describe the problems that plague traditional content-addressable systems and present the API that helped us overcome these problems. We describe a wide variety of benefits that result from the adoption of this interface.

### 6.1 Background

Self-verifying data and the content-addressable interface have proven to be a solid foundation on which to build large, distributed storage infrastructures. We begin by reviewing the concept of self-verifying data and describing how designers exploit its properties to build distributed, scalable storage systems.

### 6.1.1 Self-Verifying Data

Data is said to be *self-verifying* if it is named in a way that allows any client to validate the integrity of data against its name. Traditionally, data is made self-verifying via one of two techniques: hashing and embedded signatures. These techniques were made popular by the Self-certifying Read-only File System [FKM00]. *Hash-verified data* is named by a secure hash of its content. A client can verify hash-verified data by computing the hash of the returned data and comparing it to the name used to fetch the data. Hash-verified data is immutable—if the data changes, the hash-verified name of the data changes too.

*Key-verified data* is verified by a certificate that is signed by a user's public key. The certificate contains some token, such as a secure hash of the content, that securely describes the data. To verify key-verified data, a client checks the signature on the certificate and compares the data against the verifier in the certificate. Commonly, key-verified data is named by a hash of the public key that signs the data's certificate. With this approach, each key pair can be associated with only a single object. To allow the system to associate multiple objects with a single key pair, other schemes hash a combination of data, such as the public key and a human-readable name, to create the name for the data. Key-verified data can be mutable—a client can associate new data with a key by creating a new certificate.

Many systems employ Merkle's chaining technique [Mer88] with hash-verified data to combine blocks into larger, self-verifying data structures. Such systems embed self-verifying names into other data blocks as secure, unforgeable pointers. To bootstrap the process, systems often store the name of the root of the data structure in a key-verified block, providing an immutable name for mutable data. To update data, a client replaces the key-verified block. See, for example, CFS [DKK<sup>+</sup>01], Ivy [MMGC02], and Venti [QD02].

### 6.1.2 Content-Addressable Storage Systems

In content-addressable storage (CAS) systems, data is addressed not by its physical location but by a name that is derived from the content of that data. CAS systems provide an interface similar to the `put()`/`get()` interface of the traditional hashtable.

Figure 6.1 illustrates the primary components of a distributed content-addressable storage system. Clients issue read and write requests to the system. The storage servers provide the actual storage for blocks stored in the system. The query router routes client requests to the storage server that can fulfill the request. The distributed hash table (DHT) is a common implementation of

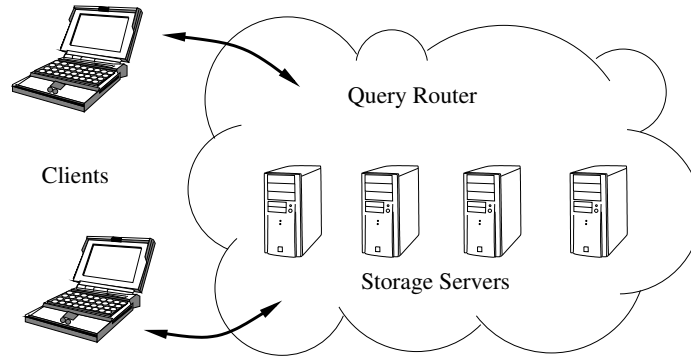


Figure 6.1: **Components of a Content-Addressable Storage System:** A distributed content-addressable storage system contains three primary components. Clients issue read and write requests to the system. The storage servers are responsible for actually storing the data. The query router routes client requests to the storage server that can fulfill the request.

**Traditional interface:**

---

```

put_hash(H(data), data);
put_key(H(PK), data);
data = get(h);

```

---

Table 6.1: **The CAS System put () /get () Interface:** First-generation distributed CAS systems use a simple put () /get () interface. The put\_hash () and put\_key () functions are often combined into a single put () function.  $H()$  is a secure, one-way hash function;  $h$  is a secure hash, as output from  $H()$ .

the query router.

In traditional CAS systems, all components operate at the granularity of a block using an interface like the one shown in Table 6.1. Using this interface, the client issues a separate request for every block access; storage servers distribute data at the block granularity; and the query router indexes each block individually. Although we have shown put\_hash () and put\_key () as distinct members of the interface, they are often implemented as a single put () function.

Not only do many system use a common interface, but they also to use self-verifying data and the put () /get () interface in a common manner, illustrated in Figure 6.2. A client divides data into small blocks, typically 4–8 KB or less. It computes the hash-verifiable name of each block and links the blocks together, using the names as unforgeable references, to create a Merkle tree. Finally, the client stores all blocks of the tree in the CAS system using the put\_hash () interface. If the system supports mutable data, the client will typically use the put\_key () function to store a key-verified block that points to the root of the Merkle tree, providing an immutable name to the mutable data.

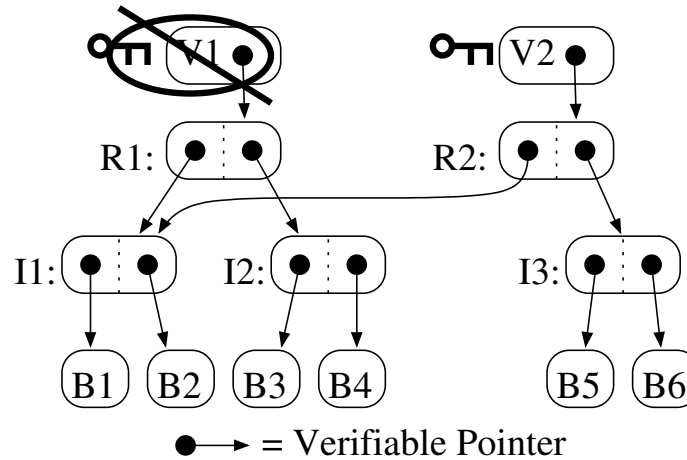


Figure 6.2: **Use of Self-Verifying Data in CAS Systems:** Clients divide data into small blocks that are combined into Merkle trees. A key-verified block points to the root of the structure. To update an object, a client overwrites the key-verified block to point to the new root. ( $V$  = version,  $R$  = version root,  $I$  = indirect node,  $B$  = data block)

To read data, a client first retrieves and validates the key-verified root block of the data structure using the `get()` function. It can then iteratively fetch and verify the other hash-verified blocks by following the chain of hash-verified names embedded in the tree.

Because each new hash-verified block of data has a unique name, CAS systems naturally provide versioning capabilities. Some systems expose the versioning feature to the end user [REG<sup>+</sup>03] while others do not. Using copy-on-write to provide efficient versioning has also been implemented in other systems predating the distributed CAS systems that we describe [MT85].

Distributed storage systems derive several favorable properties from the use of the CAS interface. First, a CAS interface helps ensure data integrity. If a system carefully selects the method used to derive names from data, clients can validate the integrity of data retrieved from the system against the name by which it was accessed. With *self-verifying* data, clients can detect data altered by faulty or compromised components and re-fetch from alternate sources. Also, a CAS interface promotes system scalability. Because the interface does not expose physical addresses to applications, the system can replicate and transfer data freely to add hardware resources or upgrade internal protocols.

Self-verifying data is naturally well-suited to work with content-address storage systems, with the self-verifying name serving ideally as a block's identifier in the system. The self-verifying property enables clients to request data from any machine in the network without concern of data

corruption or substitution attack. A malicious or compromised machine cannot deceive a client with corrupt data—its attack is limited to denying a block’s existence.

A number of recent distributed storage systems used self-verifying data and the content-addressable interface as their foundation. Despite their independent development, many systems share important design features. In identifying common design features, we have considered a number of popular, first-generation content-addressable storage systems in the research literature including CFS [DKK<sup>+</sup>01], Ivy [MMGC02], OceanStore [REG<sup>+</sup>03], Total Recall [BTC<sup>+</sup>04], and Venti [QD02].

One notable counterexample to these design patterns is the PAST [RD01] system. PAST uses the `put_hash()` function to store whole objects as hash-verified blocks. As a result, PAST cannot incrementally update objects; it can only completely overwrite old versions.

### 6.1.3 Consequences of the CAS Interface

The common design features have a significant impact on the behavior of the resulting CAS systems. For example, the `put()`/`get()` interface forces the storage infrastructure to manage data at the same granularity as the client. Although some applications, like off-line data processing, handle data in large chunks, many interactive and user-oriented applications tend to create and access relatively small blocks of data. By supporting fine granularity access, these systems allow applications to fetch data without wasting scarce bandwidth at the edges of the network retrieving data that is not needed or already cached. It allows applications to push data to the infrastructure as soon as it is created, improving durability.

Coupling the infrastructure’s unit of management with the client’s unit of access, however, has several disadvantages. For example, because each block is managed independently in the infrastructure, the storage system cannot identify locality. In fact, because most storage systems use a block’s name to determine its placements, and because block names are effectively random bit strings, the blocks comprising a single file or object are usually distributed among many servers in the storage system. This has a detrimental impact on efficiency. To retrieve a file or object, the client must flood the system with separate requests for each block and retrieve blocks from a large number of servers.

Managing blocks individually also has a negative impact on the efficiency of indexing, management, and maintenance. Some costs associated with these tasks are independent of the block size. Thus, managing the small blocks created at the client increases the load on the infrastructure.



Finally, the use of the block-based `put()`/`get()` interface dramatically increases the cost of determining the owner of data stored in the system. Identifying the owner of a piece of data is critical for any system that wishes to monitor per-user storage consumption or compute usage-based fees. The approach most commonly proposed for implementing such functionality is to include a certificate with each block of data. The certificate provides a secure, non-repudiable binding between the data and its owner. The certificate remains co-located with the data, even as the data is replicated or transferred. Although this solution is conceptually simple, the runtime cost of the scheme is steep because the client must create and sign a certificate for each block it stores in the system. In fact, some designers have deemed the feature infeasible because the cost of producing certificates is prohibitively expensive [FKM00]. To illustrate this problem, assume an application running on a 3 GHz processor wishes to store 1 TB of data. Dividing the data into 8 KB blocks and using 1024-bit RSA cryptography, it would take more than *six days* to create certificates for the data<sup>1</sup>.

## 6.2 Improving CAS with Aggregation

We observed that many of the problems associated with current CAS systems stem from the adoption of a block-based interface for storage, retrieval, indexing, management, and maintenance. We now look at how to apply the lessons of previous storage systems research that used aggregation to improve system efficiency to our domain.

The classical filesystems literature demonstrates repeatedly how aggregation can improve the efficiency of storage systems. For example, the Fast File System (FFS) [MJLF84] increases system performance, in part, by aggregating disk sectors into larger blocks for more efficient transfer to and from disk. XFS [SDH<sup>+</sup>96] further aggregates data into extents, or sequences of blocks, to reduce the size of the metadata and allow for fast sequential access to data. GoogleFS [GGL03] aggregates data from a single file further still into 64 MB chunks, improving performance and per-object maintenance costs for large files typical of their application domain.

More recently, the Glacier [HMD05] distributed CAS system, has shown how aggregation can reduce the number of objects that the system must index and manage. Glacier [HMD05] relies on a proxy trusted by the user to aggregate application-level objects into larger collections. All collections in Glacier are hash-verified and thus cannot be modified after they are created.

---

<sup>1</sup>A 3 GHz Pentium-class processor can create a signature in 4 ms, as measured with the command `openssl speed rsa1024`.

Although Glacier pioneered the use of aggregation in a distributed CAS systems, their solution does not directly apply to our target environment. We list below the goals that guide our design for using aggregation in content-addressable storage systems.

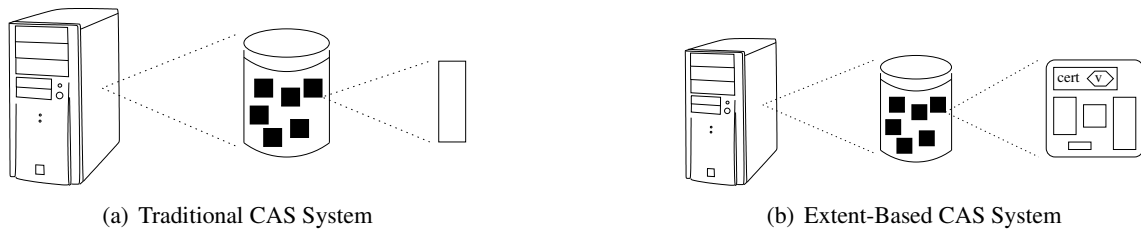
- **Self-verifiability:** To protect data integrity, all data must be self-verifying at both the fine granularity of the block and the coarse granularity of the aggregate. Self-verifiability allows clients to verify data locally without rely on secure servers.
- **Incremental update:** For data durability, a client must be able to write data to the system as it is created, without local buffering. Thus, the system must allow clients to add blocks to aggregates that are already stored in the system.
- **Fine-granularity access:** To conserve bandwidth at the edges of the network, the system must allow clients to access individual blocks without retrieving the entire aggregate.
- **Low infrastructural overhead:** The system should not require the infrastructure to index and maintain data at the block level. The design should allow the infrastructure to amortize the cost of maintaining data and verifying certificates over the larger aggregates.
- **Identification of data owners:** The system must be able to identify the owner of any piece of stored data in a secure and non-repudiable fashion without reference to other blocks. This allows individual nodes in the system to monitor consumption and compute usage fees on a per-user basis locally, without contacting other machines in the system.

## 6.3 Extent-Based Content-Addressable Storage

To exploit the benefits of aggregation, we developed an API which raises the granularity of management in content-addressable storage systems from the application-level block to container objects called extents. In this section, we describe this interface, first at a high-level and then in a detailed manner that includes the actual interface.

### 6.3.1 Overview of Extent-Based CAS

In an extent-based content-addressable storage system, storage servers stores container objects, called *extents*, instead of individual blocks. Figure 6.3 illustrates the different contents of a storage server in a traditional and extent-based CAS system. Each extent is filled with a set of



**Figure 6.3: Comparison of Storage Server Contents in Traditional and Extent-Based CAS Systems:** In traditional CAS systems, storage servers store and manage individual application-level blocks. In an extent-based CAS system, storage servers store extents, which contain a collection of application-level blocks and a certificate that identifies the owner of the data.

variable-sized application-level blocks. The contents of an extent are always collocated on a single storage server. Extents are either mutable key-verified objects or immutable hash-verified objects. All data in an extent is owned by a single principal. Associated with each extent is a certificate signed by a client that includes the client's identify. The certificate serves our goal of allowing components to identify the owner of the data. The certificate also contains a token that represents the current contents of the extent. The token is a cryptographically secure *verifier* (e.g. secure hash) that summarizes the contents of the extent.

The system supports incremental update by allowing a client to add data to a key-verified extent. The extent records the component blocks as a set ordered by the time they were written. Adding blocks to an extent, then, can be understood as appending blocks to the extent. The client submits a write request to the storage infrastructure that includes the data to be written and a new certificate. The certificate contains a new verifier, updated to reflect the data to be added.

To allow different components in the system to manage data at different granularities, the system supports *two-level naming*. In two-level naming, each block is identified not by a single name, but by a tuple. The first element of the tuple identifies the enclosing extent; the second element names the block with the extent. Retrieving data from the system is a two-step process, as illustrated in Figure 6.4. The system first locates the enclosing extent; then, it extracts individual application-level blocks from the extent. Two-level naming reduces the management overhead incurred by the infrastructure by decoupling the infrastructure's unit of management from the client's unit of access. The infrastructure needs only to track data at the extent level, and the client can still address individual blocks at random.

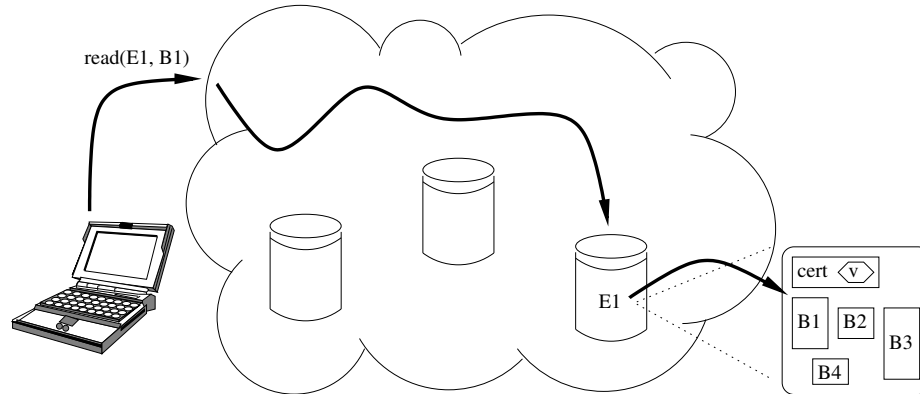


Figure 6.4: **The Two-Level Look-up Process:** In an extent-based CAS system, reading data requires a two-level look-up process. Here a client requests to read block  $B1$  from extent  $E1$ . First, the query router routes the request to a storage server that stores extent  $E1$ . Then, the storage server extracts block  $B1$  from the extent and returns it to the client.

