Collaborative Backup for Self-Interested Hosts

by

Landon P. Cox

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2005

Doctoral Committee:

Associate Professor Brian D. Noble, Chair
Professor Jeffrey K. MacKie Mason
Professor Michael P. Wellman
Associate Professor Peter M. Chen

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 3192613

Copyright 2005 by
Cox, Landon P.

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3192613

Copyright 2006 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Landon P. Cox 2005
All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For my family and and friends everywhere.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many people. Each offered ideas, guidance, criticism, and encouragement. I extend my greatest thanks to:

My PhD advisor, Professor Brian Noble. He stayed with me through my early struggles, taught me how to be a researcher, and gave me every possible opportunity to succeed. I am deeply indebted to him.

Pete Chen, Michael Wellman, and Jeffrey MacKie Mason for serving on my committee. I learned a great deal from each while writing my dissertation. They pushed me in new directions and I am a far better researcher for it.

The University of Michigan systems students and faculty: Mark Corner, Minkyong Kim, James Mickens, Sam Shah, Anthony Nicholson, Rich Hankins, Sam King, and Jason Flinn. I looked forward to coming in to work every day because of them.

Mom, Dad, and Hannie. My family has always been there for me, even when I am cranky and tired.

Melissa for her incredible love and sacrifice, often beyond what I deserve. I cannot imagine a more supportive partner.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1. INTRODUCTION 1

1.1 Motivation 1
1.2 Thesis Statement 4
1.3 Overview 4

2. BACKGROUND 8

2.1 Peer-to-Peer Routing 8
2.2 Cryptography 10
2.2.1 Identifying Common Data 10
2.2.2 Naming Chunks 10
2.2.3 Sharing with Confidentiality 12
2.3 Microeconomic Concepts 13

3. PASTICHE ARCHITECTURE 15

3.1 Chunkstore File system 16
3.2 Abstracts: Finding Redundancy 18
3.3 Overlays: Finding a Set of Buddies 19
3.4 Backup Protocol 21
3.5 Restoration 22
3.6 Alternative Designs 23
3.7 Implementation 24
3.8 Evaluation 26
3.8.1 Performance 26

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8.2 Backing Up and Recovering a File System 28
3.8.3 Finding Buddies 29
3.9 Discussion 33

4. BILATERAL, EQUAL EXCHANGE 35

4.1 Disincentives to Contribute 36
4.2 The Bilateral, Equal Exchange Protocol 38
4.3 Transient Failure 41
4.4 Exchange Model and Analysis 46
4.4.1 Problem Constraints 47
4.4.2 Problem Objective 48
4.4.3 Model Equilibria and Analysis 49
4.5 Expected Costs and Behavior 54
4.5.1 Strategic Behavior 54
4.5.2 Monthly Cost of Running Pastiche 55

5. STORAGE CLAIMS 58

5.1 Claim Construction 62
5.2 Forwarding Claims 63
5.3 Forwarding and Reliability 66
5.4 Evaluation 68
5.4.1 Prototype Micro-benchmarks 70
5.4.2 Space Overhead 73
5.4.3 Chain Length Analysis 74
5.4.4 Reliability 76

6. CYCLIC EXCHANGE 79

6.1 Searching for Cyclic Demand 80
6.1.1 Search Efficiency and Flooding 83
6.1.2 Repeatable Searches 84
6.2 Instantiating Cycles 84
6.3 Augmenting Cyclic Allocation 85
6.4 Evaluation 87
6.4.1 Characterizing Cycles 87
6.4.2 Node Availability 89
6.4.3 Node Churn 92

7. RELATED WORK 95

7.1 Approaches to Backup 95
7.2 Strategic Behavior 98

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8. CONCLUSIONS 103

BIBLIOGRAPHY **106**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Table

3.1 Andrew Benchmark 27

3.2 Micro-benchmarks 27

3.3 Backup and Restore 28

4.1 Summary of $\mathcal{S}$ and $\mathcal{N}$ for $2G < P \leq 3G$ and $3G < P \leq 4G$. 51

4.2 Cost formula. 56

5.1 `samsarad` interface. 69

5.2 Simulated space overhead. 74

5.3 Chain length distributions. 74

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure

3.1 Naming and Storing Chunks 16

3.2 Meta-data Chunk Layout 20

3.3 Chunk Skeleton 21

3.4 Varying Abstract Size 30

3.5 Expected Number of Buddies 32

3.6 Coverage-rate Simulation Results 33

4.1 Query response construction. 39

4.2 Expected consecutively missed queries before object loss. 43

4.3 Required bandwidth to create replicas. 45

4.4 Optimal values of $\mathcal{S}$ and $\mathcal{N}$ for $P = 200$ and $G = 10$. 52

4.5 Schedule of strategies for $R = 5$, $G = 3$, $B = 2$, and $\mathcal{N} = 1$. 53

4.6 Cost of running Pastiche. 57

5.1 Storage claims 60

5.2 Forwarding claims. 64

5.3 Failure in dependency chains. 66

5.4 Failure in dependency cycles. 67

5.5 Data transfer performance. 70

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.6 Query performance. 71

5.7 Query processor and disk load. 72

5.8 Chain length distributions for forwarding claims. 75

5.9 Reliability of data with capped chain lengths. 76

5.10 Reliability of data under various storage utilizations. 78

6.1 Example Search 82

6.2 Neighborhood Size and Cycle Membership. 88

6.3 Availability Distributions. 90

6.4 Effect of Node Availability on Recovery Rate. 91

6.5 Effect of Churn and Availability on Recovery Rate. 93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

INTRODUCTION

The shadow of data loss looms over PC users. An online survey of nearly 600 users by Harris Interactive found that 85% of respondents fear losing data [59]. Backup prevents data loss by storing copies at multiple locations. It must keep data safe from both independent hardware failures such as disk drive failures and larger-scale, correlated failures due to catastrophes like power surges, floods, and natural disasters.

Unfortunately, current approaches to backup are inadequate for many users because of their prohibitive cost and inconvenience. Large-scale solutions require aggregation of substantial demand to justify the costs of managing a large, centralized repository. Small-scale solutions require significant administrative effort by the end user.

1.1 Motivation

Users could pay someone to manage their backup, as most organizations do. Unfortunately, paying for backup is expensive. The University of Wisconsin campus computing service charges $4,700 to backup 32GB of data at one onsite and one off-site location for one year [111] [1]. Connected TLM, an online backup service, offers to backup at most 4 GB—but covers neither applications nor the operating system—for $15 per month [27]. Most users are unwilling to spend hundreds, much less thousands, of dollars per year on backup.

[1]This assumes no data compression, three revisions per file, six months of file retention, as well as zero monthly data growth and deletion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Users are equally unwilling to backup their data themselves because of the time, patience, and technical expertise required. Harris Interactive found that only one in four PC users performs backup "frequently", 36% do so "sometimes," 22% do so "seldomly," and 9% do "not know how" [59]. Iomega found that 69% of users backup their home PC less than once a month or never [30].

Unsurprisingly, users routinely experience data loss. A study by Boston Computing found that 6% of all PCs will experience data loss in any given year, that 31% of PC users have lost all of their files at some point, and that 57% of PC users have lost an electronic file they thought they had backed up [29].

Even the 25% of users who frequently backup their data spend hours or even days restoring their PC to a pre-failure state. Users commonly only backup collocated personal data, such as the contents of a home directory in UNIX or the `My Documents` folder in Windows. Application and operating system state must be recreated from installation CDs and online sources.

This is a significant oversight. An anecdotal account estimated that reinstalling the Windows XP operating system can take at least five hours and nearly 150 steps [40]. One can reasonably expect to spend a similar amount of time reinstalling a system's many applications. Locating the many online and offline sources from which applications and the operating system were installed, plus the patches and additional packages used throughout the lifetime of the disk, can waste hours of productivity.

Luckily, users can take advantage of two trends to gain a low-cost, convenient backup service. First, hard-disk capacity remains far ahead of users' storage needs, creating a pool of excess storage capacity. A multi-year study of nearly 10,000 disks by Microsoft Research, including 1998 and 2000 to 2004, found that company workstation hard-disks were 40% to 50% empty on average each year [15]. Another study of home-machines in 2002 found that disks were 70% empty on average [15].

Furthermore, access to broadband Internet connectivity is on the rise. Jupiter Research estimates that half of all online American households will use a DSL or cable modem connection in 2005 and that this percentage will rise to 78% by 2010 [9]. As with storage, PC users' network capacity is largely unused. As we will show in Section 4.5.2, even

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

demanding file sharing users utilize only 10% of a cable modem's upstream capacity and under 0.1% of its downstream capacity.