### 6.3.2 An API for Extent-Based CAS

Before developing an interface for an extent-based CAS, we must first define how blocks and extents are named and verified. We will present our method for naming here without completely explaining the reasons why this approach was chosen. As we subsequently present the interface, we will highlight how this naming method supports and interacts with the API.

Blocks are named by a secure, one-way hash function, like traditional hash-verified data. The key advantage of this approach is simplicity. Clients can compute and verify block names easily without considering how blocks are aggregated into extents.

The approach used to name extents differs depending on whether the extent is key-verified or hash-verified. Key-verified extents, like traditional key-verified data, are named by a hash of the public key that verifies the certificate embedded in the extent. The name of a hash-verified extent, like traditional hash-verified data, depends on the contents of the extent. We do not, however, merely hash the contents of an extent.

A hash-verified extent is named by its verifier. To compute an extent's verifier, we use a chaining method [LKMS04] as shown in Figure 6.5. Assume an extent containing a sequence of data blocks,  $D_i$ , with names  $H(D_i)$ . The verifier is computed using the recurrence relation  $N_i = H(N_{i-1} + H(D_i))$ , where  $+$  is the concatenation operator. We bootstrap the process by defining  $N_{-1}$  to be a hash of the public key that signs the extent's certificate. By defining  $N_{-1}$  in this way, we ensure that the names of extents signed by different principals do not conflict. We call the

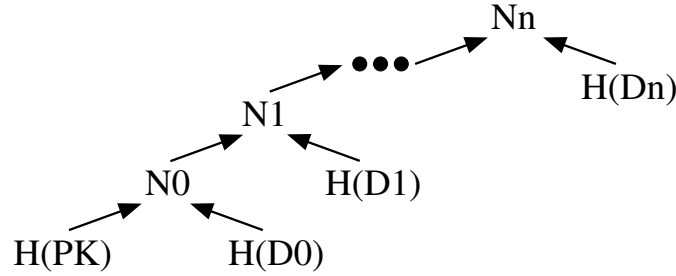


Figure 6.5: **Procedure for Computing Extent Verifiers:** To compute the verifier for an extent, the system uses the recurrence relation  $N_i = H(N_{i-1} + H(D_i))$ .  $N_{-1} = H(PK)$  where  $PK$  is a public key.

**Two-Level interface:**

---



---

```

status  = create(H(PK), cert);
status  = append(H(PK), cert, data[ ]);
status  = snapshot(H(PK), cert);
status  = truncate(H(PK), cert);
status  = put(cert, data[ ]);
cert    = get_cert(ext_name);
data[]  = get_blocks(ext_name, block_name[ ]);
extent  = get_extent(ext_name);
  
```

---

Table 6.2: **The Extent-Based CAS System Interface:** By extending the traditional `put()`/`get()` interface, CAS systems can support extents and two-level naming.

value produced by the recurrence relation a verifier because, in addition to naming the hash-verified extent, we also use that value to verify the contents of a key-verified extent. That is, to create or update a key-verified extent, the client must include in the certificate that will accompany the extent a verifier to validate the content of the extent.

Using chaining to create verifiers has several advantages. It allows the system to compute the verifier incrementally; when a block is added to an extent, the system must hash only the new data, not all data in the extent, to compute the running verifier. Also, chaining creates a verifiable, time-ordered log recording data modifications. Finally, we will see below that this approach simplifies the conversion of key-verified extents to hash-verified extents.

With these definitions for block names and extent names, we can now define an interface for extent-based CAS systems, shown in Table 6.2. All operations that modify data stored in the system require the client to provide a certificate authenticating the change; Table 6.3 lists the contents of a certificate. To support the extent-based model, the interface extends the traditional `put()`/`get()` interface, shown in Table 6.1, defining additional operations for key-verified data.

**Certificate contents:**

verifier	token that verifies extent contents
num_blocks	the number of blocks in the container
size	the size of data stored in the container
timestamp	creation time of certificate
ttl	time the certificate remains valid

Table 6.3: **Contents of an Extent Certificate:** The certificate stored with each extent includes fields to verify the contents of the contains and to identify the owner of the data.

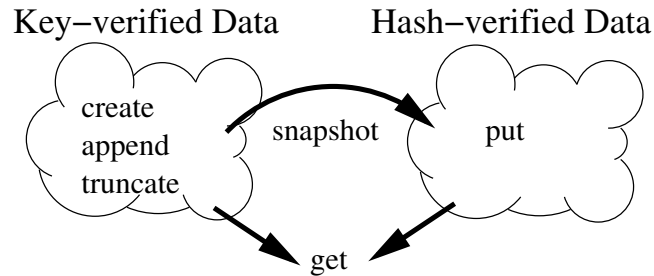


Figure 6.6: **A Comparison of the Traditional and Extent-Based CAS Interface:** The expanded interface of Table 6.2 provides different commands to operate on different types of self-verifying data. Commands `create()`, `append()`, and `truncate()` operate on key-verified extents while `snapshot()` converts a key-verified extent into a hash-verified extent.

The new operations allow the system to support extents and two-level naming. Figure 6.6 shows how the operations relate to the different extent types. The `create()`, `append()`, and `truncate()` operations allow clients to manage key-verified extents. The `create()` operation initializes a new, empty key-verified extent for the client; `append()` allows the client to add data to an existing key-verified extent; and `truncate()` deletes the contents of a key-verified extent. The first argument in each request type identifies the key-verified extent to be modified. The second argument is a certificate provided by the client that contains the verifier that securely describes the new contents of the extent. To compute a valid verifier, the client must know the contents of the extent prior to the requested change.

Another extension in the interface is the `snapshot()` operation. The `snapshot()` operation converts mutable key-verified extents to immutable hash-verified extents. Although the interface derives much of its power from the ability to append data to key-verified extents, it is not feasible to store all data in key-verified extents. Because the name of a key-verified extent is a function of the key that verifies its contents, doing so would require clients to manage a large number of key pairs (which is a difficult key management challenge) or extents to grow boundlessly large

(which is undesirable because it limits how the storage system can replicate, manage, and transfer data). Depending on the implementation, the system may copy the extent to a new set of servers during the `snapshot()` operation. Since the verifier is a cryptographically secure digest of the contents of an extent, it is a natural name for the new hash-verified extent. Since the client must sign a certificate that contains the verifier, it must know the eventual name for a hash-verified extent even before the `snapshot()` operation completes. We envision that a client-side library responsible for communicating with the storage system will call the `snapshot()` on behalf of the application when an extent reaches a specified maximum capacity.

The interface also provides three functions to access data stored in the system. The `get_blocks()` operation is the primary function for reading data, allowing applications to retrieve one or more application level blocks from the system. The `get_cert()` operation returns the certificate associated with an extent, allowing an application to determine the state of data stored in the system. Finally, the `get_extent()` operation supports bulk transfer and retrieval; it returns the entire container including all blocks and the certificate. A single set of read functions support the retrieval of data from key-verified and hash-verified extents—the query router does not distinguish between the different types of extents, and the storage servers execute the request the same for both types of extents.

### 6.3.3 An Example Use of the Extent-Based API

Consider the following example to illustrate a typical use of the proposed interface. Upon joining the system for the first time, a client uses the `create()` interface to create a new key-verified extent in the system. The storage system recruits one (or more, if an extent is replicated) storage servers to allocate space for the extent. (For the current discussion, assume the existence of a black box that can identify candidate servers. We call this the *set identification service*.) The set of servers allocate space for the extent and agree to participate in its management.

After an extent has been created, the client can add data using the `append()` operation. A client can append data to any key-verified extent for which it can create a certificate, signed with the proper key, that asserts those changes.

When the key-verified extent reaches a predetermined maximum size, the client (or more likely, a library working on behalf of the client) converts mutable extent to an immutable hash-verified extent using the `snapshot()` interface. Like `create()`, `snapshot()` must query the set identification service; however, the service may return a set containing some or all of the servers al-

ready hosting the key-verified extent. If the sets do intersect, the system may consume less network bandwidth in creating a hash-verified version of the extent.

After saving data in an immutable format, the client can reinitialize the key-verified extent with a `truncate()` operation. The `truncate()` operation removes all blocks from the extent, leaving a key-verified extent that is equivalent to a newly created extent. Although `snapshot()` and `truncate()` are typically used together, we have elected to make them separate operations for ease of implementation. Individually, each operation is idempotent, allowing clients to retry until successful execution is assured. In Section 6.5, we will show how the `snapshot()` and `truncate()` operations can be used to facilitate storing streams of data. Alternatively, an application could use `truncate()` without `snapshot()` to overwrite data stored in the system.

The `append()`, `snapshot()`, and `truncate()` operations that transform key-verified extents are useful for applications that periodically write small amounts of data, allowing the system to aggregate data in a way that was not possible previously. But in situations that applications quickly write large amounts of data, using the sequence of operations can be inefficient. Instead, applications may write collections of blocks directly to hash-verified extents using the `put()` operation. A client can have multiple outstanding `put()` operations for a single key pair. The `put()` operation also relies on the set identification service.

In addition to supporting incremental update and fine-granularity access, extents and two-level naming also allow distributed CAS systems to amortize some management costs. For example, two-level naming reduces the cost storing certificates by amortizing the storage overhead over a whole extent, allowing systems to devote a greater percentage of their resources to storing user data. Similarly, two-level naming reduces the query load on the system because clients need to query the infrastructure only once per extent, not once per block. Finally, assuming data locality—that clients tend to access multiple blocks from an extent—systems can exploit the use of connections to manage congestion in the network better.

Also, by using a single certificate to identify the owner of an entire extent, the extent-based API makes tracking the owner of data stored in the system feasible. Consider again an application running on a 3 GHz processor that wishes to store 1 TB of data with certificates produced using 1024-bit RSA cryptography. We saw that, dividing the data into 8 KB blocks, it would take more than *six days* to create certificates for the data. If instead, the application stored data in an extent-based system with 4 MB extents, it could create the necessary certificates in *17 minutes*, a reduction of three orders of magnitude over a system that implements the certificate-per-block approach.



## 6.4 A Secure, Append-Only Log Abstraction

Although the extent-based API meets the goals we defined in Section 6.2, it is not a familiar or convenient interface for application writers. To bring the power of the interface to application writers, we have layered a secure log API on top of the extent-based API. The log serves as a simple, familiar, powerful abstraction for interacting with extent-based content-addressable storage.

Within the log abstraction, a log is owned by a single principal, identified by a cryptographic key pair. Only the owner of the log can modify it. If, however, multiple devices share a private key, then they all can modify the log.

The log interface exports a very limited interface: `create()`, `append()`, `flush()`, `read()`, and `getHead()`. The log does not support a delete operation. The `create()` operation creates a new log. The `append()` function adds one or more application-level blocks to the head of the log. An application can ensure that all previously written data is stored durably by issuing `flush()`. The `read()` function reads one or more blocks from the log. Finally, the `getHead()` function returns the data block at the head of the log. For many applications, when activity to the log quiesces, the block at the log head is similar to the root directory of a file system. As such, the `getHead()` function is commonly used when “opening” or accessing a log for the first time. In the case that a client crashes leaving the log in an inconsistent state, it may be necessary to use the low-level extent-based API to scan back through the log seeking the last stable checkpoint to allow the application to recover. We do not consider recovery of the log further.

To service requests using the log API, a library at the client translates commands of the log API into commands from the extent-based API. The client library stores each log as an ordered sequence of extents, each containing multiple, variable-sized application-level blocks. The extent representing the head of the log is a key-verified extent. The client library appends blocks to the log by appending them to the key-verified extent. When the extent at the head of the log reaches a specified maximum capacity, the library snapshots the extent to convert it to an immutable, hash-verified extent. The library then truncates the key-verified extent, preparing it to store new data at the log head. As a consequence of the process, most data in the log is stored in hash-verified extents. To maintain the log, the library inserts a small block of metadata as the first block into each extent. This metadata block serves to link the extents into a single chain and aid in reading data from the log.

The biggest challenge to using extents to implement a secure log is in determining how to name blocks. When applications are serializing data structures to store in the log, they often embed

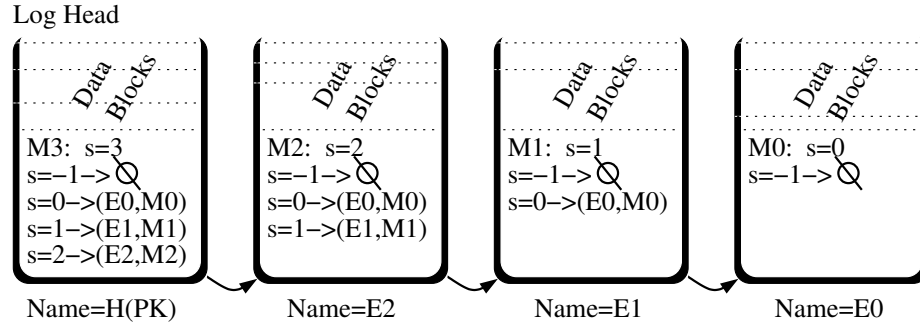


Figure 6.7: **Layering the Secure Log on Top of Extents:** Shown here is a log that extends across four extents. The head of the log is at the left; the oldest extent of the log is on the right. The first (bottom) block in each extent is a metadata block that maintains the state of the log. The metadata block (named  $M0$ ,  $M1$ , etc.) contains the extent's sequence number ( $s = 0$ ,  $s = 1$ , etc.) in the log, and a set of mappings to previous extents and metadata blocks in the log.

the names of blocks into the data structure. Recall that, using the extent-based API and two-level naming, a block is addressed by a tuple that includes the extent name and the block name. When, however, the extent at the head of the log is converted to hash-verified form, the name of an extent changes. We cannot know the eventual hash-verified name of an extent as long as it remains in key-verified form. To embed data structures in the log, applications need to know the name of a block after it is appended to the log so that they can refer to those blocks. So, without knowing the eventual hash-verified name of an extent, how do we refer to a block when serializing data structures?

To create unique, unforgeable references for blocks stored in key-verified extents, the client library assigns a sequence number,  $s$ , to each extent in the log chain. The first extent in a new log is initialized with  $s = 0$ . After the extent at the head of the log is filled and the client library issues the `snapshot()` and `truncate()` operations to prepare the log to accept more data, the library clones the metadata block of the previous extent in the chain, increments the sequence number, and inserts the block in the otherwise empty key-verified extent at the head of the log. For the log interface, blocks are named by a tuple of the form  $(s, H(D_i))$  where  $H(D_i)$  is the hash of the block. When applications store blocks in the log, the client library returns the blocks' new names. Those names can be embedded in the application-level data structures.

The block names used in the log interface cannot, however, be used to retrieve data directly because the extent-based storage system does not identify extents by log-defined sequence numbers. To convert the log interface's block names to the form used by the extent-based API, the client

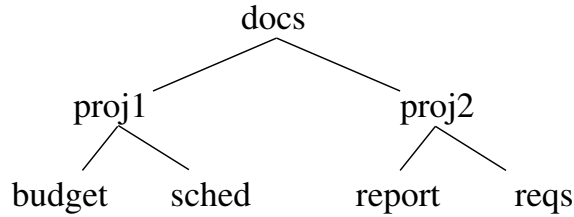


Figure 6.8: **An Example Directory Tree:** A simple file system used as a running example.

library maintains a mapping that resolves the sequence number to the permanent, hash-verified extent name. This mapping is placed along with the sequence number in the metadata block in each extent. Each time the mutable extent is made hash-verified and then truncated, the client library records the mapping  $s_{j-1} \rightarrow (E_{j-1}, M_{j-1})$  where  $E_{j-1}$  is the hash-verifiable extent name of the previous extent and  $M_{j-1}$  is the block name of the metadata block in the previous extent. Figure 6.7 illustrates how the client library uses extents to store the log, including the contents of the metadata block.

## 6.5 Example Application: Versioning Back-up

To illustrate the use of the secure log, the translation of the log API to the extent-based API, and the role of the log’s metadata blocks, we present a sample application. Specifically, we show how a versioning file system back-up application stores data into a secure log.

A file system back-up application must convert the tree-based directory structure of a file system into a form that can be stored in a log. In the simple design we present here, each directory is stored as a single, variable-sized block containing the directory entries and a verifiable reference to each child; each file is stored as a single, variable-sized block containing the file metadata and data. (A more complete implementation could use the same techniques described in this section to divide large directories and files into blocks.)

To archive a file system—for example, the simple file system shown in Figure 6.8—the application translates the file system into a self-verifying Merkle tree in a depth-first manner. The form of the resulting Merkle tree for the sample file system is shown in Figure 6.9(a). Into each directory, the application embeds a verifiable pointer that refers to each child. The verifiable pointers are the block names of the children in the log. The translation process is analogous to that used in CFS [DKK<sup>+</sup>01]. Note the similarity between Figure 6.9(a) and the first version of the object shown in Figure 6.2.

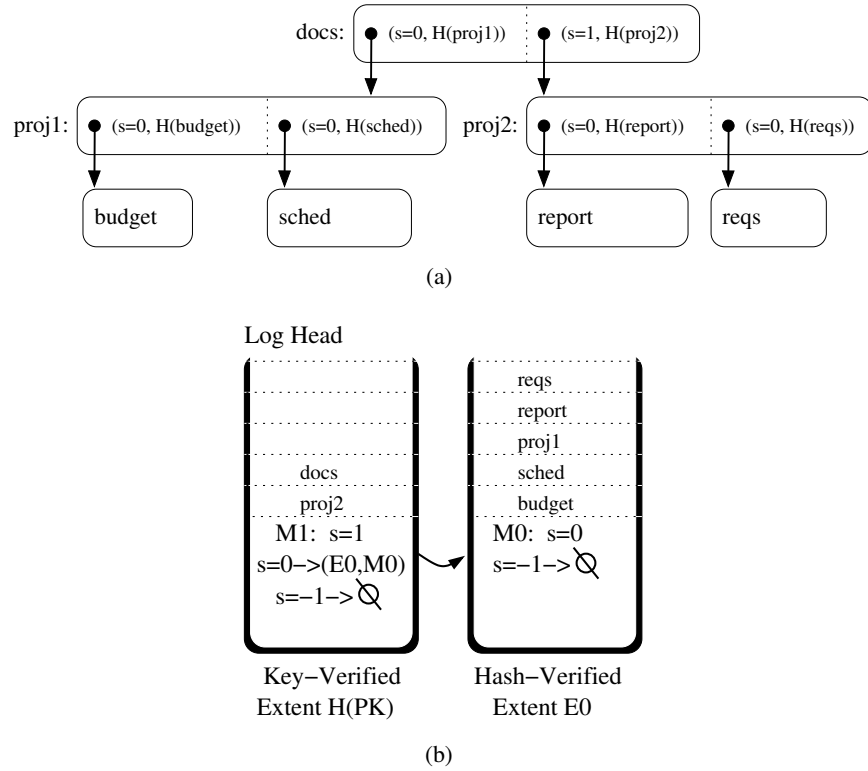


Figure 6.9: **Archiving the Initial Version of a File System:** (a) The back-up application translates the file system into a Merkle tree. The verifiable pointers are of the form  $(extent\_sequence\_number, block\_name)$ . Verifiable pointers refer to extent by sequence number because the permanent, hash-verifiable name is not known until later in the archiving process. (b) The Merkle tree is stored in two extents. The first extent,  $E_0$ , is filled and has been converted to a hash-verified extent. The second extent,  $H(PK)$ , is a partially filled key-verified extent. Notice that the first block in extent  $H(PK)$  contains metadata including a reference to the metadata block,  $M_0$  of the previous extent.

Figure 6.9(b) shows the contents of the chain of extents after archiving the first version of the file system. The first block in each extent contains the log metadata including the sequence number of the extent and the mappings between previous sequence numbers and the corresponding hash-verified names of their extent. In storing the initial version of the file system, the application completely filled one extent and partially filled another. The filled extent,  $E_0$ , has been converted to a hash-verified extent by the client library and is immutable. The partially filled extent  $H(PK)$  is a key-verified extent and can store more data at a later time. The first block in  $H(PK)$ , its metadata block, includes the mapping for the previous extent  $E_0$ .

Figure 6.10 shows how the application handles modifications to the file system. Assume

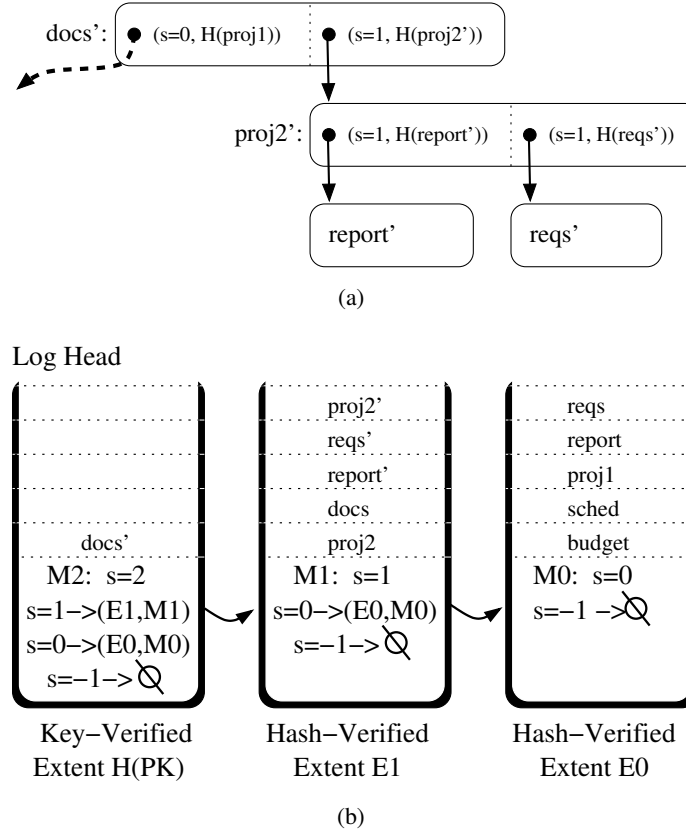


Figure 6.10: **Archiving an Update to the File System:** (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the extent chain after storing blocks of the updated file system.

the user edits the files in the `proj2` directory. Figure 6.10(a) shows the Merkle tree resulting from these changes. The dashed pointer indicates a reference to a block from the previous version, namely block  $(s = 0, H(\text{proj1}))$ . Figure 6.10(b) shows the contents of the extent chain after recording the changes. The application again filled the key-verified extent which has been converted to the immutable hash-verified extent  $E1$  by the client library. It then re-initialized the key-verified extent, filled it with a metadata block recording the hash-verified names of previous extents and the name of the metadata block in those extents, and appended the remainder of the file system data.

To recover the name of an extent corresponding to a sequence number, the client library must consult the mappings that are stored in the extent. The mapping can always be found as the first data block of the key-verified extent corresponding to the log. The client library can trade storage for lookup latency by storing more or fewer mappings in each extent. In our implementation of the log library, we maintain a logarithmic number of mappings in each extent which enables the client

library to resolve any extent name in a logarithmic number of steps. The library may also keep a local cache of these immutable mappings to accelerate future translations.

To read the file system stored in the log, the application will begin by requesting the data block at the head of the log. The client library will query the key-verified extent at the log head for a certificate to learn of the content of the extent; it will then request the last block to be added to the extent and return it to the file system application. The head block corresponds to the root directory of the file system and contains pointers to all of its children. To read one of the children, the application issues a `read()` request that includes the pointer embedded in the root directory. The pointer contains the sequence number of the extent containing the child and the block name of the child. The client library will resolve the extent sequence number to extent name by consulting the mappings stored in the metadata blocks of the log. It will then read the block from the extent directly.

## 6.6 Prototype Implementation and Evaluation

The API presented in this chapter serves as the foundation of the Antiquity distributed storage system. For a complete description of Antiquity, the reader should refer to other publications [WECK07]. In this section, we describe the high-level features of the Antiquity design that relate to the extent-based API and present some evaluation results from the system.

### 6.6.1 Antiquity: A Distributed, Extent-Based CAS System

The Antiquity prototype demonstrates how to implement the extent-based API and two-level naming interface efficiently in a distributed system. Antiquity uses a secure log to maintain data integrity, replicates each log on multiple servers for increased durability, and uses dynamic Byzantine fault-tolerant quorum protocols to ensure consistency among replicas.

Figure 6.11 illustrates the main components of the system. The primary role for most machines participating in the system is that of *storage server*. Storage servers are responsible for long-term storage of extents; they also handle requests to append data to existing extents, to convert key-verified data to hash-verified form, and to read data from the system. To ensure availability and durability in the presence of Byzantine failures, the storage servers implement replicated state machines. Storage servers rely on distributed hashtable (DHT) [DZD<sup>+</sup>03] for query routing.

To access the system, a client communicates with a nearby *gateway* using RPC. The

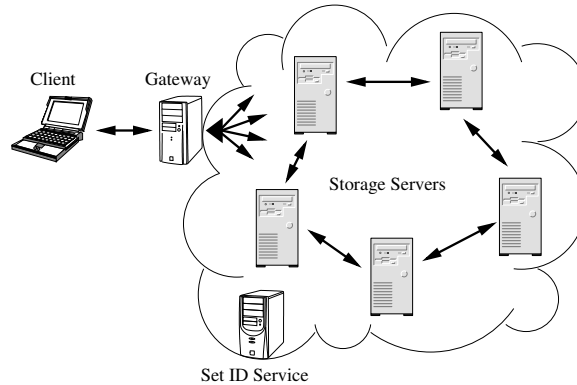


Figure 6.11: **Components of Antiquity:** Each node in Antiquity serves as a storage node and a gateway for nearby clients. The administrator identifies sets of storage nodes to host extents.

gateway uses the underlying DHT to locate the appropriate storage servers to execute the request. For operations that update the state of the storage servers, the gateway forwards the message to each replica. If a client cannot obtain a satisfactory result through a given gateway, it can retry the operation through an alternative gateway.

The *administrator* tracks the availability of storage servers provides the storage set identification service described in Section 6.3. That is, it identifies sets of storage servers to participate in the management of an extent. The prototype implements this service as a single machine that creates storage sets based on the neighbor lists from the underlying DHT. Design of a more robust and fault-tolerant storage set identifier service is planned.

This prototype is written in Java. It uses the Java implementation of 1024-bit RSA cryptography and makes extensive use of RPC. It uses the Bamboo DHT [RGRK04].

### 6.6.2 Experimental environment

We use several different shared test-beds to run our experiments. The *storage cluster* is a 64-node cluster; each machine has two 3.0 GHz Pentium 4 Xeon CPUs with 3.0 GB of memory, and two 147 GB disks. Nodes are connected via a gigabit ethernet switch. Signature creation and verification routines take an average of 7.8 and 0.5 ms, respectively.

The *test cluster* is a 42-node cluster; each machine has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory, and two 36 GB disks. Signature creation and verification takes an average of 24.9 and 1.4 ms, respectively. The cluster shares a 100 Mbps link to the external network.

Finally, *PlanetLab* is a distributed test-bed for research. We use 300 heterogeneous ma-

chines spread across most continents in the network. While the hardware configuration of the PlanetLab nodes varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. Signature creation and verification take an average of 19.0 and 1.0 ms, respectively.

The prototype has been running semi-continuously in two separate installations. We run a small test installation on the storage cluster and a larger installation on PlanetLab. Both installations are configured to replicate each extent on four storage servers.

### 6.6.3 Performance results

At an early stage of the Antiquity development, we used the several microbenchmarks to measure the preliminary performance of the system and verify the benefits of the extent-based API.

**The throughput microbenchmark:** The extent-based API should improve client write throughput. When storing multiple blocks, the client needs to provide only a single signed certificate to authorize all of the changes. Because computing the signature for a certificate is a relatively expensive operation, this change should result in a significant improvement.

We measure this effect using a simple throughput microbenchmark. In this test, a single client write data as quickly as possible using the `append()` operation. When an extent contains 1 MB of data, the client library calls `snapshot()` and `truncate()` on behalf of the application. Each update contains one or more 4 KB application-level blocks; we measure the throughput of the application as we vary the size of the update. Each test lasts 60 seconds.

Figure 6.12 shows the throughput of a client as the update size varies for the deployment running on the storage cluster. Each data point is the average of 5 tests. The plot shows that clients using the proposed interface can achieve a near-linear improvement in write throughput by amortizing the cost of computing signatures over larger updates. The graph also shows the throughput of a clients using the traditional `put()/get()` interface which requires a separate signed certificate for each block.

**The latency microbenchmark:** The second microbenchmark measures the latency of individual operations. For each operation, we define the latency to be the time from when client begins sending a request to the gateway until the client receives the response. In Figure 6.13, we report the cumulative distribution function (CDF) of the latency of each type of operation assuming 1 MB extents, 32 KB updates, and 4 KB blocks. Figure 6.13(a) reports on the system hosted on the cluster;



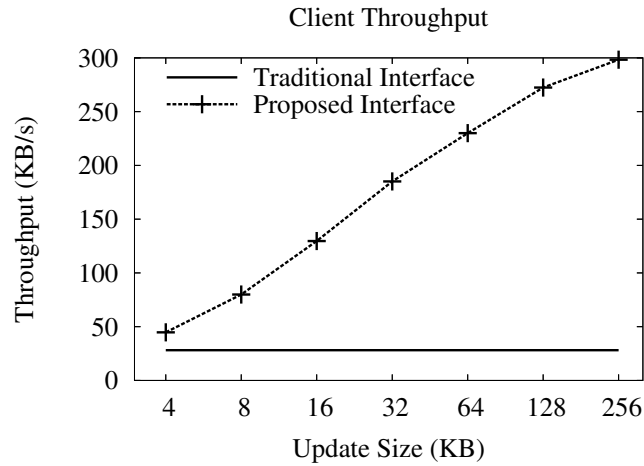


Figure 6.12: **Throughput Improvement Due to Extent-Based API:** Client throughput increases with update size by allowing the clients to amortize the cost of producing certificates over multiple blocks of data.

Figure 6.13(b) reports on the system hosted on PlanetLab.

First, consider the performance on the cluster (Figure 6.13(a)). In general, the latency of an operation depends on how much data the operation must transfer and whether the operation requires the use of the storage set identifier service. The `append()` and `truncate()` operations are the fastest. These operations transfer little or no data and do not require the use of the set identifier service. The `append()` operation (median latency of 78 ms) is faster than the `truncate()` operation (median latency of 132 ms) because the former requires one sequential write to the local disk whereas the latter requires several writes to delete data that is scattered across the disk, and, in the local cluster, disk latency dominates network latency. The `create()` operation is the next fastest operation. Although it does not transfer any data, it must query the set identification server for a set of servers to host the new extent. Finally, the `snapshot()` and `put()` operations are the slowest. Not only must they query the set identification service, but they must also transfer a full 1 MB extent to multiple storage servers. The `snapshot()` operation copies data to a new set of storage servers because the set identification service determines storage sets based on extent name.

The performance trends on the PlanetLab deployment (Figure 6.13(b)) mirror those on the cluster. The `append()` and `truncate()` operations are the fastest. On PlanetLab, however, the `truncate()` operation (median latency of 1293 ms) is faster than the `append()` operation (median latency of 2466 ms) because network latency dominates disk latency. Because of the cost of transferring large extents across the wide area, `snapshot()` and `put()` are very high latency operations;

the median latency for `put()` operations is 13,462 ms and the median latency for `snapshot()` is 23,332 ms. We believe that `snapshot()` takes longer than `put()` because the disk is a highly contended resource in PlanetLab and reading the key-verified extent from disk before transfer can take a significant time.

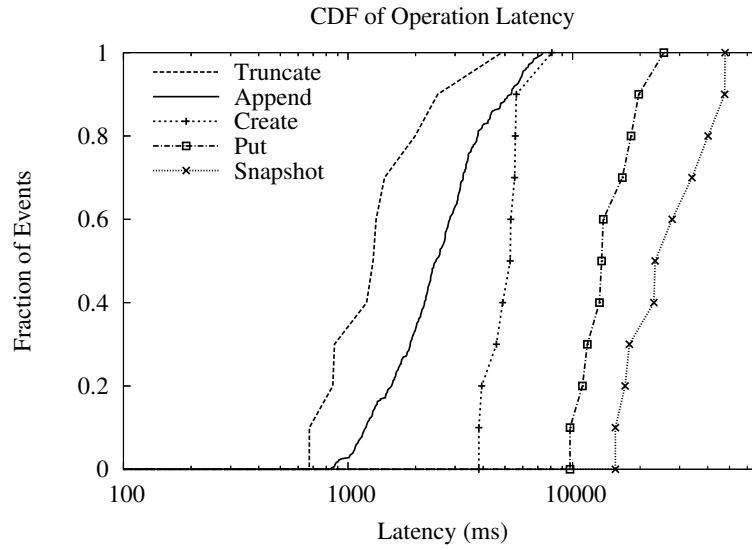
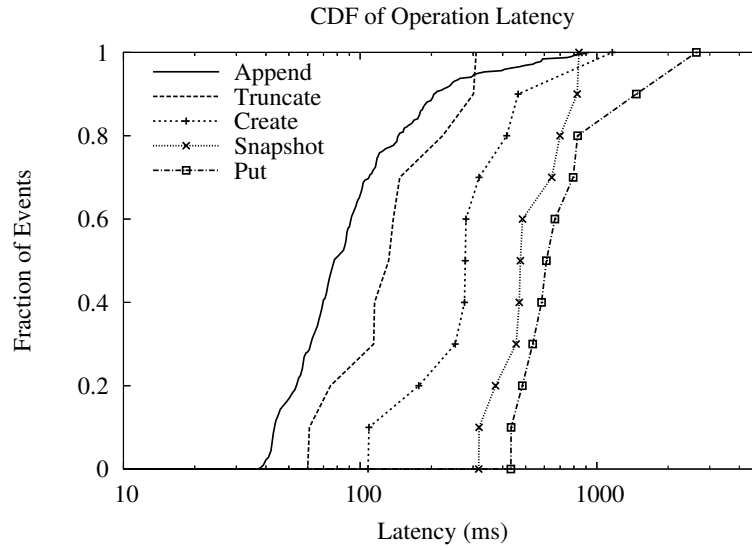


Figure 6.13: **Operation Latency of Extent-Based API:** The CDF of operation latency for a single client through a single gateway on (a) the cluster and (b) PlanetLab. (Block Size = 4 KB, Update Size = 32 KB, Extent Size = 1 MB)

## **Part III**

# **Putting It All Together**

## Chapter 7

# Moxie: A Prototype Remote Storage System for Weakly Connected Clients

In this chapter, we present the design and evaluation of Moxie, a prototype remote storage system for weakly connected clients. Moxie implements the delta-encoding and intention updates techniques described previously in this thesis. It stores data in a secure log that is accessed through the two-level API described earlier. In building the Moxie prototype, we seek to validate the use of the techniques previously studied in simulation in real-world systems. We also demonstrate that these techniques can be combined in a single system.