This spare capacity creates the opportunity to build an alternative backup service for users unwilling to either pay for a conventional service or manage backup themselves. The excess storage and bandwidth can be used to form a peer-to-peer storage collective for backup over the Internet. This approach has two primary advantages.

First, the cost to maintain the collective is less than the cost of maintaining the data centers used by conventional services. Administration [78], cooling [63], and power[23], dominate data center costs.

Peer-to-peer storage's greatest savings stem from its lack of administrative costs. The collective is self-organizing and uses PC owners to keep their machines running. IBM reports that hardware and software administration accounts for approximately two-thirds of the total cost of owning a commercial storage system, while actual hardware accounts for less than 10% [78].

It is difficult to compare the power and cooling costs of each approach with much precision. At the very least, the sum of each individual's cost to power and cool their PC should be no more than a data center's and may be less. Users are not expected to run their machines all the time, while data center machines often draw power continuously. This may be changing, however, as more intelligent data center power-management schemes have been introduced in recent years [23, 63]. Also, the aggregate cost to cool a distributed storage network may be less than the cost to cool a data center since peer machines are not collocated and data centers contain hundreds or thousands of machines within single space.

The second major advantage of a peer-to-peer approach is that storing data in the collective is convenient. There is no extra hardware such as a second hard disk or an archive of removable media to manage. Members are only required to choose and remember a password. After users have chosen their password, the system software can manage their backup state and recover from failure automatically.

In exchange for lower cost and greater convenience, this approach to backup must contend with storing on unreliable, untrusted, and self-interested hosts. Without adminis-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tration, machines will likely be less reliable than in a data center. For example, hosts are free to come and go as they please, which could jeopardize reliability. To ensure that data is available with very high probability at all times, users can create replicas at multiple hosts. Also, because hosts are untrusted, the service must protect users' privacy. Encryption helps, but it must be applied carefully and efficiently.

Dealing with self-interested hosts is the greatest challenge of all. The cost of sharing excess storage and bandwidth is low, but non-zero. For example, users may be holding on to this unused storage for insurance against future needs. Thus, it is in every host's interest to contribute as little as possible. If many hosts consume more resources than they contribute, the collective may collapse. To encourage contribution, we can treat the collective as a barter economy in which hosts trade local storage for remote storage. The rules regulating this market must be carefully construed to eliminate abuse without over-constraining resource allocation.

1.2 Thesis Statement

In response to the cost and inconvenience of current approaches to backup, we set out to demonstrate the following thesis statement:

> **It is feasible to provide a convenient, low-cost backup service using unreliable, untrusted, and self-interested storage hosts.**

1.3 Overview

To validate this thesis, we present the design and implementation of the *Pastiche* backup service. This dissertation is organized as follows:

Chapter 2 presents the underlying technologies that enable Pastiche. Chapter 3 describes the basic Pastiche architecture and how untrusted, unreliable hosts shape its design. Users must be able to recover, even though hosts will come and go without warning. Because of this, Pastiche replicates backup state at multiple sites so that recovery is always possible with very high probability. Excessive replication can strain system resources. If

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

disks are 50% or 70% empty, the collective will support replication factors of only one or two.

We must also take care to conserve disk space and network bandwidth. Thus, Pastiche members reduce their burden by biasing in favor of sites that already have much of their backup state. This redundant data is composed of operating system, application, and media files. By identifying hosts with significant overlap, users only need to ship the small fraction of data that is unique to them.

To evaluate how well Pastiche can identify common inter-host data, we compared the content of seventeen graduate student and faculty workstations in the EECS department at the University of Michigan. We found that machines with common installations can expect to find 30% to 70% overlap with multiple hosts. Furthermore, our approach to identifying inter-host data can also find redundancy within a single file system. The degree of local redundancy varied between 20% and 50% of the total, which was similar to previous results [79]. If we assume a network with 60% empty disks (between the observed emptiness of home and office disks), 30% local redundancy, and 50% overlap between hosts, Pastiche can support a replication factor of over five.

In addition to addressing users' need for backup, Pastiche provides a platform for exploring broader problems. In particular, Pastiche shares the challenge of self-interested hosts with other collaborative systems. Hosts in such systems attempt to simultaneously maximize their benefit and minimize their cost. This can lead to to rampant *free-loading*, where users contribute nothing, and, ultimately, system collapse.

Chapter 4, Chapter 5, and Chapter 6 introduce three approaches to stopping free-loaders. Each chapter addresses the shortcoming of the previous chapters while building on their advantages.

Chapter 4 proposes to eliminate free-loaders through a bilateral, equal storage exchange (BEE) protocol. Hosts are only allowed to store data on those for whom they are storing an equal amount in return. Once data has been swapped, hosts query one another to ensure that theirs is intact. If a host fails a query, its data is discarded in retaliation. This simple protocol ensures that all hosts contribute as much storage as they consume. More important, it provides a strong incentive to contribute storage since discarding another

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

host's data will result in the loss of one's own.

To evaluate BEE, we created a formal model of the protocol. Through analysis of the model, we are able to describe the conditions under which a rational host obeys the protocol. In some environments, free-loading is optimal. However, under realistic storage and bandwidth prices, our model predicts that users will contribute to Pastiche as long as others obey BEE. These prices also allow us to give a conservative estimate of the monthly cost of using Pastiche. We found that users with up to 24GB of unique data can run Pastiche for less than one dollar per month.

While BEE provides a strong incentive to contribute storage, it is, by itself, inadequate. First, trades between hosts with asymmetric demand are problematic. If Host A needs to store 1GB and Host B needs to store 40GB, it is unclear how they should negotiate terms that are satisfactory to both. Second, allocating storage exclusively through bilateral exchanges severely over-constrains the system because they require a *double coincidence of wants*. Trades are only possible between mutually interested parties, which may be difficult to locate in a network as large as the Internet. Ideally, allocation would be *flexible* enough to allow nodes to use storage on any preferred host.

Chapter 5 augments BEE with *storage claims*. A storage claim is an incompressible, "junk" storage placeholder that is easy to compute for its owner and must be stored by the host responsible for it. Claims allow Pastiche to manufacture BEE when it does not arise naturally. Furthermore, claim *forwarding* establishes transitive arrangements that eliminate the need for direct, reciprocal interest between hosts. Through simulation we show that forwarding enforces equal consumption and contribution while enabling trades between arbitrary nodes.

Unfortunately, claims are not a perfect solution. In the worst case, they can double the global storage burden. Forwarding eases this burden, but it creates dependency chains along the forwarding path. These chains are vulnerable to cascading failures if data is discarded. However, if a claim is forwarded back to its owner to create a cycle, data can be rerouted around failures. Chapter 6 shows how Pastiche avoids the overhead of claims and the vulnerability of dependency chains through *cyclic exchange*.

Cyclic exchange provides flexible storage allocation with low network and storage

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

overhead. Under cyclic exchange, hosts construct a distributed *demand graph* based on their preferred replica sites. Storage is then allocated along cycles in the graph, creating data dependencies between adjacent edges. Within a cycle, contribution and consumption are equal. Furthermore, because the graph is created from preferences, hosts can store their data exactly where they want. We evaluated cyclic exchange through simulation and found that it is robust in the face of low host availability and observed rates of churn.

Chapter 7 provides an overview of backup options available to users and other work similar to the techniques used in Pastiche. We evaluated each backup option using four criteria: cost, convenience, local safety, and catastrophic safety. Cost is how many dollars a user spends to backup her data with a particular approach. Convenience measures the effort required to backup data. Local safety measures whether or not data can be recovered from a local disk failure and catastrophic safety measures whether or not data can be recovered from a local catastrophe. Finally, Chapter 8 gives our concluding thoughts on Pastiche.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

BACKGROUND

This chapter aims to give Pastiche a technological and conceptual context. Pastiche depends on several enabling technologies. The first is the Pastry peer-to-peer overlay network, a scalable, self-organizing, routing and object location infrastructure [97]. Pastiche also uses several cryptographic techniques, such as Rabin-fingerprinting [74, 79], cryptographic hash functions [82], and convergent encryption [17]. Content-based indexing finds common data across different files. Cryptographic hash functions efficiently and uniquely summarize an object's content. Convergent encryption allows sharing without compromising privacy.