### 7.1 Moxie Overview

The key components of the Moxie prototype are shown in Figure 7.1. The Moxie prototype includes an implementation of the client library and middleware server. Clients communicate with servers via Remote Procedure Call (RPC) using a custom protocol defined using the External Data Representation (XDR) standard. The protocol supports delta-encoding as described in Chapter 4 and intention updates as presented in Chapter 5. The middleware server stores data to a secure log implementation with an interface as described in Chapter 6.

To support delta encoding, the prototype is designed around the use of the data structure described in Section 4.1. The client and server can manipulate and interpret this structure. The client/server protocol supports operation-based updates, allowing the client to describe changes to be made as a series of block insert and delete operations. The server applies updates and writes new blocks to the secure log.

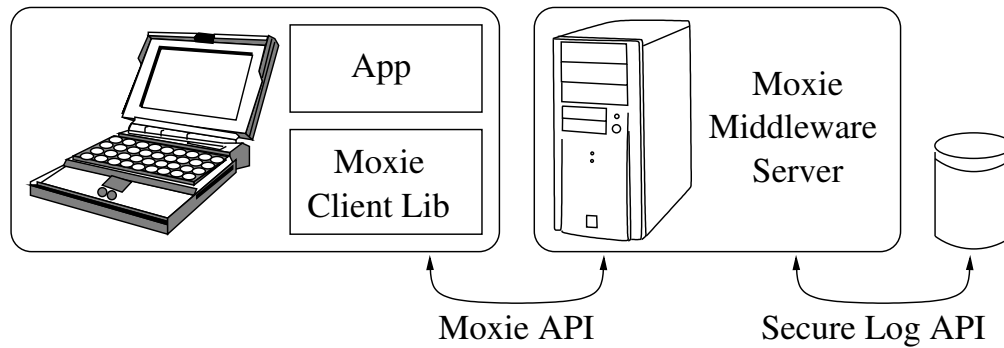


Figure 7.1: **Components of the Moxie Prototype:** The Moxie prototype include an implementation of the client library and the middleware server. The server writes data to a secure log.

The prototype also implements intention updates. When a client modifies a file, it sends a synchronous intent to the server. After the server processes the intent and responds to the client, the application may continue. The corresponding user data is transferred to the server asynchronously as bandwidth is available. By default, buffered data is transferred in a FIFO manner. Data transfer messages have the lowest priority for the network; all other file system traffic is transferred with higher priority and in the order it was generated. The prototype includes two implementations of the agent responsible for managing the transfer of data. The *DB transfer manager* buffers data in a local database ensuring data durability should the client crash. This durability comes with the cost of lower performance because of the data must be copied to and from the database on each update. The *hash transfer manager* buffers data locally in in-memory hashtables until bandwidth is available to transfer it to the server. This transfer manager is fast, avoiding extra data copies. However, the hash transfer manager does not ensure data durability if the client should crash, and the amount of data that can be buffered is limited by the amount of client memory. The hash transfer manager understands how blocks comprise files, and, thus, can support more advanced forms of data transfer reordering. We will show the effectiveness of this advanced feature in Section 7.5. To support intention updates, we extended the secure log API of Chapter 6; we will discuss these changes in more detail below.

The Moxie middleware file server writes data to any backing store that supports the two-level API presented in Section 6. Currently, we have two backing stores that support this interface. The *database log* stores the log in a database. The database log is designed to run on the same machine as the file server. Data is written to the local disc only; it is not replicated. Antiquity [WECK07] is a wide-area distribute storage system that exploits the secure log and API to

provide stable storage from a collection of loosely connected, geographically distributed machines. Antiquity replicates each extent on multiple machines for durability, uses Byzantine quorum protocols to maintain consistency of replicas, and integrates fault-tolerance protocols to recover from outages and Byzantine attacks.

The Moxie prototype parameterizes many features to allow rapid reconfiguration. By reconfiguring the system, we can observe and measure the system with different feature sets. To allow us to evaluate some of the key topics of this thesis, the prototype can be configured to use different write-back strategies and methods for update encoding. The Moxie client implements two write-back strategies: intention updates and unified, synchronous updates. When using intention updates, the client sends intents to the server synchronously; if the intent is accepted, the corresponding data is transferred asynchronously. Intents are signed; data transfer messages are not. To allow comparison, the client can also be configured to use synchronous, unified updates for write-back. The client must sign the entire unified update message. To compute updates, the client can be configured to use one of three methods. Using whole-file write-back, the client updates the entire file if even a single byte has changed. In block-based write-back, the client divides a file into fix-sized blocks and updates only those blocks that have changed. The prototype can also be configured to compute differential updates using Rabin fingerprinting as described in Chapter 4.

To allow us understand how mechanisms for ensuring data privacy and integrity affect the performance of a remote storage system, we can configure the cryptography algorithms used by the prototype. Data privacy is guarded through client-side encryption. We compare the performance of prototype using the null cipher, which leaves the data unencrypted, to data encrypted with the Advanced Encryption Standard (AES) cipher with 128-bit keys. To guard data integrity, the system relies on public key cryptography and digital signatures. We compare the performance of the prototype without signatures, with 3-kilobit RSA signatures, and 256-bit elliptic curve cryptography (ECC) signatures. The key sizes for RSA and ECC cryptography are chosen to provide comparable security [Age06].

## 7.2 Moxie Implementation

The Moxie prototype is implemented as a user-level filesystem. Developing the prototype at user-level allowed us increased flexibility in selecting the programming language and development tools for the project. It decreased development time because bugs in Moxie did not crash the kernel. Finally, it allowed us to use a number of libraries built as part of the OceanStore project.

The Moxie client and server are implemented in the Java programming language, written in an staged, event-driven style [WCB01] using the Bamboo toolkit [RGRK04]. This style, combined with the use of asynchronous I/O libraries, simplifies the implementation of network services by dividing the design among multiple, cooperating stages and avoiding the thread-based concurrency models that are hard to understand [WCB01]. The event-driven model also allows us to exploit a number of libraries and modules built for related systems in the OceanStore project to reduce development time. For example, we use RPC libraries developed for the Antiquity prototype and an asynchronous interface to Berkeley DB originally used in the OceanStore Pond prototype.

The core implementation of the Moxie is approximately 23,000 lines of Java code. Test code and custom libraries are another 11,000 lines of code. The Moxie client/server API defines 15 remote procedure calls.

### 7.2.1 User-Level File System Toolkits

We elected to build Moxie using the FUSE (Filesystem in **USE**rspace) toolkit [Sze04]. The FUSE distribution includes a Linux kernel module and a user-level library. The kernel module can be installed in a running kernel to intercept system calls to the file system at the VFS layer. The kernel module passes all file system requests to a user-level library through a domain socket. The library, in turn, propagates the event to a user-level implementation that has been registered with the library. After handling the request, the file system implementation returns the result to the user-level library, which responds to the kernel module through the domain socket. Figure 7.2 illustrates this architecture.

While numerous options for building user-level file systems exist, we chose FUSE for several reasons. It appears to be a stable, well-supported project with a vibrant user community and has recently been accepted into the official Linux kernel source tree. It has been used to build a number of file systems, including at least one that is in the research literature [CDB04]. Finally, it provides Java bindings so that we can reuse much of code we have already written—including code from simulations, code in the Antiquity prototype [WECK07], and even code from the OceanStore Pond prototype [REG<sup>+</sup>03].

We debated using several other approaches and toolkits. A standard approach for developing user-level file systems is to build a NFS loopback server. We rejected this approach because it limits a file system to using the NFS protocol that, among other limitations, does not include `open()` and `close()` system calls. We considered using the SFS toolkit [Maz01]. We rejected this



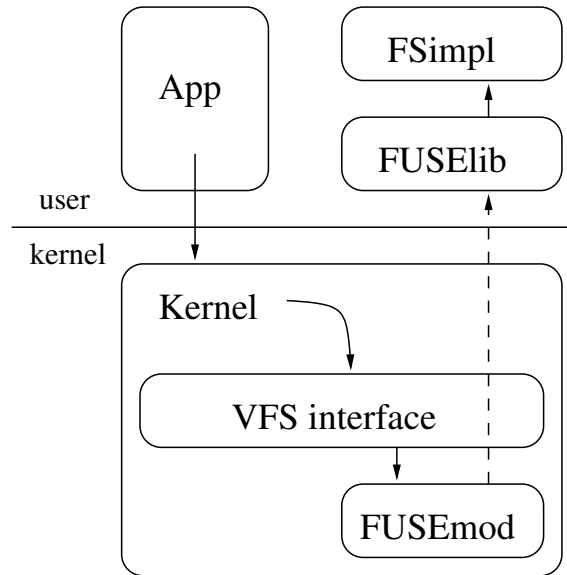


Figure 7.2: **The FUSE Architecture:** The FUSE toolkit includes a kernel module and user-level library. The kernel module receives file system requests through the VFS interface and passes the requests to the user-level library through a domain socket. The library then passes the requests to the file system implementation that has registered with the library. Results pass through components in the reverse order before reaching the application.

library because it provides an NFS interface and requires the use of a complicated programming model. The Arla project [DW98] is a free re-implementation of AFS. Although most of the code runs at user-level, it uses a kernel module called `xfs` or `nnpfs` (depending on the version) to forward file system calls to user level. While `nnpfs` is meant to provide similar functionality to the FUSE kernel module, it is not a complete or mature implementation. The LUFs (**L**inux **U**serland **F**ile**S**ystem) [Mal02] is a project with similar goals and approach as FUSE. Like Arla, however, it is less mature and not well supported. Also, the SFS toolkit, `nnpfs` kernel module, and LUFs project provide no support for programming in Java. Finally, FiST [ZN00] (**F**ile **S**ystem **T**ranslator) is a tool that generates code for stackable file system layers. We rejected this tool because the code it produces is intended to run in kernel-mode which slows development time.

### 7.2.2 Moxie Client

Figure 7.3 shows the key components in the Moxie client implementation. To support the use of Java and the event-driven programming model, we use FUSE-J and a custom programming model translator. FUSE-J [Lev05] is a set of Java bindings that provide access to FUSE through the

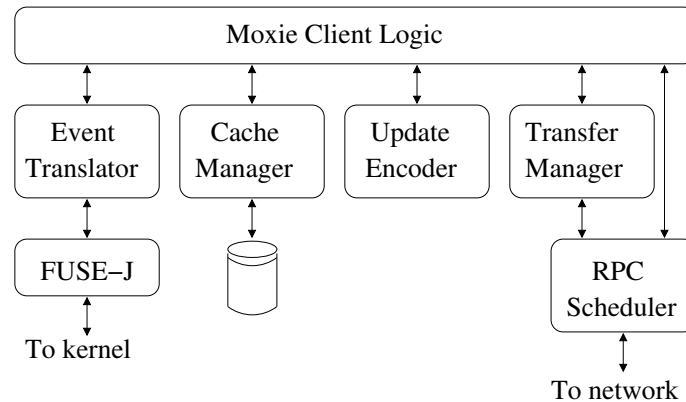


Figure 7.3: **Key Components in the Moxie Client:** The Moxie client is implemented by several by several cooperating stages. The Moxie Client Logic oversees the execution of each request, relying on other stages for handling specialized tasks. All components shown in the figure execute with a single JVM.

Java programming language via the Java Native Interface (JNI). The programming model translator converts FUSE’s threaded programming model to Moxie’s event-driven style. It supports multiple concurrent requests and records a number of statistics for evaluating the performance of the file system implementation on the different types of requests. The translator is not specific to Moxie; it could be used by other developers building user-level file systems in the Bamboo framework.

The Moxie Client Logic oversees the actions initiated by the request, relying on other stages to help service the request. The Cache Manager maintains the local disk cache. Because the prototype implements whole-file caching with write-back on close, all `read()` and `write()` requests are executed in the cache. When an application issues a `close()` request after modifying file data, the Update Encoder computes the changes that must be sent to the server. The Update Encoder also maintains the data structures, like the file summaries, that are used to compute updates. The Transfer Manager supervises the asynchronous write-back of data when using intention updates. When the client uses unified updates, the Transfer Manager is not needed. Finally, all requests that must be passed to the server are sent to the RPC scheduler that manages the order that messages are transmitted to the network.

### 7.2.3 Moxie Server

Figure 7.4 highlights the key components in the implementation of the Moxie middleware file server. In a similar fashion to the client implementation, the Moxie Server Logic oversees the handling of each request. The server maintains two caches: the Data Cache stores recently

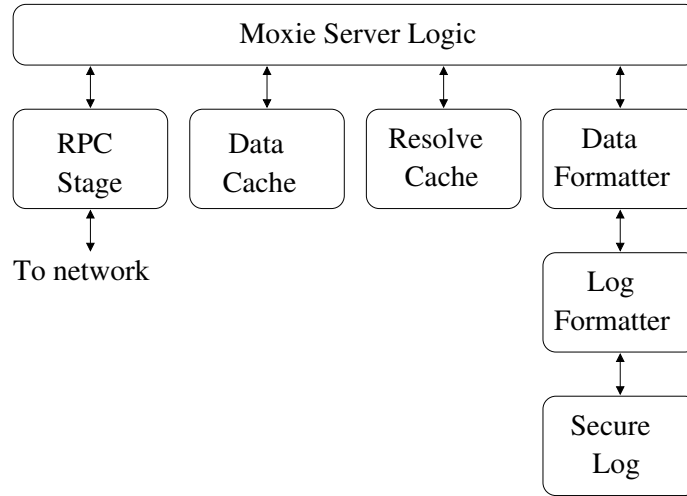


Figure 7.4: **Key Components in the Moxie Middleware File Server:** The Moxie server is implemented by a collection of cooperating stages. While the Moxie Server logic oversees the execution of each request, most of the complexity at the server serves to interact with the log. All components shown in the figure execute in a single JVM.

accessed data blocks and the Resolve Cache stores information to allow the server to map file paths to addresses in the log. Most of the complexity at the server serves to translate data structures to and from the format stored in the log. The Data Formatter manages the storage and retrieval of the B-tree-like data structure to and from the log. The Log Formatter manages the inode map [RO92] that allows the server to find files and directories stored in the log. The Secure Log stage uses the log API to store and retrieve data in the log which may be stored locally or distributed and replicated across the wide-area.

## 7.3 Implementation Issues

In translating the ideas of Chapters 4, 5, and 6 into a single prototype implementation, we discovered several interesting challenges. In this section, we describe three of those issues.

### 7.3.1 File Summaries and Delta Encoding

As the development of the prototype progressed, we were frustrated to observe that the delta-encoding mechanisms were not detecting any commonality between consecutive versions of files. This was especially puzzling because the implementation of the core algorithms was the same code that was used in the simulations of Chapter 4.

Upon further investigation, we discovered that the most common uses of the file system API would prevent our implementation from ever detecting any commonality. When writing a new version of a file to storage, most applications and utilities first issue a `truncate()` or `remove()` operation to erase the old contents of the file. As a consequence of this procedure, between any two application-level versions of a file, there is, in the file system, an additional version that contains no data. Consequently, in the common case, no two consecutive versions of a file would ever exhibit any commonality. Of course, the intermediate version which contained no data is typically very short-lived.

To overcome this issue, we exploit the fact that, at the lowest level, the data stored in Moxie is stored in a log and is not ever deleted. Because all data is stored in non-erasable, log-based storage, the client can compute delta-encoded updates against *any* previous version of the file, not just the immediately preceding version. Based on common usage patterns, the client should compute an update based not on the version of a immediately preceding version of the file, but rather against the version preceding the transient, empty version of the file.

To allow the client to compute updates against older versions of a file, the client must maintain the file summaries of multiple previous versions. Practically, Moxie maintains the file summary for the previous two versions of the file. Considering that one of these summaries typically describes a version of the file that contains no data, the storage overhead of one extra summary per file is small. A couple of simple heuristics determine which of the previous file summaries to use as the base version for computing updates. With this additional insight and upgrade, the Moxie prototype was able to exploit commonality between versions of a file as predicted by the simulations of Chapter 4.

### 7.3.2 Control of Data Write-back

While testing Moxie's support for intention updates, we observed that the order requests were being sent to the network did not match the desired results. In particular, we observed that low-priority data transfer messages were often transmitted before other high-priority requests. We will present the experimental results that led to this observation in Section 7.5.4.

We learned, after further investigation, that the problem was due to the fact that the Moxie client, operating at the user-level and inside the JVM has no direct access to the network buffers. The Moxie client can query when the underlying network connection can accept more data to write, however, it has no control over how long the data will wait in the network buffer before being

---

**Interfaces for Intention Updates:**


---

```
status = reserve(H(PK), cert, reservation[ ]);
status = fill(H(PK), reservation[ ], data[ ]);
```

Table 7.1: **Extensions to the Extent-Based API for Intention Updates:** To support intention updates in an extent-based content-addressable storage system, we extend the API of Chapter 6 with two new operations that reserve space in the extent and fill the reservation.

transmitted. This makes it very hard to schedule network traffic at the application level.

Interestingly, this is the same problem described in the documentation for Linux’s Advanced Routing and Traffic Control [Hub04] that we use to shape traffic to emulate wide-area network connections. That documentation explains that to schedule traffic effectively, one must “own the queue” of the bottleneck link.

We believe that, to implement intention updates in an optimal manner, the Moxie client must have low-level access to the queue at the network. It is not sufficient merely to place an order on requests as they *enter* the queue. To prioritize traffic, the Moxie client must be able to control the order that requests *exit* the queue and enter the network.

### 7.3.3 Supporting Intents in the Secure Log

To support intention updates, the data format and underlying storage layer must be able to record the existence of a data block without access to the actual block. The extent-based API presented in Chapter 6 does not support this.

To support intention updates in the extent-based API, we extended the API with two new operations. The extension introduces the concept of a *reservation* in an extent. A reservation describes a block of data that the system will receive at a later time. The reservation includes a secure hash of the block and the amount of data in the block.

Table 7.1 shows the new operations that an extent-based content-addressable storage system must implement to support intention updates. The `reserve()` operation is similar to the `append()` but includes one or more reservations instead of data blocks. The storage system handles a `reserve()` request in a similar manner as an `append()` request; it confirms that the certificate includes a valid verifier that summarizes the state of the extent after adding the data described by the reservations and, if the certificate is valid, updates the extent to reflect the changes. A reservation supplies the storage system with enough information to compute the extent’s new verifier and track the size of the extent’s contents.

The client uses the `fill()` operation to consummate a reservation with the actual data. The storage system finds the reservation stored in the extent, confirms that the block matches the data described in the reservation, and then inserts the data into the extent. Note that the `fill()` does not require a certificate. The secure hash in the reservation allows the storage system to verify the integrity of the data in the request. If the storage system cannot find a matching reservation or the data does not match that described in the reservation, then the request can be ignored.

## 7.4 Experimental Environment

To evaluate the Moxie prototype, we run the client and server implementations on two machines connected via a local-area network. We emulate weak network connectivity by restricting bandwidth and imposing latency on all traffic between the two machines.

The Moxie client runs on a machine with a dual-core, 3 GHz Intel Pentium 4 processor and 1 GB of memory. The machine runs the Debian “Etch” distribution with the 2.6.17 Linux kernel. The Moxie server runs on a machine with two dual-core, 3.8 GHz Intel Xeon processors and 6 GB of memory. The server machine runs the Red Hat 3.4 distribution with a 2.6.9 Linux kernel. Both the client and server run in Sun’s Java 1.6 “Mustang” JVM. Elliptic curve cryptography is provided by Mozilla’s Network Security Services (NSS) libraries.

We use Linux’s traffic control mechanisms [Hub04] (and the `tc` utility) to emulate weak network connectivity. All traffic related to the prototype is enqueued to a queueing discipline that includes a token bucket filter (`tc`’s `tbpf` queueing discipline) to limit outbound bandwidth and a network emulator (`tc`’s `netem` queueing discipline) to impose latency on all packets. We control outbound traffic on both the client and server, enabling the emulation of asymmetric connections. We verified the latency and bandwidth of the traffic control configuration using a custom ping application written using the same tools as the Moxie prototype that can be configured to carry variable-sized payloads.

### 7.4.1 User-Level NFS Implementation

For comparison against a known system, we have built a user-level NFS (version 2) client and server. The file system is implemented using the same tools used to create the Moxie prototype including FUSE, FUSE-J, Bamboo, the programming model translator, and RPC libraries. At the server, data is stored on the local file system using Java’s I/O facilities. To access file sys-

tem interfaces not accessible by Java’s API, we spawn external processes. This approach is used to handle `link()`, `symlink()`, `readlink()`, `rmlink()`, `stat()`, `chown()`, `chgrp()`, `chmod()`, `atime()`, and `mtime()` requests. Although the number of request types that must use external processes is significant, these request types comprise only a small percentage of total requests in most workloads. The common file operations (`create()`, `open()`, `close()`, `read()`, `write()`, and `remove()`) and directory operations (`mkdir()`, `rmdir()`, `readdir()`) can be handled purely in Java. The NFS implementation supports attribute caching and data caching as described by Stern, Eisler, and Labiaga [SEL01] and Smith [Smi06].

## 7.5 Experimental Results

We now describe a number of experiments and their results that demonstrate the features and advantages of the Moxie prototype.

### 7.5.1 Delta-Encoded Updates

We begin by demonstrating the effectiveness of delta-encoded updates. To exercise this feature, we apply a workload that upgrades a software distribution. We first copy the source code distribution for Tcl 8.4.13 into the remote file system. This distribution includes 754 files and directories and 16 MB of data. We then upgrade the tree to the development version 8.5a4. This latest development version includes 23 MB of data in 1501 files and directories.

To update the distribution, the file system performs 2112 updates. The workload produces more than one update per file because files that already exist in the previous version are first truncated and then filled with new data. Without delta-encoding, the updates would write 18.5 MB to the remote file system. Delta-encoding is able to reduce the amount of data written by 16.6%.

We execute this workload on a file system hosted by Moxie under several different network types. We measure the total runtime of the workload with and without delta-encoding enabled. For this experiment, we configure the prototype to use unified updates. The client does not sign requests or encrypt data.

Figure 7.5 shows the effect that delta-encoding has on the total runtime of this workload. The total runtime of the workload increases dramatically as the bandwidth of the network decreases. Delta-encoding has little if any impact on the initial copy phase of the workload; this is not surprising because the initial phase does not include any opportunities to find similarity to previously stored

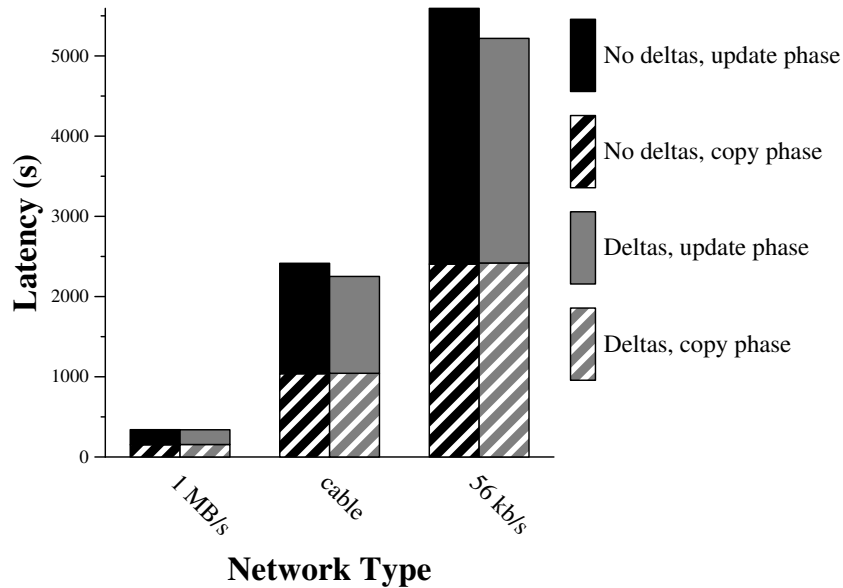


Figure 7.5: **Effect of Delta Encoding on Runtime:** Delta-encoding improves the total runtime of a workload depending on the amount of commonality in the workload and the bandwidth of the network. Comparing the copy phase of the workload shows that delta-encoding imposes little overhead when copying files into the file system. Delta-encoding decreases the runtime of the update phase by transferring fewer bytes to the server.

versions. It is encouraging, however, that the overhead of fingerprinting each file and creating file summaries does not slow the client appreciably.

Delta-encoding does reduce the total runtime of the second phase. At high bandwidths, the improvement due to delta-encoding is negligible because the small amount of bandwidth conserved is insignificant compared to the available bandwidth. At lower bandwidths, however, the improvement is appreciable. At 56 kb/s, the workload completes more than 6 minutes faster when delta-encoding is used. Measured in percentages, the improvement in runtime approaches the fraction of commonality in the workload.

### 7.5.2 Intention Updates

To demonstrate the benefits of intention updates, we write a large data set to the file system hosted by Moxie. Specifically, we store 10 digital photos with a cumulative size of 24 MB using the standard file system copy utility. We record the client-perceived response time, the visibility latency, and the durability latency using both traditional, synchronous, unified updates and intention updates. For this experiment, the client does not sign requests or encrypt data.



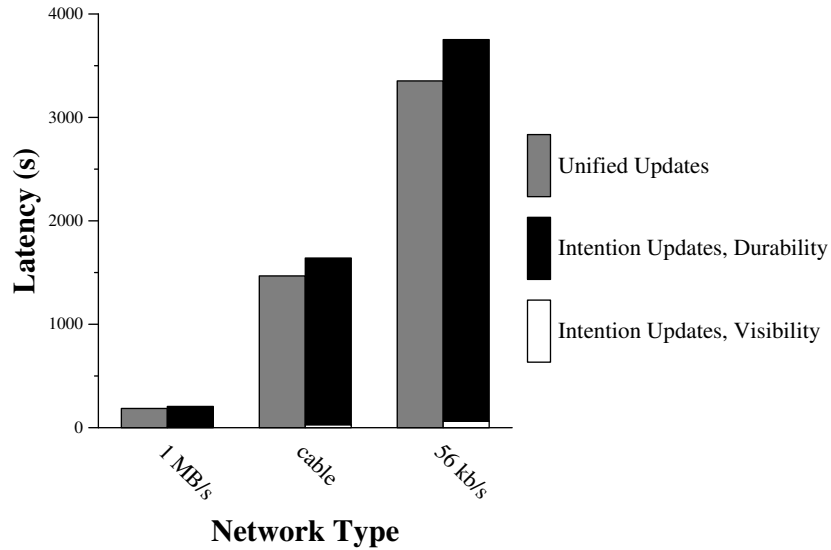


Figure 7.6: **Effect of Intention Updates on Runtime:** Intention updates reduce the visibility latency and response time by more than an order of magnitude compared to traditional, synchronous, unified updates. With intention updates, the durability latency is roughly equal to the response time of traditional updates. The current implementation introduces a slight additional delay in durability.

Figure 7.6 compares the amount of time to execute this workload using a variety of different network types. The bars corresponding to unified updates show the response time. The client machine blocks during this period. With unified updates, the data is visible and durable before the client received a response.

The bars corresponding to intention updates show both the visibility latency (the thin white sliver at the x-axis) and the durability latency. The response time is slightly longer than the visibility latency by the one-way network latency. With intention updates, the visibility latency and response time are more than an order of magnitude faster than the response time of traditional updates.

The time to durability is greater for intention updates than unified updates. Some durability latency penalty is expected because of the bandwidth overhead of using intents and the low priority given to data transfers. The durability latency penalty of the Moxie prototype is greater still due to implementation issues. The current implementation of the DB transfer manager is block-based rather than file- or object-based. Although this gives the data transfer manager great flexibility in ordering data transfers messages, it does introduce additional overhead in the data transfer phase. We believe that further optimization of the data transfer manager component could reduce the durability latency penalty.

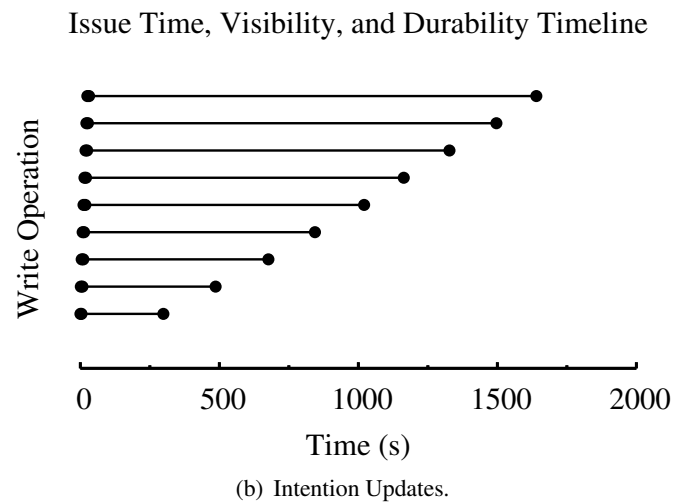
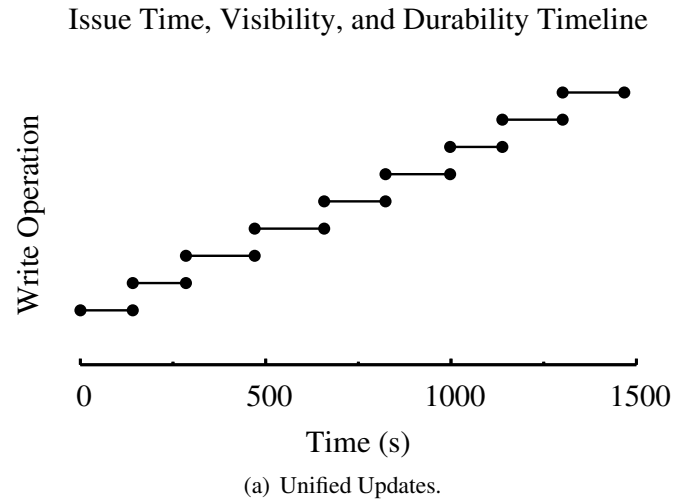


Figure 7.7: **Visibility and Durability Timelines:** The timelines illustrate how the execution of individual updates leads to the behavior reported in Figure 7.6. Each horizontal line corresponds to an update; the points on the line highlight important points in the life of the update. (a) For unified updates, the points show when the update is issued and when the response is received. (b) For intention updates, the points represent when an update is issued, when the update is made visible, and when the update is made durable.

Refer to Figure 7.7 to visualize the behavior of intention updates. These graphs present a timeline of activity for the workloads reported on in Figure 7.6. Each horizontal line on the timeline refers to the copy of a single photo to Moxie. The points on the line show the timing of significant parts of the process. Figure 7.7(a), for example, shows the time when the client issues the update request to write a photo to the server and the time the client receives the response. The “step-shaped” pattern in the figure shows that the client transfers a photo entirely before continuing on to another photo. The user does not receive a response until the entire operation (the transfer of 10 photos) completes.

Figure 7.7(b) shows the timeline when intention updates are used. The points on each line correspond to when the update is issued, when the update is visible, and when the update is durable. Given the scale of the figure, the first two points are mostly overlapping. The client receives a response to each update quickly (after the update is visible) and can then issue the next update. Notice that the issue time of the updates is staggered only slightly. As long as the client continues to issue updates, the data transfer proceeds only with low priority. This allows the updates to complete promptly, and the user receives a response quickly. After the last update completes, the client allocates the full bandwidth of the link to transferring the data. The data transfer manager pushes the data out to the network in a first-in, first-out manner. This timeline illustrates how intention updates benefit weakly connected clients by prioritizing small intent messages to make updates visible and overlapping the transfer of data with other operations.