Pastiche uses several concepts from microeconomics to reason about self-interested hosts. In particular, it is important to understand the *opportunity costs* incurred by users and how *barter* can transform them from a cost of production to a cost of consumption.

2.1 Peer-to-Peer Routing

Pastiche eschews the use of a centralized authority to manage backup sites. Such an authority would be a single point of control, limiting scalability and increasing expense. Instead, Pastiche relies on Pastry, a scalable, self-organizing, routing and object location overlay for peer-to-peer applications. There are many other overlays with properties similar to Pastry's, such as CAN [94], Chord [105], and Tapestry [116]. However, we found that Pastry's routing logic best fit our needs.

Each Pastry node is named by a *nodeId*; the set of all nodeId's are expected to be uniformly distributed in the nodeId space. Throughout the remainder of this dissertation,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we will use the terms "host" and "node" interchangeably. Any two Pastry nodes have some way of measuring their *proximity* to one another. Typically, this metric captures some notion of network performance with better connected nodes being "closer" to one another.

Each node N maintains three sets of state: a *leaf set*, a *neighborhood set*, and a *routing table*. The leaf set consists of L nodes; $L/2$ are those with the closest numerically smaller nodeIds, and $L/2$ are the closest larger ones. The neighborhood set of M nodes contains those closest to N according to the proximity metric.

The routing table supports *prefix routing*. There is one row per hexadecimal digit in the nodeId space. The first row contains a list of nodes whose nodeIds differ from the current node's in the first digit; there is one entry for each possible digit value. The second row holds a list of nodes whose first digit is the same as the current node's, but whose second digit differs. To route to an arbitrary destination, a packet is forwarded to the node with a matching prefix that is at least one digit longer than that of the current node. If such a node is not known, the packet is forwarded to a node with an identical prefix, but that is numerically closer to the destination in nodeId space. This process continues until the destination node appears in the leaf set, after which it is delivered directly. The expected number of routing steps is $\log N$, where N is the number of nodes.

Many positions in the routing table can be satisfied by more than one node. When given a choice, Pastry records the closest node according to the proximity metric. As a result, the nodes in a routing table sharing a shorter prefix will tend to be nearby since there are many such nodes. However, any particular node is likely to be far.

Pastry is self-organizing; nodes can come and go at will. To maintain Pastry's locality properties, a new node must join with one that is nearby according to the proximity metric. Pastry provides a *seed discovery protocol* that finds such a node given an arbitrary starting point [21]. Pastiche uses two separate Pastry overlay networks with two different cost metrics, but uses them only during replica site discovery. Once a node has identified its backup set, all further traffic is routed directly via IP.

Pastiche adds two mechanisms to Pastry. The first is a technique called the *lighthouse sweep* that guarantees that distinct Pastry nodes are queried during replica site discovery.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The second is a distance metric based on file system contents; this is used to find sites for machines with rare installations.

2.2 Cryptography

Pastiche utilizes a number of cryptographic algorithms, including Rabin fingerprinting, cryptographic hash functions, and convergent encryption. These techniques provide independent, untrusted Pastiche users with a common way to name and represent data.

2.2.1 Identifying Common Data

To minimize storage overhead, Pastiche must find redundant data across versions of files, files in a system, and files on different machines. Rsync [108] and Tivoli [2] employ schemes for finding common subsets in two versions of (presumably) the same file. However, these techniques cannot easily capture more general forms of sharing.

The challenge is to find sharing—and hence structure— across seemingly unrelated files *without* knowing the underlying structure. Content-based indexing accomplishes this by identifying boundary regions, called *anchors* [74], using Rabin fingerprints [93]. A fingerprint is computed for each overlapping k-byte substring in a file. If the low-order bits of a fingerprint match a predetermined value, that offset is marked as an anchor. The algorithm only depends on the content of the file, so any pair of hosts with the same file will always compute the same anchors. Anchors divide files into *chunks*.

Since anchors are purely content-driven, editing operations only change the chunks they touch, even if they change offsets. This can result in substantial storage savings. For example, LBFS reports that the build tree of emacs 20.7 is nearly 40% redundant and that the `/usr/local` directory is nearly 20% redundant. Content-based indexing found that nearly half of the data on my laptop (3.0 GB out of 6.3 GB) was redundant.

2.2.2 Naming Chunks

As with LBFS [79], Pastiche names each chunk by computing a *cryptographic hash* of its contents. Cryptographic hash functions, $H()$, have two important properties. First, they map a very large set of variable-length—but possibly bounded—inputs to a smaller

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

set of fixed-length outputs. Second, they are *collision-resistant*; given some input x it is computationally infeasible to find some input y such that $y \neq x$ but $H(x) = H(y)$. Many such hash functions exist, including MD5 [96] and SHA-1 [82]. Pastiche and LBFS both use SHA-1.

Cryptographic hash functions have been used in a number of security-conscious settings, including integrity checking [109], digital signatures [80], and digital time-stamping [53]. More recently, hash functions have been used in applications beyond security. For example, they have been used as hints to identify similar files [74] and web pages [19]. They have also been used to generate globally-meaningful, *self-verifying* names for objects in a fully decentralized way. Names are self-verifying because any change to the content of the object would result in a different name.

The probability that two different chunks will hash to the same value is much lower than other sources of error in computer systems. The SHA-1 hash produces a 160-bit output, meaning that the probability of random inputs not colliding is approximately $1 - 2^{-160}$ or 48 "nines." By comparison, TCP provides only 8 or 9 nines of assurance that a bit-flip will be detected [106]. This allows systems to assume that chunks that hash to the same value are in fact the same chunk, and Pastiche adopts this custom. These chunks form the basis of on-disk file structures to easily share data between hosts.

The safety of naming by cryptographic hash has been questioned within the software systems community [55]. Recent theoretical work has also shown that MD5, which is similar in technique to the SHA-1 function used by Pastiche, can be manipulated to produce collisions [114]. While collisions have always been known to theoretically exists, actually manufacturing them was believed to be infeasible. Because the SHA-1 and MD5 algorithms share much of the underlying structure exploited by Wang, et al, it is not unreasonable to expect SHA-1 to eventually be broken.

Even in light of Wang's results, Pastiche's use SHA-1 appears to be safe. This is due to the difference between a *collision attack* and a *pre-image attack.* A collision attack produces two inputs that hash to the same value. For example, an attacker can create x and y such that $H(x) = H(y)$. Wang's work enables this kind of attack.

A pre-image attack is when the attacker, given a hash value, can find inputs that hash

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to that value. This is a much more serious attack and could undermine the self-verifying nature of naming by hash. For example, say that there is a file named $H(x)$ with content x. If an attacker can construct $y \neq x$ such that $H(y) = H(x)$, it can overwrite file $H(x)$ with y. Subsequently verifying the content of the file by "correctly" computing $H(y) = H(x)$ is incorrect, because $x \neq y$. Pastiche is vulnerable to this attack, but fortunately, Wang's work does not enable it.

2.2.3 Sharing with Confidentiality

A well-chosen replica site has much of a Pastiche node's data, even before the first backup. However, Pastiche must still guarantee the confidentiality and integrity of its participants' data using *cryptography.* There are two cryptographic primitives: *encryption* and *decryption.* Encryption takes *plaintext* data, d, and an *encryption key*, K_e, and produces an obfuscation of d, denoted $\{d\}_{K_e}$. Decryption takes the obfuscated data, $\{d\}_{K_e}$, and a *decryption key*, K_d, and returns the original plaintext, d.

Under *symmetric encryption*, $K_e = K_d$. That is, a single key is used to both encrypt and decrypt. *Asymmetric encryption* requires the encryption and decryption keys to be different. One key, called the *public key* is known to all users, while a second *private key* is kept secret by its owner [38]. This allows user A to encrypt data with user B's public key to create obfuscated data that can only be decrypted by B's private key. In many systems, public keys act as unique identifiers [103], like a Pastry nodeId, and managed by a trusted *public key infrastructure* (PKI).

One important decision we made in designing Pastiche was not relying on unique public keys or anything else to provide strong identities. Users are expected to create and manage their own unique nodeId, for example, by computing the SHA-1 hash of their email address. This decision was primarily motivated by cost. To support strong identities users must pay a PKI or something similar to manage them, which can be quite expensive. Verisign's cheapest SSL certificate costs $795 over three years.[1] This alone is more than many users are willing to pay for backup.