### 7.5.3 Reordering Data Transfers

Intention updates allows clients to keep server state current while providing clients with exceptional flexibility in scheduling how the user data is transferred to the server. To demonstrate the potential of this feature we devised a simple microbenchmark in which two clients share data. One weakly connected client serves as the “writer”. The writer first populates the shared filesystem with a set of files and then sends a continuous stream of intention updates that target a randomly selected file. We adjust the update rate of the writer to control the amount of data buffered at the client in the data transfer manager. Another client acts as the “reader”, repeatedly issuing reads to the server. The server responds to each read with the latest visible data. Because the latest visible data may not be durable, the server must sometimes block until the writer transfers the data to the server. Without data reordering, the server (and the reader) must wait until the data buffered at the writer reaches the front of the data transfer queue and is transmitted. With data reordering,

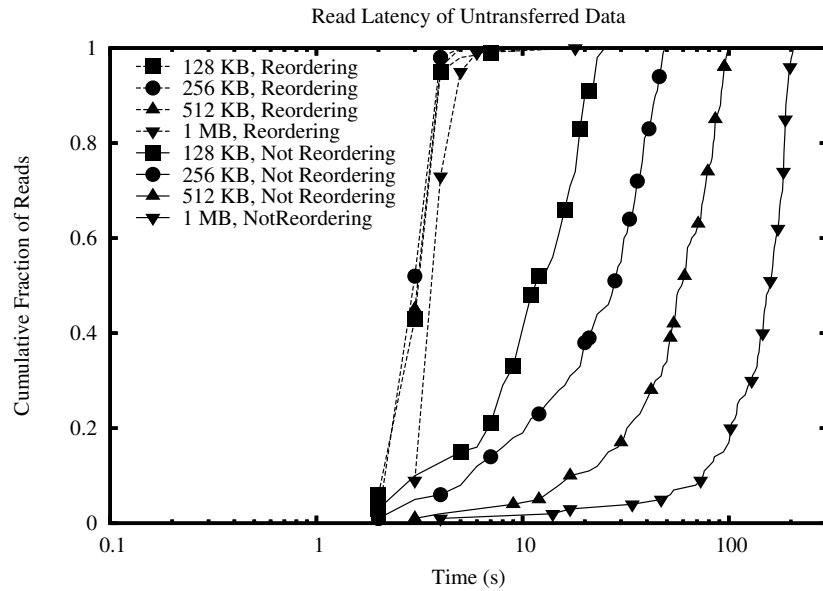


Figure 7.8: **The Benefits of Reordering Data Transfers:** Intention updates afford the client great flexibility in ordering how data is sent to the server. This sharing microbenchmark shows how the server can suggest reordering to improve service for clients reading data from the server. Two users share a set of 100 8-KB files. The user constantly writes files at random; the other repeatedly reads random files. The writer is weakly connected, using a telephone modem; the reader is strongly connected.

however, when the server receives a read request for data that is not durable, it sends the writer a “hint” requesting that the requested data be moved to the front of the data transfer queue. With this optimization, the data is transferred to the server sooner, and the server can respond to the reader with less delay.

For this experiment, the clients use the hash transfer manager, which includes support for reordering data in the transfer queue and understands how blocks relate to files. The data shared between the reader and writer consists of 100 8-kilobyte files. The writer is connected to the server via a telephone modem. The reader is strongly connected to the server, allowing us to focus on clearly on the benefits of reordering data transfers between the writer and server. We measure the latency of 100 read operations which require the server to block.

Figure 7.8 shows the results of this set of experiments. We vary the update rate of the writer to maintain a specified amount of data buffered in the data transfer queue and alternate whether the server sends hints to the writer to reorder data in the transfer queue. The solid lines correspond to tests in which the server passively waits for requested data without sending the writer

hints. The average latency to satisfy a read request increases as the amount of data buffered at the writer increases. In effect, the reader is requesting data that is randomly and uniformly distributed in the queue. As the queue length increases, the time for the randomly selected block to reach the front of the queue and be transmitted to the server also increases. Of course, sometimes the reader randomly selected a block that is already near the front of the queue. In those instance, the reads will still return relatively quickly.

The dashed lines correspond to tests in which the server sends hints to the writer suggesting that data in the transfer queue be reordered to better server the needs of other clients. When data reordering is used, the latency observed by the reader is independent of the size of the data transfer queue on the writer. When the reader requests data and the server submits a hint to the writer, that requested data is immediately moved to the front of the transfer queue and transmitted to the server.

Of course, the writer's buffer size affects the likelihood that the reader will be able to fetch the file it is requesting without blocking. When the writer has a large local buffer, the data the reader requests is more likely to be in that buffer, and the reader will be more likely to request a file that requires the server to block. In these experiments, the server was required to block on between 14% (with a 64 KB buffer) and 81% (with a 512 KB buffer) of all requests from the reader.

These results show that reordering data transfers can be effective in satisfying requests for data from other clients. Although this technique can reduce the time requested data is stuck in the writer's network queue, it cannot reduce the time it takes to transmit data across the network. In the test above, the reader was able to retrieve files in just a few seconds because the benchmark included only small files. Even with data reordering, the time to retrieve large file would be much longer. For example, if the reader requests a 5 megapixel ( $\approx 2.5$  MB) digital photo, it would take roughly 2.5 minutes for the writer to upload the photo over a cable modem (and more than 6 minutes using a telephone modem). It is unlikely that a reader would wait that long. The writer could, however, offer a smaller, transcoded version of the requested data to the client. For example, by offering a small, thumbnail photo ( $\approx 25$  KB), a writer connected via cable modem could provide the reader with data in just a couple of seconds (and in less than 5 seconds using a telephone modem). Because intention updates ensure that the server always stores the current metadata for each object, the reader can make the ultimate decision about how long to wait for data. The utility of this transcoding scheme will vary greatly depending on the application. While it seems well-suited for media (photo, video, audio) applications, it is likely not appropriate for office and development workloads.

Phase	Action
1	Construct a directory tree.
2	Copy files into the directory tree.
3	Examine the status of every file in the tree.
4	Scan every byte in the tree.
5	Compile the application in the tree.

(a) Andrew Benchmark.

Phase	Action
1	Extract the archive.
2	Examine the status of every file in the tree.
3	Scan every byte in the tree.
4	Configure the source for compilation.
5	Compile the application.

(b) Tcl Benchmark.

Table 7.2: **Comparison of the Andrew and Tcl Benchmarks:** (a) The Andrew benchmark consists of five phases that populate a directory tree, access data stored in the tree, and then compile an application in the tree. (b) The Tcl benchmark consists of five phases that populate a directory tree, access data stored in the tree, and compile an application. The actual work performed in each phase differs slightly from the Andrew benchmark.

#### 7.5.4 The Tcl Benchmark

To observe the response of Moxie under more rigorous workloads, we created the Tcl benchmark. The Tcl benchmark is a synthetic benchmark based on activities common for software developers. It is modeled after the famed Andrew benchmark [HKM<sup>+</sup>88]. While we do not necessarily believe the most weakly connected users of a remote storage system will be executing such workloads, such a benchmark does reflect a least one type of real-world workload. It is also a powerful tool for comparing Moxie against other file systems in the literature because it has become customary for researchers to evaluate their systems with benchmarks like the Andrew benchmark.

The original Andrew benchmark was developed to provide a synthetic workload that had similar characteristics to typical developer workloads. The benchmark ran a script that executed five phases shown in Table 7.2(a). The directory tree for used by the Andrew benchmark included 93 files with a total size of 616 KB. The small size of the original Andrew benchmark renders it unacceptable for evaluating today’s systems; however, similar benchmarks using larger data sets do remain useful. We construct such a benchmark based on the Tcl source code distribution. The phases of the Tcl benchmark are enumerated in Table 7.2(b). The source code for Tcl (version 8.4.13) is distributed as a 3.4 MB compressed archive. When extracted, the archive contains 754

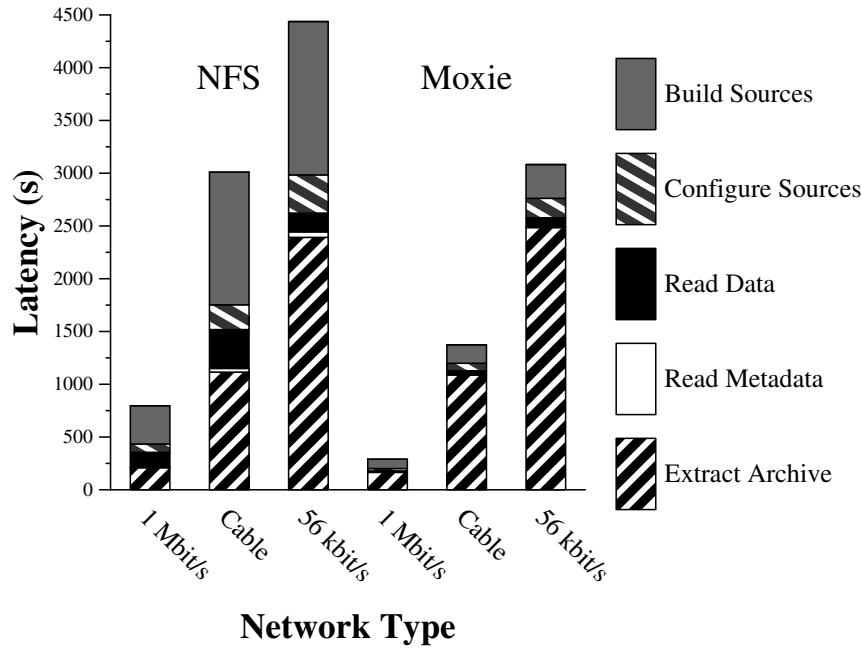


Figure 7.9: **Comparing Moxie Against NFS:** With the Tcl benchmark, we compare the behavior of Moxie and NFS. The benchmark contains mostly small files which limit the effectiveness of Moxie's mechanisms. However, the benchmark demonstrates a surprising benefit from delta-encoding in the configuration and compilation phases of the benchmark.

files and directories with a combined size of 16 MB. After configuring and building the tool for a Unix-based machine, the tree contains 18 MB of data among 838 files.

We use the Tcl benchmark to compare the behavior of Moxie against the well-known remote file system NFS. We use the implementation of NFS version 2 described in Section 7.4.1. Figure 7.9 displays the time it takes Moxie and NFS to execute the different phases of the benchmark with varying network types. For these experiments, Moxie is configured to use intention updates and Rabin fingerprint-based delta encoding. The client does not sign requests or encrypt data. Though Moxie outperforms NFS, most of the improvement is observed in the compilation phase of the benchmark.

We were surprised and disappointed that Moxie's intention update mechanism did not result in reduced runtime for the first phase of the benchmark that extracted the archive. Upon further inspection, we found several reasons for this behavior. First, to copy a file extracted from the archive to the remote file system, the application performs a series of operations including querying the remote file system to ensure that the file does not exist, creating the file, and finally writing the data to the file. The client must issue a synchronous request to the server to satisfy each

request. The latency of the synchronous requests impacts the runtime of the operation more than the cost of transferring the data. One might consider a form of speculative execution to reduce the cost of the multiple, synchronous round trips [NCF05]. Secondly, the small size of the files extracted from the archive reduces the effectiveness of Moxie's intention updates. Given Moxie's overhead, including the cost of computing file summaries and delta-encoded updates and managing intents, masks the benefits that are obtained by moving the transfer of small amounts of data off the critical path. Finally, we observed that we were not able to control the order of requests on the network as is required for intention updates. We will provide more details regarding this last point below.

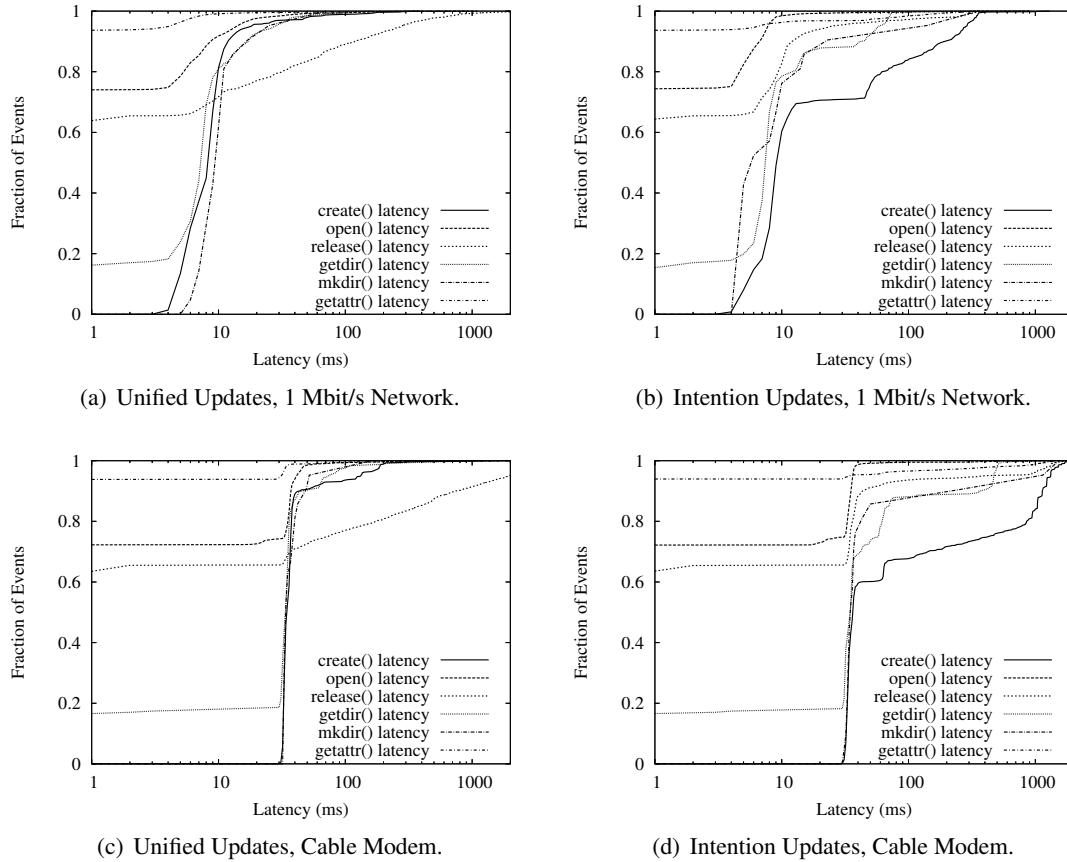
Both NFS and Moxie execute the next two phases of the benchmark quickly. The phases read the metadata and data for each object. Efficient caching policies enable both file system implementations to perform well. NFS does generate some network traffic in these stages because its time-based caching policies require periodic contact with the server to exchange state about each file. Moxie's callback-based caching policies allow it to satisfy reads without contacting the server.

Finally, Moxie outperforms NFS in the last two stages of the benchmark that prepare the source for compilation and build the software package. These phases require the file system to service interleaved read and write requests to many files. Moxie's caching policies help in these phases, satisfying most reads from the local cache. Interestingly, Moxie's support for delta-encoding seems to be a significant part of Moxie's advantage over NFS. It is not, however, the delta-encoded updates which benefit Moxie. Rather, the support for delta-encoding enables Moxie to detect when, though a file's metadata has changed, the file's data remains unchanged. This reduces the amount of data that the server must transfer to the client.

We also use the Tcl benchmark to observe Moxie's response to different types of file system requests. Figure 7.10 includes a series of plots that show the cumulative distribution function of latency for several of the most common and important operations performed by the file system. Plots 7.10(a) and 7.10(c) show the distribution when the prototype is configured to use unified updates at two different network speeds. Plots 7.10(b) and 7.10(d) show the results of similar tests when the client uses intention updates. Curves that do not meet the x-axis represent operations that may be serviced from the local cache without communication with the server. The y-intercept corresponds to the percentage of such requests that can be satisfied locally.

Consider Plot 7.10(a) showing the system response using unified updates across 1 Mbit/s network connection. It shows that most requests (that cannot be handled locally) take between several milliseconds and several tens of milliseconds. The performance for most operations is relatively consistent. The curve with the longest tail corresponds to the `release()` operation which is issued





**Figure 7.10: Operation Latency Distribution in Various Configurations:** The series of plots show the distribution of latency for various operations as write-back strategy and network speed vary. These plots demonstrate how intention updates reduce update latency compared to unified updates. They also illustrate the problems with Moxie's network scheduling implementation. Because it does not have direct access to the network interface's buffer, the Moxie client is not able to ensure that message priorities are strictly observed. Consequently, some high-priority messages are queued behind background transfer traffic.

when the last open file descriptor to a file is closed. If a file is modified, the `release()` operation triggers write-back to the server. Note that the curve does not cross the x-axis; in more than 60% of all `release()` calls, the file is unmodified and the client does not need to write-data back to the server. The plot shows that it can take up to 1 second to complete the the operation when write-back is required. The key factor determining the latency to complete the `release()` operation is the amount of data that must be transferred to the server.

Compare Plot 7.10(a) to Plot 7.10(b) which describes an identical configuration except that intention updates are used. First note that the tail of the distribution corresponding to the `release()` operation is much shorter with intention updates. With intention updates, the latency of `release()` depends much less on the amount of data that must be written back and is, in general, much shorter. The short tail shows that intention updates are improving response time, as intended. Unfortunately, this plot also shows that the performance of other operations is much less consistent than in a system that employs unified updates. This inconsistency occurs because the Moxie client cannot strictly enforce the priority assigned to network messages. Because the Moxie client does not have direct access to the network buffers, some low-priority data transfer message are queued ahead of other high priority messages. We describe the problem above and in Section 7.3.

The latency of `create()` operation is especially affected in this workload because, when extracting the archive during the first phase of the benchmark, the application triggers the write-back of one file immediately before creating another file. Some data transfers requests for the write-back operation are queued to the network ahead of the subsequent `create()` request.

Plots 7.10(c) and 7.10(d) show similar trends using a slower network connection.

To estimate the potential performance of an implementation that can schedule traffic effectively, ensuring that high-priority messages are not delayed by low-priority messages, we modified Moxie to use the Linux traffic shaping functionality to enforce message priorities. Specifically, we changed the middleware server to accept requests on multiple ports. All data transfer messages were directed at one port; all other messages were destined for another port. We then altered the Linux traffic control configuration running on the client to filter messages based on the destination port and prioritize data transfer messages below other traffic.