Thus, Pastiche users cannot rely on a trusted third party to provide them with encryp-

[1]Quoted from `www.versign.com` in July, 2005.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tion keys. Unfortunately, if each client chooses their own cryptographic keys, chunks with identical content will be represented differently, precluding sharing. The Farsite file system solves this problem with *convergent encryption* [17]. Under convergent encryption, each file is encrypted by a symmetric key derived from the file's contents. Farsite then encrypts the file key with a *key-encrypting key*, known only to the client; this key is stored with the file. As a file is shared by more clients, it gains new encrypted keys; each client shares the single encrypted file.

Pastiche applies convergent encryption to all on-disk chunks. If a Pastiche node backs up a new chunk not already stored on a peer, the peer cannot discover its contents after shipment. However, if the peer has that chunk, it knows that the node also stores that data. Pastiche allows this small information leak in exchange for increased storage efficiency.

2.3 Microeconomic Concepts

We can use concepts from microeconomics to frame the discussion of self-interested Pastiche users. One way of thinking about Pastiche is that each user produces a unique good—access to blocks of its excess storage. Users also want to consume a subset of goods in the network—the set of hosts where they want to store their backup state. An *allocation* maps a set of produced goods to the users that consume them. *Rational* users attempt to force allocations in which their own benefit is maximized. This occurs when they consume their preferred set of goods at minimal personal cost.

Users incur some cost to consume goods, namely the bandwidth required to upload data. More important, users also incur costs during production. Because users expend excess resources, the costs of acquiring storage and network bandwidth to contribute are *sunk*. A cost is sunk if it has already been incurred, as when a disk or network connection has already been purchased.

This does not mean that the cost of production is zero. Users still incur *opportunity costs*. The opportunity cost of an action is the cost of not taking a mutually exclusive, alternative action. For example, the opportunity cost of staying at home includes the cost of missing a friend on the street multiplied by the probability of such an encounter.

In Pastiche, there are two opportunity costs involved in "producing" access to a block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of excess storage. The first is the cost of network bandwidth to download a block times the likelihood that downloading the block will interfere with normal network traffic. Because bandwidth is not infinite, interference is possible, though it may be low.

The second opportunity cost is the cost of allowing another user to read and write disk blocks. Contributing storage costs disk bandwidth that can interfere with the owner's disk IO. In addition to direct interference, extraneous disk traffic can pollute on-disk and in-memory caches and degrade system performance even if it occurs during "off-peak" usage.

Rational users eliminate these opportunity costs by ceasing production. If many users behave this way, the collective will collapse. To avoid this fate, Pastiche forces users to the incur the opportunity costs of production by transforming them into a cost of consumption. Pastiche accomplishes this by allocating storage through a *barter* economy, in which goods are traded for one another. Barter makes production a precondition to consumption. As long as the total cost of production is less than the benefit of using Pastiche, users' incentive will be properly aligned.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

PASTICHE ARCHITECTURE

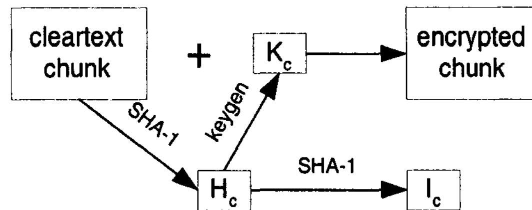
Pastiche nodes form an untrusted collective of machines that provide mutual backup services. Because individual machines may come and go [17], each Pastiche node must replicate its archival data on more than one peer. Most of these replicas are placed nearby to reduce cross-Internet traffic and minimize restore time, though at least one replica must be elsewhere to guard against catastrophe. With no effort on the part of the user and modest additional disk space, backups are provided automatically. Pastiche is primarily aimed at end-user machines, but it can be used for back-end repositories with some care.

With disks between 40% and 70% empty on average, Pastiche cannot afford to keep duplicate copies of data on each replica. Luckily, much of the data on a given machine is not unique. Aside from personal photographs, most users' data is imported from a remote source rather than produced locally. Operating system and application files are almost always copied from another source, as are large music and video archives. Furthermore, most data will be shared widely. The default installation of Office 2000 Professional requires 217 MB; it is nearly ubiquitous and different installations are largely the same. Randomly grouping disparate file systems and coalescing duplicate files produces significant savings [16]. Pastiche identifies systems with overlap to increase this savings.

Even with these building blocks, Pastiche still faces a number of challenges. How can nodes discover *backup buddies* with substantial overlap without a centralized directory? How can nodes reuse their own on-disk state to backup others? How can nodes restore files—or an entire machine—without administrative intervention?

Pastiche computes a small *abstract* of a file system's content that potential backup bud-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure depicts how chunks are stored and named. A cleartext chunk is hashed, producing its handle. The handle is used for key generation, and hashed again to produce the chunkId. The chunk is stored encrypted by the key and named by the chunkId.

Figure 3.1: Naming and Storing Chunks

dies can inspect to approximate overlap. Pastiche is able to limit the size of the abstract by taking advantage of the fact that arbitrary, small pieces of larger logical entities are almost always unique and can, therefore, stand for the whole. This allows machines with common installations to find suitable buddies with very little effort. Machines with uncommon installations may need to use a Pastry overlay with a new routing metric, *coverage rate.*

Because sharing is supported at a sub-file granularity, Pastiche provides a new file system, *chunkstore*. Chunkstore stores all data—the host's as well backup state—in the units of sharing, without compromising the performance of common-case workloads.

Archive state is described by a *skeleton* tree of meta-data. The root of this tree can be recovered from the Pastry overlay with only the name and passphrase of the machine to be restored. Entire file systems can be restored as easily as a single file.

An examination of file system data shows that abstracts of a few hundred bytes effectively discriminate between candidate buddies. Simulations show that Pastiche nodes with common installations can easily find others with good overlap. The chunkstore file system induces overhead of 7.4% on a modified Andrew Benchmark, despite its unoptimized layout.

3.1 Chunkstore File system

Pastiche data is stored on disk as chunks. Chunk boundaries are determined by content-based indexing, and encrypted with convergent encryption. Chunks carry *owner lists*, which name the set of nodes with an interest in a chunk. Chunks may be stored on a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machine's disk for that machine, a backup client, or both. Data chunks are immutable, and each chunk persists until no node holds a reference to it. Pastiche ensures that only rightful owners are capable of removing a reference to (and possibly deleting) a chunk.

When a newly written file is closed, it is scheduled for chunking. Each chunk c is hashed; the result is called the chunk's *handle*, H_c. Each handle is used to generate a symmetric encryption key, K_c, for its chunk. The handle is hashed again to determine the public chunkId, I_c, of the chunk. Each chunk is stored on disk encrypted by K_c and named by I_c. This process is illustrated in Figure 3.1.

Before writing a chunk to disk, Pastiche first checks to see if it already exists. If so, the local host is added to the owner list if necessary, and the local reference count is incremented. Otherwise, the chunk is encrypted, a message authentication code [81] is appended, and the chunk is written out to disk with a reference count of one for the local owner.

Chunking and writing to disk are deferred to avoid needless overhead for files with short lifetimes [113], at the cost of slightly weaker persistence guarantees. The list of chunkIds that describes a node's current file system is called its *signature*.

Data chunks are immutable. When a file is overwritten, its set of constituent chunks may change. Any chunks no longer part of the file have their local owner's reference count decremented; if the reference count drops to zero, the local owner is removed. If the owner list becomes empty, the chunk's storage is reclaimed. File deletion is handled similarly.

The meta-data for a file contains the list of handles for the chunks comprising that file, plus the usual contents: ownership, permissions, creation and modification times, etc. The handles in this list are used to derive the decryption key and chunkId for each constituent chunk.

Meta-data chunks are encrypted to protect the handle values and hence cryptographic keys. This differs slightly from Farsite's use of convergent encryption. Farsite stores keys with data, encrypting each derived key with a key private to the writing host. Pastiche stores handles, and hence keys, in the meta-data blocks.

Unlike data, meta-data is is not chunked and is mutable. Pastiche does not chunk meta-data because it is typically small and unlikely to be shared. Meta-data is mutable to avoid

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cascading writes. Each write to a file changes its constituent chunkIds. If meta-data were immutable, Pastiche would have to create a new meta-data chunk with a new name for every update. This new name would have to be added to the enclosing directory, which would also change, and so on to the file system root. Instead, the H_c, K_c, and I_c for a file's meta-data are computed only at creation time, and are re-used thereafter.

The meta-data object corresponding to a file system root is treated specially: its H_c is generated by a host-specific passphrase. As Section 3.5 explains, this passphrase plus the machine's name is all that is required to restore a machine from scratch.

A chunk that is part of another node's backup state includes that nodeId in its owner list. Remote hosts supply a public key with their backup storage requests. Requests to remove references must be signed by the corresponding secret key, otherwise those requests are rejected. This prevents third-party deletions, though it does not prevent the buddy from dropping chunks of its own accord.

Storing files directly as chunks simplifies a number of Pastiche's tasks and imposes modest performance costs. It simplifies the implementation of chunk sharing, convergent encryption, and backup/restore. Without chunkstore, Pastiche would have to maintain a persistent index consistent with on-disk files. This is the approach taken by LBFS [79].

The index would have to be consulted during backup and restore, and complicates garbage collection of chunks retired during snapshot. Furthermore, convergent encryption requires that each chunk be encrypted separately, complicating a contiguous layout. The only alternative would be to detect sharing only at the file level, with a corresponding increase in storage costs for backup.

3.2 Abstracts: Finding Redundancy

Much of the long-lived data on a machine is written once and then never overwritten. Observations of file type [42] and volume ownership [104] suggest that the amount of data written thereafter will be small. In other words, the signature of a node is not likely to change much over time. Therefore, if all data had to be shipped to a backup site, the initial backup of a freshly installed machine is likely be the most expensive.

An ideal backup buddy for a newly-installed Pastiche node is one that holds a superset

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the new machine's data; machines with more complete coverage are preferred to those with less. One simple way to find such nodes is to ship the full signature of the new node to candidate buddies, and have them report degree of overlap.

Unfortunately, signatures are large: 20 bytes per chunk. Expected chunk size is a function of how anchors are selected. In our implementation, this size is 16 KB, so signatures are expected to cost about 1.3 MB per GB of stored data. If this cost were paid only once, it might be acceptable. However, a node's buddy set can change over time as buddies are found to be unreliable or as degrees of overlap change.

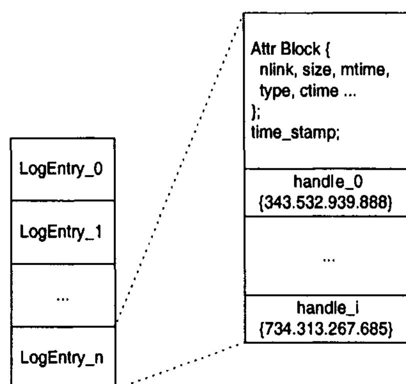
Rather than send a full signature, Pastiche nodes send a small, random subset of their signatures called an *abstract*. This is motivated by the following observation: most data on disk belongs to files that are part of a larger logical entity. For example, a Linux hacker with the kernel source tree has largely the same source tree as others working on the same version. Any machine holding even a small number of random chunks in common with this source tree is likely to hold most of them. Preliminary experiments show that tens of chunkIds—a few hundred bytes—are enough to distinguish good matches from bad ones. This size is similar to that reported by Border for individual web objects [20].

3.3 Overlays: Finding a Set of Buddies

All of a node's buddies should have substantial overlap with it to reduce storage and bandwidth overhead. In addition, most buddies should be nearby to reduce global network load and improve restore performance. However, at least one buddy must be located elsewhere to provide geographic diversity. As a rule of thumb, each Pastiche node maintains five buddies.

Pastiche uses two Pastry overlays to facilitate buddy discovery. One is a standard Pastry overlay organized by network proximity. The other is organized by file system overlap. Every Pastiche node joins a Pastry overlay organized by network distance. Its nodeId is a hash of the machine's fully-qualified domain name. Once it has joined, the new node picks a random nodeId and routes a discovery request to it. The discovery request contains the new node's abstract. Each node encountered on the route computes its *coverage*–the fraction of chunks in the abstract stored locally–and returns it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure depicts how a meta-data chunk is stored on disk. The chunk is stored as a log of file states, where each entry in the log represents the state of the file after the update. Entries are comprised of an attribute block, a time stamp, and a list of constituent chunk handles.

Figure 3.2: Meta-data Chunk Layout

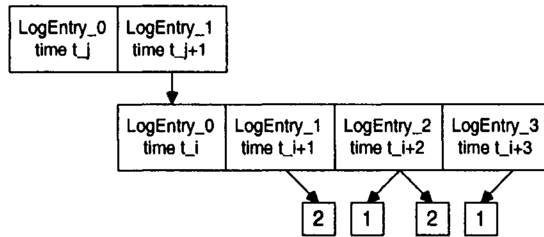
If the initial probe does not generate a sufficient candidate set, the probe process is repeated. Subsequent probes are generated by varying the first digit of the original nodeId. Since Pastry uses prefix routing, each probe will generate sets of candidates disjoint from those already examined. We call this rotating probe a *lighthouse sweep*.

Nodes with common installations should find a sufficient candidate set easily. However, nodes with rare installations will have more difficulty. Nodes that do not find an adequate set during a lighthouse sweep join a second overlay, called the *coverage-rate* overlay. This overlay uses file system overlap rather than network hops as the distance metric. The new node chooses backup buddies from its Pastry neighbor set—the set of nodes encountered during join with the best coverage available.

The use of coverage rate as a distance metric has interesting implications for Pastry. Like network distance, coverage rate does not obey the triangle inequality. Unlike network distance, coverage rate is not symmetric; if A holds all of B's files, the converse is probably not true. This means that an individual node must build its routing state based on the correct perspective. Likewise, the seeding algorithm must be supplied with the new node's abstract, so that it can compute coverage from the correct point of view.

It is possible for a node to habitually under- or over-report its coverage. If it under-reports, it can avoid being selected as a buddy. If it over-reports, it can attract unsuspecting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure depicts a small skeleton. Each chunk is stored as a log, and each entry in the log has references to other chunks. The top chunk begins empty, and then adds another. The bottom chunk adds a data chunk, appends another to the end, and then removes the first chunk.

Figure 3.3: Chunk Skeleton

clients only to discard their backup state. Unfortunately, this is possible no matter who computes coverage rates. An honest node can be given a random list of chunkIds as an abstract; such an abstract is unlikely to match anything. Likewise, nodes can cache and report abstracts sent by others with commonly-appearing chunkIds, hoping for a false match.

3.4 Backup Protocol

A Pastiche node has full control over what, when, and how often to back up. Each discrete backup event is viewed as a single snapshot. Nodes can subscribe to a calendar-based cycle, a landmark-based scheme [99], or any other schedule. Because a machine is responsible for its own archival plan, it keeps a meta-data *skeleton* for each retained snapshot. A file that is not part of the local machine's current file system, but is part of one or more archived snapshots, has a corresponding meta-data entry stored on the local machine.

The skeleton for all retained snapshots is stored as a collection of persistent, per-file logs, as shown in Figures 3.2 and 3.3. The skeleton representing a machine's current file system state plus all retained snapshots is stored both on the machine and all of its backup buddies.

The state necessary to establish a new snapshot consists of three things: the chunks to be added to the backup store, the list of chunks to be removed, and the meta-data objects

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the skeleton that change as a result. We call these the add set, delete set, and meta-data list.

The snapshot process begins by shipping the host's public key. This key will be associated with any new chunks to validate later requests to delete or replace them. The snapshot node then forwards the chunkIds for elements of the add set. If any of those chunks are not already stored on the buddy, the buddy fetches the chunks from the node.

Next, the node sends the delete list. The snapshot host adds a chunkId to the delete list only if it does not belong to any snapshot the client wishes to retain. The delete list must be signed, and this signature is checked to ensure it matches the public key associated with any chunks scheduled for deletion. Note that deletion is not effected immediately. It is deferred to the end of the snapshot process.

Finally, the snapshot node sends any updated meta-data chunks. Since they may overwrite old meta-data chunks, their chunkIds must also be signed. When all state has been transferred, the host requests a *commit* of the checkpoint. Before responding, the buddy must ensure that all new chunks, changed meta-data objects, and deleted chunkIds are stored persistently. Once that is complete, the buddy can respond, and later apply the new snapshot by performing the appropriate deletions.

The performance of snapshots is not crucial, since they are asynchronous. The only exception is marking chunkstore copy-on-write, which must be done synchronously. However, as with AFS's volume clone operation [57], this is inexpensive. The load induced on a buddy by the backup protocol can be regulated with resource containers [6] or progress-based mechanisms [43]. This load is quantified in Section 3.8.2.