Figure 7.11 shows the runtime of Moxie using Linux's traffic shaping tools to support intention updates versus the runtime of Moxie when it attempts to schedule traffic at the application-level. The results show that properly enforcing message priorities can have a significant impact on total runtime, especially at lower bandwidths. At higher bandwidths, the penalty of for failing to prioritize message properly is small because the transmission delay for data transfers is small.

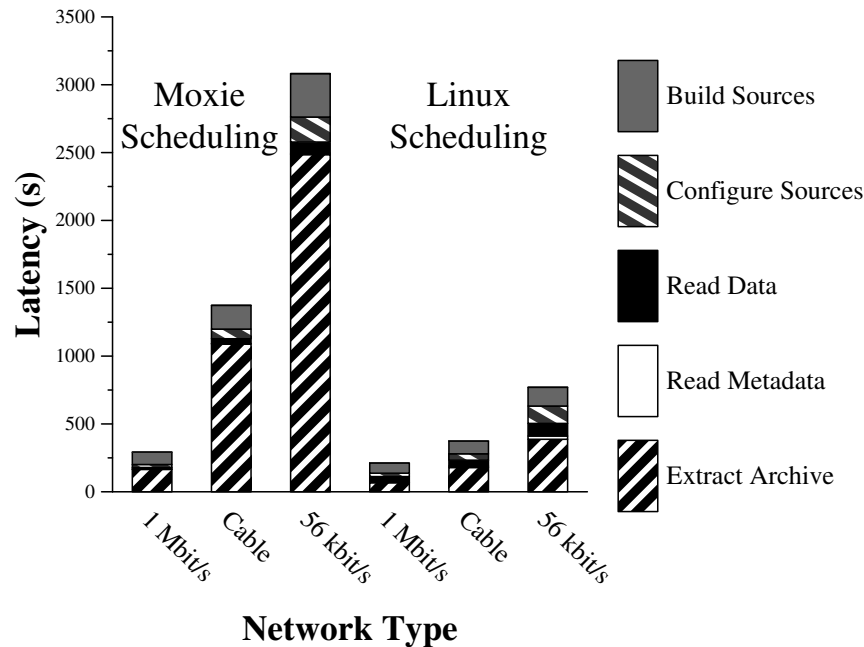


Figure 7.11: **Scheduling Traffic in Moxie Versus Linux:** Figure 7.10 shows that Moxie is unable to enforce the priority of network messages at the application level. To estimate the performance of system that does schedule traffic properly, we modified Moxie to use the Linux traffic shaping tools to enforce message priorities. This plots shows the results of that experiment. By scheduling network traffic correctly, we reduce the runtime of the Tcl benchmark dramatically, especially at lower network bandwidths.

Also, in the Tcl benchmark, most files are small, so even if the transfer was in the critical path, the added latency would be small. At lower bandwidths, however, the time to transmit data blocks increases, and, consequently, the penalty for mis-scheduling message increases. By using the Linux traffic shaping tools to schedule traffic correctly, the prototype can reduce the runtime of the Tcl benchmark by 3x over a cable modem and by more than 4x over a telephone modem.

Figure 7.12 shows the effect that proper message scheduling has on the latency of various operations. Compare these graphs to those presented in Figure 7.10. These graphs show that, with proper message scheduling, latency of all operations is relatively predictable. The curves in these graphs do not exhibit the same long tail that we observed in Figures 7.10(b) and 7.10(d). Some of the curves do show a slight tail. This should be expected due to instances when a high priority message finds a low-priority message already partially transmitted. The high-priority message must wait for transmission of the other message to complete. Using the Linux traffic shaping tools to schedule traffic, we do not see evidence, however, that high-priority messages must wait for multiple blocks to be transferred like we did when attempting to enforce message scheduling in Moxie.

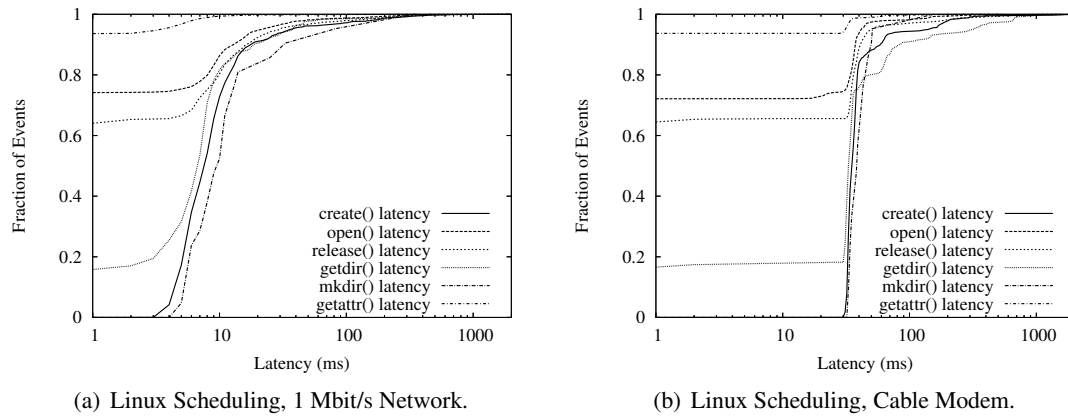


Figure 7.12: **Operation Latency Distribution Using Linux Network Scheduling:** These graphs show the distribution of operation latency in Moxie with network scheduling performed by the Linux traffic shaping tools. Compared to the graphs in Figure 7.10, these curves exhibit a much shorter tail, indicating that high-priority messages are transmitted quickly and not delayed by low-priority transfer messages.

The Moxie prototype is unable to enforce the scheduling policies required by intention updates in the Java Virtual Machine. These experiments, however, demonstrate the potential of intention updates when those policies are strictly enforced and the importance of managing the network schedule at a low level. It also suggests that an user-level implementation of intention updates might benefit from being designed to use the Linux traffic shaping tools for network scheduling.

### 7.5.5 The Cost of Security

In all of the experiments described previously in this section, Moxie has been configured to leave requests unsigned and data unencrypted. We now consider the cost of securing the communications and data. We again execute the Tcl benchmark on Moxie, this time varying the algorithms used to sign requests and encrypt data. The Moxie client is configured to use intention updates and to compute updates using Rabin fingerprint-based delta-encoding. The network between the client and middleware server is configured to restrict bandwidth to 1 Mbit/s without imposing any additional latency.

We measure the overhead of two different public key cryptography technologies: 3072-bit RSA cryptography and 256-bit elliptic curve cryptography (ECC). The key sizes used for these experiments was chosen to provide comparable levels of security. Reduced computation and signature size are key advantages of ECC. We also measure the overhead of encrypting data using the

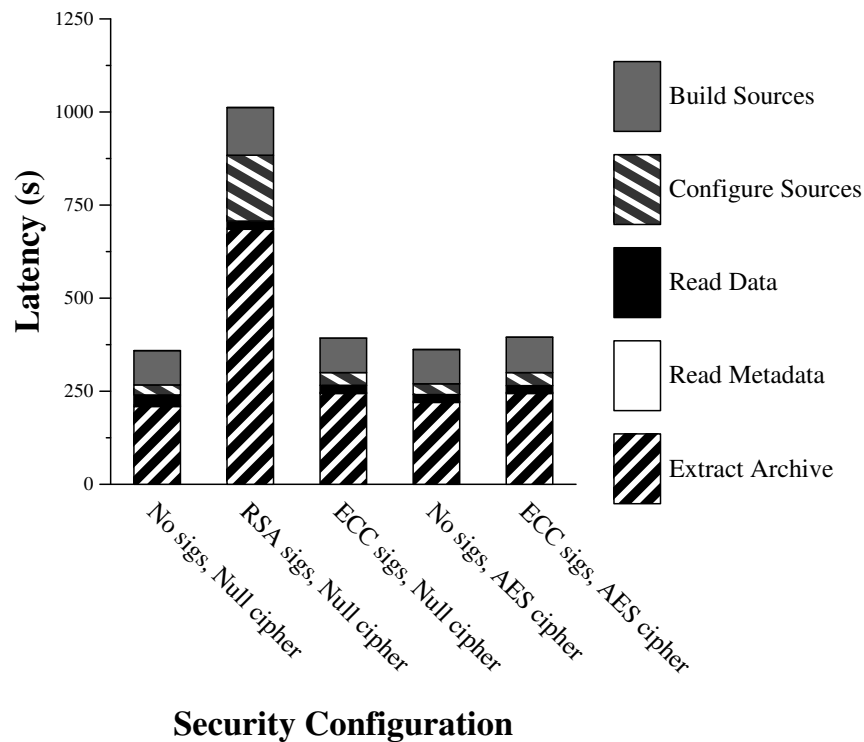


Figure 7.13: **The Cost of Security:** The use of standard public key cryptography algorithms, like RSA, for signing requests can slow Moxie considerably, newer technologies, like elliptic curve cryptography, can provide similar security with only marginal cost. Clients can also protect data privacy by encrypting data with little cost in total runtime.

Advanced Encryption Standard (AES) with 128-bit keys.

Figure 7.13 shows the runtime of the Tcl benchmark on Moxie using differing combinations of signature and encryption technologies. Clearly, changing the signature and encryption algorithms should only affect the phases that write data to the file system. The Tcl benchmark writes data in the first, fourth, and fifth phases, with the first phase being write intensive. RSA cryptography more than doubles the runtime of the benchmark because of the high computational costs of computing RSA signatures with large keys. ECC cryptography slows the client only marginally, demonstrating the advantages of ECC. Encrypting data on the client also has a relatively minor effect on total runtime of the benchmark.

Table 7.3 reports the average latency of the intent in the different configurations. The Tcl benchmark issues 1029 intent operations. Several general trends emerge: RSA signatures are very expensive, ECC signatures have a much lower computational overhead, and encryption adds little delay. Attempting to draw more specific conclusions, however, is difficult. Interference from

Operation	No sigs, Null cipher	RSA sigs, Null cipher	ECC sigs, Null cipher	No sigs, AES cipher	ECC sigs, AES cipher
write-back	40.7	430.1	70.9	50.5	59.5

**Table 7.3: Mean Latency of Write-back Operations with Different Security Configurations:** The cryptographic algorithms used by the Moxie client affect the latency of operations that change file system state. This table reports the latency of file write-back operations. Although general trends can be deduced from these measurements, interference from low-priority data transfer traffic prevents more detailed conclusions.

the background transfer operations obscures the true latency of the intent. This problem can be observed in the “ECC sigs, Null cipher” and “ECC sigs, AES cipher” columns in the table. In the latter configuration, the system is performing more work, but, due to the interference, the former configuration was faster on average. Refer to the discussion in Section 7.3.2 for an explanation of why message priority is not strictly enforced in Moxie.

## 7.6 Interfaces for Intention Updates

Intention updates introduce several new failure modes and options for interacting with storage. The traditional file system API does not provide interfaces to support this interaction. Consequently, for the experiments described in this chapter, we specify the behavior of the system through static configuration files that defined the behavior for an entire mounted file system. In this section, we discuss the new operating modes of intention updates and propose some interface extensions to allow application writers to control interaction with the storage system better.

### 7.6.1 Defining Data Freshness Requirements

The key new scenario that can arise with intention updates is that a client can request data which the server knows exists but does not store. This corresponds to a file for which an intent has been processed but the subsequent data transfer has not yet taken place. In this case, data is visible but not yet durable. The two-phase update process of intention updates exposes a new state—visible but not yet durable—but the traditional file system API cannot respond to that state.

Consider the potential response that a client requesting data that is visible but not durable may wish to make.

- **Fail:** The reader will fail, returning an error code to the application indicating the data is unavailable.

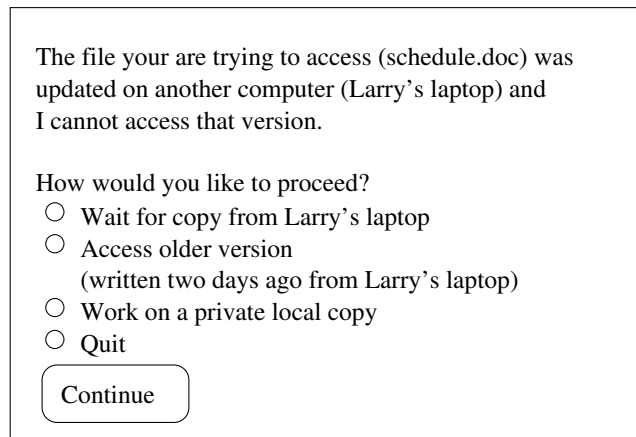
- **Block:** The reader will block until the visible changes become available from the server.
- **Block until deadline:** The reader implements a hybrid approach, combining “Fail” and “Block”. The client waits for the data for a specified time period. If data is not available before the deadline, an error is returned to the application.
- **Use stale version:** The reader will use a version cached locally or an older version that is wholly available at the server.
- **Fetch data from source:** The reader will identify the most recent version from server and attempt to fetch unavailable data from source client. This could be appropriate for devices on the same side of the weak network link.

If a client attempts to update an object, but cannot access the latest version at the server, it may also define one of several approaches.

- **Fail:** The writer will refuse to update a file if it cannot first access the current state of the file from the server.
- **Branch:** The writer will create a private, but related, copy of the data. Although branching can prevent clients from making conflicting updates to a single copy of the file, it can lead to divergent versions on different branches. To control divergence, the application may wish to merge changes back into the main trunk.
- **Overwrite with local version:** The writer simply overwrites the current, unaccessible version with a local copy. This is similar to a “last-writer wins” consistency model.

In the Moxie prototype, we have selected a static policy for dealing with situations when a client attempts to read data that is not yet durable. Currently, Moxie blocks the reader until data is available. It may be desirable to allow applications to define alternative responses to such situations. We describe two different approaches to allow applications to specify the system response at a fine granularity.

A system could support a configuration file that specifies the desired behavior for different collections of file in the file system. The configuration file would be consulted at mount time to initialize a set of policies that would guide how the file system responds to future requests. Users (or applications on behalf of users) could define policies based on directory structures, file types, or custom-defined projects. This approach could be considered similar to Coda’s hoard database [SKK<sup>+</sup>90].



The file your are trying to access (schedule.doc) was updated on another computer (Larry's laptop) and I cannot access that version.

How would you like to proceed?

- ☐ Wait for copy from Larry's laptop
- ☐ Access older version  
(written two days ago from Larry's laptop)
- ☐ Work on a private local copy
- ☐ Quit

Continue

Figure 7.14: **Mock-up of a Graphical Interface for Managing Intents:** In a system with intention updates, a user may attempt to access data that is visible but not yet durable. A graphical interface, such as this, may be helpful in helping users understand this uncommon case. The exact implementation of such an interface should be guided by research in the field of human-computer interaction.

To allow the application to specify the behavior of the file system programmatically, a system could extend the use of flags in the file system's `open()` call. The flags would specify how the system should service the request. The flags could, for example, specify “open latest version or fail” or “open latest version or block” or “open latest available version”. The return values should include information about the version that was opened. This approach would allow the application to change the behavior of the file system on a per-access basis.

The extended file system interface could be used to build a library that would allow users to request certain behavior interactively. Figure 7.14 illustrates a mock-up of how such an interface might look. The interface should help a user understand that a new version of the requested file exists but cannot currently be accessed. It should help the user choose an appropriate course of action to proceed. The actual implementation of such a library and interface should be guided by the conventions and results of researchers in the field of human-computer interaction.

## 7.6.2 Defining Write-back Requirements

Applications may also wish to control how data is buffered at the client and transferred back to the server.

For most applications, simple policies to guide the order of data transfers would likely suffice. The simplest write-back policy is the “first-in, first-out” (FIFO) policy. With FIFO, data is



transferred to the server in the order that it is created. The oldest block of data is always transferred to the server first.

For workloads in which files are frequently overwritten, a “most recent version first” policy would be more appropriate. Under this policy, updates are transferred to the server in a FIFO order unless an update in the transfer queue is overwritten. In that case, the update that was overwritten would be removed from the queue and placed in a buffer, not to be transferred until the transfer queue is empty. This policy works to ensure that the most recent version of each file is available at the server. If a file is overwritten frequently, the transfer manager would probably want to ensure that a complete version of the file is transferred periodically to protect against potential client failure.

Some usage patterns may call for even greater control over data write-back. For example, in a specialized device with control over the entire suite of application, developers may wish to prioritize the transfer of data from different applications. A system could provide such a feature by supporting an additional parameter to the `write()` that defines the relative priority of the data being written.

Although the two-phase update process does provide many benefits, the data buffered at the client awaiting transfer to the server can be considered a liability. Applications may wish to limit that liability by limiting the amount of data that may be buffered at the client. To support this, the file system interface could be extended to support a `limit()` operation that takes as an argument a number that defines that maximum amount of untransferred data to be buffered locally. Attempts to write data that would cause the limit to be exceeded would fail with an appropriate error code like `EFBIG` or `ENOSPC`. To ensure that all data has been flushed to the infrastructure (for example, when putting a device to sleep), the application could issue a `limit(0)` request.