The snapshot process is restartable. The most expensive phase—shipping new data chunks—makes progress even in the presence of failures, since new chunks are stored as they arrive. After the new snapshot is applied, a faithful buddy will have a complete copy of the meta-data skeleton, plus all data chunks the skeleton names.

3.5 Restoration

A Pastiche node retains the transitive closure of its archive skeleton, so performing partial restores due to inadvertent file deletion is straightforward. Recovering the entire

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machine after a hardware failure requires a way to bootstrap the skeleton. To do so, a Pastiche node keeps a copy of its root meta-data object on each member of its network-distance leaf set. When a machine must recover from disaster, it rejoins the distance-based overlay with the same nodeId, which is computed from its host name. It then obtains its root node from one of its leaves, and decrypts it with the key generated from the host's passphrase. Since the root block contains the set of buddies in effect when it was replicated, the node can recover all other state. While not currently supported, recovery may also be able to take advantage of other, local sources of chunks, such as installation CDs. This would improve restore times as well as reduce network traffic.

3.6 Alternative Designs

Before settling on Pastiche's current approach, we considered an alternative that appears to be a more natural fit to a peer-to-peer substrate. Instead of having a small list of backup buddies, each holding a complete backup, this alternative stores each chunk on the K Pastry nodes with nodeIds numerically closest to the chunk's identifier. This alternative is commonly called a distributed hash table (DHT) [44, 36] although we will also refer to it as the *fine-grained approach*.

The fine-grained approach has two advantages over Pastiche. First, it ensures that only K backup copies of a chunk exist anywhere in the network. Second, Pastry takes care of detecting failed or unresponsive hosts, and individual nodes need not keep track of them.

However, the fine-grained approach also has two primary disadvantages. The first is the loss of network proximity for most replicas, increasing cross-Internet traffic and latency during restoration. Restoration costs can be avoided by caching along the Pastry route taken by backup chunks, but this increases global disk overhead. The second disadvantage is that dealing with self-interested nodes is more expensive. Pastiche's approach to self-interest is to require a transaction between hosts for each storage request. By replicating at a course-grain, we can amortize the cost of many small object stores across a single transaction. This makes the cost of eliminating free-loaders on the order of the replication factor rather than on the order of the *product* of the replication factor and the number of objects.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Another possibility is to replicate at the granularity of a file system sub-tree. In this scheme, a Pastiche node would try to find different buddies for uncorrelated packages, such as the Windows XP operating system, a collection of "The Matrix" movies, and an archive of Grateful Dead live sets. This approach is attractive because it could significantly increase disk and bandwidth savings due to overlap with only a minor increase in system complexity. Specifically, under sub-tree replication every meta-data chunk would have to contain a list of where its children were replicated.

One can imagine other ways of computing the abstract as well. Rather than randomly sampling the signature, it might be more effective to utilize the chunkstore structure. For example, nodes could compute the abstract by including a single chunk from a file in each directory. This would likely capture a unique chunk from each package.

3.7 Implementation

The Pastiche prototype consists of two main components: the chunkstore file system and a backup daemon. Chunkstore is written in C and is implemented primarily in user space for simplicity. The user-space component is called `pclientd`. A small, in-kernel portion implements the vnode interface [67], integrating chunkstore with Linux 2.4.18. Pastiche uses the XFS device from Arla [115], an open source AFS implementation, for this in-kernel portion.

Data is stored as individual chunks in an underlying file system. For performance reasons, Pastiche also maintains a cache of contiguous, decrypted copies of recently used files, called *container files*. Our prototype does not yet support whole-machine backup, because we have not implemented booting a kernel from chunkstore.

The XFS device sees only container files, and `pclientd` acts as mediator between the device, the container files, and chunkstore. When an application requests a file that is not in a container, `pclientd` retrieves the meta-data chunk for that file from chunkstore and uses it to form a contiguous container file. `pclientd` then returns the inode of the container file to the device, and subsequent operations are applied to the container. The container file cache is managed with LRU replacement, given a maximum size.

`pclientd` is notified of each `close`. If the corresponding file is dirty, it is scheduled

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for chunking. Chunking is deferred for 30 seconds, to avoid needless overhead for short-lived files [113]. We implemented convergent encryption using the `openssl-0.9.7-beta3` cryptographic library. Each chunk was encrypted using a 128-bit key and the AES stream cipher [37].

Container files restore the parity between logical and on-disk proximity that storing chunks individually eliminates. However, storing chunks individually still induces some storage overhead. By storing each chunk separately, Pastiche files will yield more internal fragmentation than if they had been stored contiguously. Recall that content-based indexing generates chunks by examining the lower k bits in a Rabin fingerprint; if these bits match some target value, that offset is marked as a chunk boundary. On average, one would expect to lose half of a disk block per chunk. So, we set k to 14, giving an expected chunk size of 16KB and expected fragmentation overhead to 3.1%.

Meta-data chunks are stored as a log of updates to the file. Each time a file is re-chunked, the list of its constituent chunks is appended to the log. Deletion is represented with a terminal log entry. `pclientd` only appends to these logs, and thus never removes a chunk from chunkstore.

The backup daemon, called `backupd`, is written in C and uses the rpc2 remote procedure call package for communication [102]. It acts as both the backup server and client. The server manages remote requests for storage and restoration, while the client supervises selection of buddies and snapshots. Additionally, `backupd` cleans meta-data logs and reaps deleted chunks.

`backupd` communicates with `pclientd` through file locking of on-disk chunks. This is simple and can be efficient, since `backupd` need not hold all locks to guarantee a consistent snapshot. Once the root meta-data chunk is read, all reachable chunks are guaranteed to remain reachable, since none of them will be deleted by `pclientd`. Some meta-data chunks may still become tainted [66] with additional log entries. However, these entries can be detected via timestamps during backup and restore, rendering their inclusion in a snapshot harmless.

We also provide several utilities to allow users to manage the file system: `forcesnap` forces `backupd` to take a system snapshot immediately, `forcechunk` forces `pclientd`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to chunk all files in its queue immediately, and `rfile` restores a file or subtree to a previous state. Each utility communicates with `pclientd` and `backupd` through Unix domain sockets.

3.8 Evaluation

In evaluating our prototype, we set out to answer the following questions:

- What is the performance of the file system? Which operations perform well and which perform badly?
- How long do backups and restores take?
- How large must fingerprints be? Is the lighthouse sweep able to find buddies?
- Does the coverage-rate overlay yield suitable backup buddies?
- Are the costs to detect malicious nodes reasonable?

All experiments were run on machines with a 550 MHz Pentium III Xeon processor, 256MB of memory, and a 10k RPM SCSI Ultra wide disk, with 4.7 ms seek time, 3.0 ms rotational latency, and 41 MB/s peak throughput.

3.8.1 Performance

What is the overhead induced by chunkstore? To answer this, we compare the performance of chunkstore to the underlying, native file system, `ext2fs`. We measure this overhead with a modified Andrew Benchmark [57]. Our benchmark is identical to the original in form, but uses the `apache 1.3.26` source tree. This source tree is 9.6MB in size; when compiled, the tree occupies 12MB.

We ran five trials; the results are shown in Table 3.1. While the `make` step is not I/O bound, it does experience slight overhead. This is due in part to the cost of computing the Rabin fingerprints of the copied tree and the extra cost of creating and deleting files. The copied data is scheduled to be chunked when written, and 30 seconds later—during the `make` step—chunking begins.

The total overhead of 7.4% is reasonable, though the `copy` phase is expensive; it takes 80% longer in chunkstore. Workloads with intensive I/O will likely experience very poor performance. We believe that this overhead is due to excess meta-data management

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

AB phase	ext2fs		chunkstore	
mkdir	1.23	(0.04)	1.03	(0.05)
cp	3.47	(0.28)	6.26	(0.16)
scandir	0.0	(0)	0.03	(0)
cat	1.75	(0.02)	2.23	(0.02)
make	38.62	(0.45)	38.88	(0.5)
total	45.08	(0.39)	48.43	(0.58)

This table presents the results of a modified Andrew Benchmark. Times are reported in seconds, and standard deviations are given in parentheses.

Table 3.1: Andrew Benchmark

Task	ext2fs		chunkstore	
wide create	2.44	(0.06)	6.99	(0.02)
wide mkdir	2.30	(0.02)	6.31	(0.03)
deep mkdir	4.07	(0.02)	5.64	(0.01)
bulk xfer	12.79	(0.01)	12.75	(0.02)

This figure presents the results of file creation and I/O throughput benchmarks. Times are reported in seconds, and standard deviations are given in parentheses.