This functionality could alternatively be supported by defining a new resource type to be managed by the `getrlimit()/getrusage()/setrlimit()` interfaces. If the amount of buffered data is to be managed at a per-process, the transfer manager would need to support multiple queues and ensure fairness among the queues.

### 7.6.3 Failure Recovery

Buffering data at clients introduces one other liability: an update may be made visible just before a client permanently fails, thus never completing the data transfer operation. This is similar to the situation discussed above in which a reader attempts to access data that is visible but

not durable. The difference is that, in this scenario, the data will never be available at the server regardless of the amount of time a reader waits.

To recover from this failure, another client would need to use one of the options described above to write a new version of the lost file. In most circumstances, the source for the new version of the file would likely be a previous version of the file.

Detecting when a client has permanently failed, how to present that information to the user, and how to help the user reason about how to restore the file are unsolved problems that require further research.

## Chapter 8

# Future Work and Conclusion

To conclude this thesis, we enumerate several interesting future directions for this research and review the key results presented in this work.

### 8.1 Future Work

The results presented Chapter 7 showed that the techniques proposed in this thesis can improve access to remote storage systems for weakly connected users. That evaluation used a number of short-running benchmarks. An evaluation of long-term usage of such a system would like provide other valuable insights.

As part of a long-term evaluation driven by day-to-day usage of a system like Moxie, we could address a number of questions. How bursty is write traffic and how does it affect the amount of time data is buffered at the client before transfer to the server? How often is data “trapped” on a client device for extended periods of time because the device is turned off or put in a power conservation mode? How frequently does a user attempt to access data that has not yet been transferred to the server?

Results from a long-term evaluation could help drive some of the user interface questions mentioned in Chapter 7. By understanding how often users attempt to access data that is unavailable and how they react, we might be able to begin to develop policies and interfaces to help them manage those situations. One area of computing that might react most positively to such interfaces would be cellular phone-based mobile computing. Many multimedia capture devices, like cameras and audio recorders, and bundled with newer mobile phones. Simultaneously, many new applications and services are being developed for these devices. Finally, many service plans for mobile phones

include per-byte or air-time charges for data transfers. All of these features combine to suggest that the cellular phone platform may be an interesting application domain for the ideas we have discussed.

Finally, Moxie’s implementation of the data transfer manager is relatively primitive. The prototype schedules data transfers in a FIFO manner and can respond to hints from the server to move requested data to the front of the queue. We believe it would be beneficial to study further the design of the data transfer manager. Are there techniques that enable effective network scheduling at the application level? How must the design be altered to support more advanced scheduling policies, like most recent version first? Is it feasible to support application-defined data priorities? How should the data transfer manager optimize the use of the link among multiple, competing applications that are writing back data?

## 8.2 Key Lessons

The experience of working on this thesis has taught the author several important lessons.

At a grand scale, the work for this thesis has highlighted the importance of considering how a system is used by other systems and its role in the entire application stack. Failure to consider the broader perspective may result in systems that are useful only within the rigid boundaries of a single component.

For example, to the cynic, it may appear as though early distributed CAS systems were designed with little regard for how applications might actually use them. Although the block-level interface is simple and elegant when considered from the perspective of the CAS system, it is terribly inefficient and wasteful of client resources. In Chapter 6, we described how the block-level interface also wasted resources within the storage system itself. Through our development of the extent-based API, we showed how considering the requirements of both the clients using the storage system and the CAS system itself can lead to improved designs.

Similarly, our study of write-back strategies and their impact on client-perceived performance showed how limiting the POSIX file system API can be. While it is hard to fault the design of the POSIX API—after all, it was designed in a different era when storage systems had drastically different performance characteristics—our development of intention updates demonstrated how extending the API to differentiate between visibility and durability enabled optimizations in the storage system resulting in better performance for the clients.

In some sense, the lesson of considering a system’s role from a broad perspective is a

reinforcement of the end-to-end argument for system design [SRC84].

The development of intention updates also provided a valuable lesson in balancing the competing interests of the common case and the uncommon case. With regard to data write-back, the common case is that the data written back is only being used by the client that is propagating the changes to the server. The uncommon case is that the data being written back is being actively shared with other clients. In the common case, the user wants the system to respond quickly and cares little for any mechanisms for enforcing consistency that slows it down. In the uncommon case when sharing is present, the clients sharing the data will often be more concerned that data not be corrupted by conflicting writes and the accesses reflect current modifications. With intention updates, we were able to satisfy these competing goals, with very reasonable cost. While certainly it is not always possible to satisfy all competing parties, it was especially satisfying to be able to do so in this case.

### 8.3 Summary

In this thesis, motivated by the emergence of new distributed storage architectures with the potential to deliver managed storage services to new user populations, we have explored how to improve access for weakly connected clients to remote, content-addressable storage systems.

After discussing the social and technological trends that drive our work and defining the system model that we assume throughout our research, we used simulations to identify the issues that weakly connected users face when accessing remote storage systems. These simulations revealed that weak-connectivity can increase the time to execute a sequence of operations in a session, making the system seem sluggish and unresponsive to users. Using telephone modem connections, the penalty can be 10's of minutes. If users also use multiple devices to access data, they face challenges of sharing data between devices. Simulations show that cache management traffic can increase by as much as 40%, and traditional file systems may allow inconsistent accesses in half of all sessions.

Understanding the challenges that weakly connected users face, we examine several techniques to improve the effective bandwidth and latency of the network connection to serve users better. To increase the effective bandwidth of the link without sacrificing data privacy, we developed privacy-preserving delta-encoding. The technique uses LBFS's approach to split a file into blocks and identify blocks shared between files. To apply updates to encrypted data, the client and server share a data format that allows efficient insertion and deletion of variable-sized blocks of ci-

phertext. Although the exact performance improvement depends on the amount of data unchanged between versions of a file, the approach approximates the improvement of other delta-encoding approaches while maintaining data privacy.

To reduce the effective latency of the link, we propose a new write-back method called intention updates. The approach uses a small, synchronous intent message to update metadata at the server quickly. The data is later transferred to the storage system asynchronously. The two-phase update process makes changes visible quickly, with little cost to data durability. It also allows clients to schedule data write-back, prioritizing the transfer of important data.

We also explore how to improve the efficiency of content-addressable systems via aggregation and two-level naming. With two-level naming, the storage system aggregates multiple application-level blocks into extents. While the storage system needs only to manage data at the granularity of the extent, clients can still address blocks individually and update data incrementally. Two-level naming also dramatically decreases the cost of creating certificates to authenticate data stored in the system.

Finally, we present the Moxie prototype storage system for weakly connected clients. Moxie demonstrates how to combine the techniques we explored individually into a single system. Moxie exposed the importance of being able to control the schedule of the network accurately. Even small deviations from the ideal message schedule can quash the benefits of intention updates. Unfortunately, enforcing a precise message order is difficult to do at the application level. Experience with the prototype revealed that the techniques can be combined to improve dramatically access to remote storage for weakly connected users. For example, we measured Moxie to execute a workload similar in format to the Andrew benchmark 75% faster than NFS (version 2).

# Bibliography

- [ABC<sup>+</sup>02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, December 2002.
- [Age06] National Security Agency. The case for elliptic curve cryptography. [http://www.nsa.gov/ia/industry/crypto\\_elliptic\\_curve.cfm](http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm), 2006.
- [Ama06] Amazon.com, Inc. Amazon simple storage service. <http://aws.amazon.com/s3>, 2006.
- [App06] Apple Computer, Inc. Apple .mac service. <http://www.apple.com/dotmac/>, 2006.
- [BHK<sup>+</sup>91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proc. of ACM SOSR*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.
- [Bre00] Eric Brewer. Towards robust distributed systems. Invited talk at Principles of Distributed Computing, July 2000.
- [Bru01] Bruskin Research. Nearly one in four computer users have lost content to blackouts, viruses and hackers according to new national survey. <http://www.iomega.com/about/prreleases/index.html>, February 2001. Survey on data loss.
- [BTC<sup>+</sup>04] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *Proc. of NSDI*, pages 337–350, March 2004.

- [CDB04] Brian Cornell, Peter A. Dinda, and Fabin E. Bustamante. Wayback: A user-level versioning file system for linux. In *Proc. of USENIX Technical Conference, Freenix Track*, pages 19–28, June 2004.
- [CDH<sup>+</sup>06] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubi-  
atowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, San Jose, CA, May 2006.
- [CDRS86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of VLDB*, pages 91–100, August 1986.
- [Chi06] One Laptop Per Child. One laptop per child. <http://www.laptop.org/>, 2006.
- [CN01] Landon P. Cox and Brian D. Noble. Fast reconciliations in fluid replication. In *Proc. of IEEE ICDCS*, pages 449–458, 2001.
- [DGN<sup>+</sup>06] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proc. of NSDI*, May 2006.
- [DJD05] Jennifer Cheeseman Day, Alex Janus, and Jessica Davis. Computer and internet use in the united states: 2003. Technical Report P23-208, US Census Bureau, <http://www.census.gov/>, October 2005.
- [DKK<sup>+</sup>01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, pages 202–215, October 2001.
- [DW98] Johan Danielsson and Assar Westerlund. Arla—a free AFS client. In *Proc. of USENIX Technical Conference, Freenix Track*, June 1998.
- [DZD<sup>+</sup>03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatoicz, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of International Workshop on Peer-to-Peer Systems*, February 2003.
- [FKM00] Kevin Fu, M. Frans Kaashoek, and David Mazieres. Fast and secure distributed read-only file system. In *Proc. of OSDI*, October 2000.



- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of ACM SOSP*, pages 96–108, October 2003.
- [GNA<sup>+</sup>97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proc. of ACM SIGMETRICS*, June 1997.
- [Gra05] Jim Gray. Advice to CACR on research directions/what to do? a research agenda. [http://research.microsoft.com/ Gray/talks/](http://research.microsoft.com/Gray/talks/), February 2005. Presentation about data management challenges.
- [Har93] John H. Hartman. Using the sprite file system traces. University of California Report, May 1993.
- [Har02] Harris Interactive. Americans love their home computers, but do they trust them? <http://ir.imation.com/>, September 2002. Survey on public perception of computer storage.
- [HH95] L.B. Huston and Peter Honeyman. Partially connected operation. In *Proc. of USENIX Symposium on Mobile and Location-Independent Computing*, pages 91–97, April 1995.
- [HKM<sup>+</sup>88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, Boston, Massachusetts, May 2005.
- [HO93] John Hartman and John Ousterhout. The Zebra striped network file system. In *Proc. of ACM SOSP*, pages 29–43, December 1993.
- [HRBSD03] Martin Heusse, Franck Rousseau, Gilles Berger-Sabbatel, and Andrzej Duda. Performance anomaly of 802.11b. In *Proc. of IEEE INFOCOM*, pages 836–843, April 2003.

- [Hub04] Bert Hubert. Linux advanced routing and traffic control HOWTO. <http://lartc.org/>, 2004.
- [KBC<sup>+</sup>00] John Kubiawicz, Davic Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, November 2000.
- [KCN02] Minkyong Kim, Landon P. Cox, and Brian D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proc. of USENIX FAST*, January 2002.
- [KE02] David Kotz and Kobby Essien. Analysis of a campus-wide wireless network. In *Proc. of ACM MobiCom*, pages 107–118, September 2002.
- [Kle75] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. John Wiley & Sons, Inc., 1975.
- [Kle76] Leonard Kleinrock. *Queueing Systems, Volume II: Computer Applications*. John Wiley & Sons, Inc., 1976.
- [KS91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proc. of ACM SOSP*, volume 25, pages 213–225, Pacific Grove, CA, USA, 1991. ACM Press.
- [Lev05] Peter Levart. FUSE-J java bindings for FUSE. <http://sourceforge.net/projects/fuse-j>, 2005.
- [IGA06] Jean loup Gailly and Mark Adler. zlib: A massively spiffy yet delicately unobtrusive compression library. <http://www.zlib.net/>, 2006.
- [LKMS04] Jinyuan Li, Maxwell Krohn, David Mazires, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proc. of OSDI*, pages 121–136, December 2004.
- [LLS99] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proc. of USENIX Technical Conference*, June 1999.
- [LT96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. of ASPLOS*, pages 84–92, Cambridge, MA, October 1996.

- [Mac00] Joshua P. MacDonald. File system support for delta compression. Master's thesis, University of California, Berkeley, 2000.
- [Mal02] Florin Malita. LUFFS (linux userland filesystem). <http://lufs.sourceforge.net/lufs/>, 2002.
- [Maz01] David Mazieres. A toolkit for user-level file systems. In *Proc. of USENIX Technical Conference*, pages 261–274, June 2001.
- [MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proc. of ACM SOSP*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001.
- [Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378, 1988.
- [MES95] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. of ACM SOSP*, pages 143–155, December 1995.
- [Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In *Proc. of CRYPTO*, pages 417–426, 1985.
- [MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, December 2002.
- [MRWHZ04] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proc. of USENIX FAST*, pages 115–128. The USENIX Association, March 2004.
- [MS94] Lily B. Mummert and M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. In *Proc. of Summer USENIX Conference*, pages 279–289, Boston, MA, USA, June 1994.
- [MT85] Sape J. Mullender and Andrew S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proc. of ACM SOSP*, pages 51–62, December 1985.

- [Mye86] Eugene Myers. An  $O(ND)$  difference algorithm and its variations. In *Algorithmica*, pages 251–266, 1986.
- [NCF05] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proc. of ACM SOSP*, volume 39, pages 191–205, New York, NY, USA, December 2005. ACM Press.
- [NWO88a] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.
- [NWO88b] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [OCH<sup>+</sup>85] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. of ACM SOSP*, pages 15–24, Orcas Island, WA, December 1985.
- [PGK88] David Patterson, Garth Gibson, and Randy Katz. The case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM SIGMOD*, Chicago, IL, USA, May 1988.
- [Pon06] Larry Ponemon. U.S. survey: Confidential data at risk. Technical report, Ponemon Institute, LLC, <http://www.ponemon.org/>, August 2006.
- [PST<sup>+</sup>97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of ACM SOSP*, pages 288–301, Saint Malo, France, October 1997.
- [QD02] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proc. of USENIX FAST*, January 2002.
- [Rab81] M.O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.
- [Rab89] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

- [RD01] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSR*, pages 188–201, October 2001.
- [REG<sup>+</sup>03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiawicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, pages 1–14, March 2003.
- [RGK<sup>+</sup>05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiawicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proc. of SIGCOMM*, pages 73–84, New York, NY, USA, August 2005. ACM Press.
- [RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiawicz. Handling churn in a dht. In *Proc. of USENIX Technical Conference*, June 2004.
- [Ris04] Al Riske. Contrarian minds: John gage. <http://research.sun.com/minds/2004-0610/>, 2004.
- [RLA00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proc. of USENIX Technical Conference*, pages 41–54, June 2000.
- [RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [SCJ04] Kevin D. Simler, Steven E. Czerwinski, and Anthony D. Joseph. Analysis of wide area user mobility patterns. In *Proc. of IEEE Workshop on Mobile Computing Systems and Applications*, December 2004.
- [SCR<sup>+</sup>03] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. RFC 3530, April 2003.
- [SDH<sup>+</sup>96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of USENIX Technical Conference*, pages 1–14, January 1996.
- [SEL01] Hal Stern, Mike Eisler, and Ricardo Labiaga. *Managing NFS and NIS, Second Edition*. O’Reilly Media, 2001.

- [SGK<sup>+</sup>85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. of Summer USENIX Conference*, pages 119–130, 1985.
- [SKK<sup>+</sup>90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI*, December 2002.
- [Smi06] Christopher M. Smith. Linux NFS overview, FAQ and HOWTO documents. <http://nfs.sourceforge.net/>, 2006.
- [SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [SSS99] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why does file system prefetching work? In *Proc. of USENIX Technical Conference*, June 1999.
- [Sze04] Miklos Szeredi. FUSE (filesystem in userspace). <http://fuse.sourceforge.net/>, 2004.
- [TAS06] Niraj Tolia, David Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *IEEE Computer*, 39(3):46–52, March 2006.
- [TKS<sup>+</sup>03] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas Bressoud, and Adrian Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proc. of USENIX Technical Conference*, pages 127–140, June 2003.
- [TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, Australian National University, 1996.
- [TTP<sup>+</sup>95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system, December 1995.

- [WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, Banff, Canada, October 2001.
- [WECK07] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiawicz. Antiquity: Exploiting a secure log for wide-area distributed storage. March 2007.
- [XBLG02] Kaixin Xu, Sang Bae, Sungwook Lee, and Mario Gerla. TCP behavior across multi-hop wireless networks and the wired internet. In *Proc. of ACM WoWMoM*, Atlanta, Georgia, USA, September 2002.
- [ZN00] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proc. of USENIX Technical Conference*, pages 55–70, June 2000.