Table 3.2: Micro-benchmarks

in chunkstore, rather than limits on peak I/O throughput. To confirm our hypothesis, we performed several micro-benchmarks to isolate the operations involved in copying a source tree - writing data and creating files.

To examine chunkstore's performance when creating files and directories, we ran three different experiments—`wide create`, `wide mkdir`, and `deep mkdir`. In `wide create`, 1000 new files were created in the same directory. In `wide mkdir` 1000 new directories were created in the same directory, and in `deep mkdir` 1000 new directories were made recursively inside of one another. We again ran five trials of each; the results are in Table 3.2.

`wide create` and `wide mkdir` each ran about 186% and 174% slower than `ext2fs`, respectively, while `deep mkdir` ran about 38% slower. Chunkstore's poor performance is due to meta-data chunk and container file maintenance. When chunkstore creates a file, it must update or create three files: a new meta-data chunk, a new container file, and the parent meta-data chunk.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Task	time	
cp	6.26	(0.07)
backup	6.55	(0.01)
rm	1.24	(0.01)
restore	5.54	(0.07)
nfs cp	3.76	(0.16)

This figure presents the results of the backup and restore experiment. Times are reported in seconds, and standard deviations are given in parentheses.

Table 3.3: Backup and Restore

The deep mkdir experiment shows that the number of entries in the parent directory is also significant. This is because of the way directory entries are laid out in the metadata chunks and the container files. In both cases, directory entries are stored in a linear array. Our current implementation rewrites the entire list to the container file and chunk whenever a new entry is added. During deep mkdir, there is only ever one entry in the list, which makes creating a file faster.

It is also interesting to note that wide mkdir is somewhat faster than wide create. The reason for this is related to how the in-kernel XFS device handles file and directory creation. When a regular file is created, the XFS device makes an extra upcall to pclientd to close the newly created file, and does not make this call when a directory is created.

To further verify that I/O throughput was not responsible for chunkstore's slow copy phase in the modified Andrew Benchmark, we also ran a bulk xfer experiment. In this experiment, a new file was created, 256MB of data were written to it, and then the file was closed. As before, we ran five trials; the results are in Table 3.2. Chunkstore and ext2fs performed within 1% of each other, meaning that their I/O throughput are statistically identical.

3.8.2 Backing Up and Recovering a File System

To determine the performance of our backup and restore utilities, we applied them to a file system consisting of the openssl-0.9.7-beta3 source tree. This 13.4MB tree of 1641 files and 109 directories is stored in Pastiche as 4004 chunks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each of five trials consisted of four phases - copying the source tree into the file system, sending it to a backup buddy, removing the local source tree, and restoring the source tree from the backup buddy. Pastiche's backup and restore performs comparably to the time to copy the source tree over NFS. The results are in Table 3.3.

It should also be noted that the demand on resources the buddy experiences while carrying out backup and restore is very bursty. During the five trials, `backupd` used a maximum of 8MB of memory, averaged 12 disk transfers/sec with a maximum of 414 transfers/sec, and averaged a 70% idle CPU with a minimum of 13%.

3.8.3 Finding Buddies

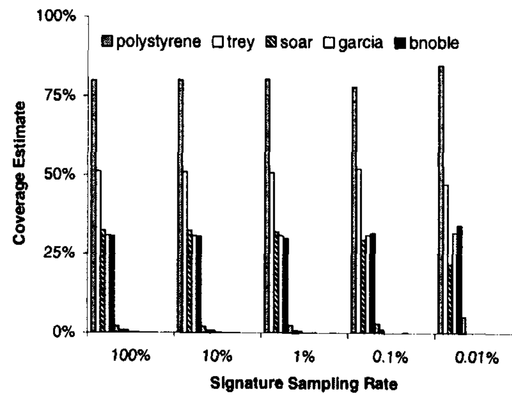
Next we turn our attention to the buddy discovery process. There are three questions to answer. First, how large must an abstract be to discriminate good buddy candidates from bad ones? Second, how effective is the lighthouse sweep in discovering buddies? Third, how effective is the coverage-rate overlay in discovering buddies? To answer the first question, we took the signatures of seventeen machines at Michigan. These machines run Windows, Linux, Solaris, and various flavors of BSD. We also took the signatures of two freshly installed machines.

The first ran Windows 98 with an Office 2000 Professional installation, but without any service packs applied. This machine held roughly 90 thousand chunks[1]. The second was a Linux machine running a Debian `unstable` release, configured as a conventional workstation with development and document processing tools. This machine held approximately 270 thousand chunks. We chose this machine as a worst case. Some of our comparison machines are Debian, but only one runs the `unstable` distribution. This distribution changes quickly, and this machine is updated infrequently.

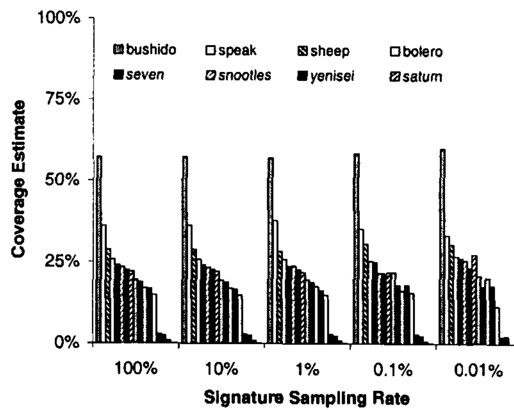
We computed the actual coverage for each of these machines given full signatures. To estimate the impact of smaller abstracts on coverage estimates, we took uniform random samples of the signature at rates of 10%, 1%, 0.1%, and 0.01%; there are six samples at each rate.

[1]For this experiment, we used a smaller expected chunk size of 4KB; Pastiche's larger chunks may require slightly larger sampling rates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Windows 98/Office 2000



(b) Debian Developer

Figure 3.4: Varying Abstract Size

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The results for the Windows 98 machine are plotted in Figure 3.4(a). The x axis gives sampling rate, and the y axis shows coverage rate. The 100% "sample" shows exact coverage; each of the others is an estimate given a smaller sample. Each group of bars represents the coverage estimate for each of the seventeen hosts. Within each group, the hosts are sorted by actual coverage rate, from highest to lowest. The top five matches are identified in the legend. `polystyrene` is a Win98 machine running Office 2000, with all relevant service packs and security updates applied.

The estimates are surprisingly independent of sample size; the lowest rate produces abstracts of around 10 chunkIds. Only `soar`'s estimate changes appreciably. However, its coverage rate is comparable to `garcia`'s and `bnoble`'s; choosing either of the latter in favor of `soar` is of no consequence.

Figure 3.4(b) shows the results for our Linux machine, the top eight matches are identified in the legend. The overall match rates are lower, but machines with other distributions still have substantial matches. As with the Windows 98 host, coverage estimates do not change materially as abstract sizes go down. Interestingly, `bnoble`, a Windows 2000 machine, has a coverage rate for this Debian machine of almost 15%. This is because `bnoble` also has a `stable` release of Debian, installed in a VMware virtual machine; the VMware disk image is stored as a regular file in Windows. Ordinarily, files form implicit chunk boundaries in content-based indexing. When viewed from the windows host, all of these file boundaries disappear. Despite this, content-based indexing is still able to find substantial overlap.

Small abstracts are effective only if they are delivered to a host that can provide good coverage. We conducted a simulation to determine how effectively lighthouse sweeps find useful buddies. This simulation uses SimPastry [76], a Pastry simulation/visualization tool.

For the simulation, we populated a graph with 50 thousand Pastiche nodes drawn from a distribution of 11 *types*. 30% of all nodes are the first type, types two and three each comprise 20% of all nodes, types four and five each comprise 10%, type six comprises 5%, and types seven through eleven each represent 1% of the population.

We simulated 25 different Pastry networks under these conditions. For each network,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

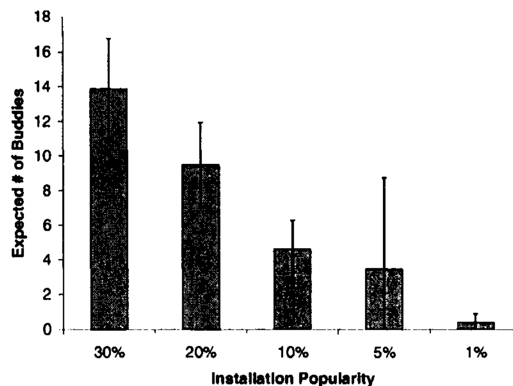


Figure 3.5: Expected Number of Buddies

we randomly selected 1000 nodes of each type, and performed a lighthouse sweep from that node, counting the number of hosts of identical type found during the sweep. The results are shown in Figure 3.5. Each bar gives the average number of matches found per sweep for each category of popularity; the error bars show one standard deviation.

As expected, common nodes with representation of 10% or higher should find an adequate number of buddies on the distance overlay. Those with lower popularity will need to join the coverage-rate overlay as well. We built a Pastry simulator to determine the effectiveness of this network. Our experiments involved 10,000 nodes. Each node was assigned to one of a thousand species, one of a hundred genera, and one of ten orders. Nodes of the same order share 20% of their content; nodes of the same genus, 30%; and nodes of the same species, 70%. Only nodes of the same species can serve as backup buddies for one another.

The results of our simulations are in Figure 3.6. The x axis gives the size of the neighborhood set, and the y axis shows the percent of all nodes who found a given number of buddies. We ran four series of trials, varying the size of the neighborhood set. We found that for a neighborhood set size of 256, 85% were able to find at least one buddy in its routing table, and 72% were able to find at least 5.

The results show that most nodes should be able to find buddies in the coverage-rate overlay. It also shows how important a role the neighborhood set plays in locating buddies. Increasing the neighborhood set from 0 to 256 increases the percent of nodes who can find at least one buddy from 38% to 85%; the percent of nodes who can find at least 5 increases

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

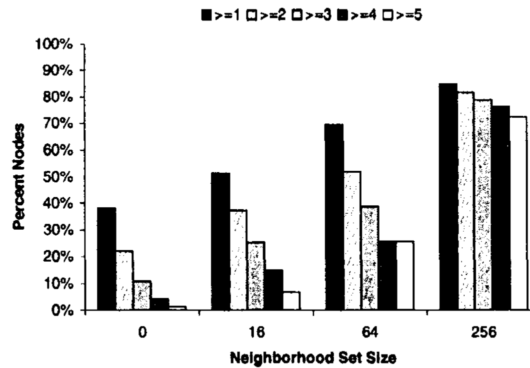


Figure 3.6: Coverage-rate Simulation Results

from 1% to 72%. The Pastry group has deprecated the neighborhood set, but this result shows that removing it may make it difficult for nodes with uncommon installations to locate buddies.

3.9 Discussion

Pastiche depends on nodes contributing excess storage to the collective. In some environments nodes will reliably contribute, although it may require outside encouragement. For example, within an organization, administrators likely have enough control over machines to regulate how much storage users consume and contribute. However, in the absence of such a governing entity or an altruistic membership, Pastiche could face two problems as a result of its self-interested hosts: over-consumption and free-loading.

Over-consumption occurs when greedy hosts aggressively consume space by using storage on many hosts. This is commonly referred to as the "tragedy of the commons." [54] One way to address this problem is to place nodes into equivalence classes based on the resources they consume. Each node could monitor the overall storage costs imposed by its backup clients, and compare these costs to its own usage. Those that are much more space-intensive are ejected, and must search for a more suitable partner. Unfortunately, this mechanism can be easily circumvented by creating multiple identities. This is known as the Sybil attack [41].

Another approach is to force nodes to pay some price for the storage they use. For ex-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ample, nodes might have to solve cryptographic puzzles [65] in proportion to the amount of storage they occupy. Forging identities is no defense against this, nor is spreading snapshots across more than the usual number of buddies. However, this solution is unsatisfactory for two reasons. First, it trades something other than storage for storage space. CPU cycles and blocks of storage are incommensurable resources, which makes pricing one in terms of the other problematic. Second, not all nodes will have equivalent processing power, so it is difficult to provision the solution properly. The amount of storage a node is allowed to consume would be more fairly bound to the amount of storage it contributes rather than the idleness of its processor.

The third approach is to account for space with some form of electronic currency [24]. It is sufficient to use an off-line protocol [13]; some amount of double-spending is tolerable as long as abusers can be detected eventually. However, currency accounting requires that backup be *goods atomic* [110]; the exchange of currency and backup state must be an atomic transaction. Once a node has recirculated any received currency, there is no way to punish the node if it discards the data it is responsible for. Also, any fiat currency must be certified by a trusted authority. This would incur costs similar to those of a public key infrastructure.

The second potential problem is free-loading. Free-loading is when nodes refuse to contribute storage to the collective. It is similar to over-consumption, but in some ways is more difficult. Consumption can be limited by exacting a toll for each consumed unit, such as CPU cycles or currency. However, if excess local storage is more valuable than what it is exchanged for, nodes have no incentive to contribute. This is most clearly true of the cryptographic puzzle scheme, where puzzle solutions have no inherent value. If fiat currency were used, nodes would need to generate enough revenue selling access to their storage to pay the trusted authority to certify the currency. It is unclear how common such a scenario might be. Thus, to deal with self-interest, Pastiche must provide hosts with an incentive to share their excess storage.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

BILATERAL, EQUAL EXCHANGE

Pastiche networks with rampant free-loading are not sustainable. If the demand for storage is much more than the supply, what storage is available will quickly become occupied. The result would be substantial unmet demand, some of which will fall on the very nodes who are contributing to the collective. The danger is that contributers will opt out if the cost of sharing outweighs the benefit. Losing part of the already limited storage supply will lead to even greater unmet demand and more unsatisfied contributers, triggering a series of cascading withdrawals and the collapse of the storage pool.

Pastiche assumes that users are self-interested, or *rational*. Rational users want to maximize the difference between the value and cost of using Pastiche. In addition to normal costs of running networked software such as the security risks posed by bugs and additional CPU and memory overhead of another process, running Pastiche costs storage and network bandwidth contributions to the collective. Pastiche shares the goal of building services from privately owned over-capacity with computational technologies such as grid computing [4] and wireless mesh networking [52] as well as with non-computational services like carpooling. Benkler [10] calls this mode of production "social sharing" and has identified two fundamental disincentives to contribution.

First, users cannot be expected to compromise their *security*. For example, if a driver carpools with someone into work, they must not feel physically endangered by the rider. Second, services must address the *opportunity cost* of contributing. In the case of carpooling, the opportunity cost is the additional time it takes to pick up and drop off other passengers. To compensate for this opportunity cost, many cities give cars with multiple

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

passengers access to less congested High Occupancy Vehicle (HOV) lanes. In the absence of a trusted authority, Pastiche must address both disincentives to avoid collapsing the storage pool.

The rest of this chapter is organized as follows. In Section 4.1, we explore the applicability of Benkler's disincentives to Pastiche. We can use observations of peer-to-peer file sharing to give us an idea of how these disincentives will affect Pastiche users. The wide use of file sharing software indicates that users are willing to incur some extra security risk to run software that provides a valuable service. However, several studies of users' behavior in file sharing indicate that opportunity costs present a strong disincentive to contribution.

In Section 4.2, we describe our first building block for encouraging contribution, called *bilateral, equal exchange* (BEE). This protocol ensures that all nodes contribute as much storage as they consume and punishes nodes who attempt to cheat by discarding their data.

Unfortunately, simple BEE is too severe in practice. Nodes in peer-to-peer networks often experience periods of temporary disconnectedness called *transient failures*. These are not attempts to cheat, but can appear to be under BEE. In Section 4.3, we explore the challenges of transient failure in BEE and offer a technique called *probabilistic discard* to address it. Probabilistic discard attempts to punish chronic abusers while allowing transiently failed users to retain their data.

Finally, in Section 4.4 we present a formal model of BEE. Analysis of the model shows that rational nodes will obey the protocol under realistic conditions. More important, this model provides a conservative estimate of to cost of running Pastiche. According to this calculation, running Pastiche under BEE costs less than one dollar per month for even up to 24GB of unique data, making Pastiche significantly less expensive than existing backup options.

4.1 Disincentives to Contribute

Benkler identifies two fundamental disincentives to contributing to the "social sharing" model of production utilized by Pastiche—security concerns and opportunity costs. Security concerns are inherent to the Internet. Hackers and viruses are a fact of on-line

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

life and Pastiche cannot make these risks disappear. However, sharing excess storage does not increase this threat. Users do not compromise their security by storing other nodes' data because it is only stored and never executed. Pastiche does expose some user information to enable inter-host data sharing, but users can limit this exposure by encrypting their blocks with independently chosen keys.

Contributing to Pastiche also requires users to run `pclientd`. This incurs additional security risks, because `pclientd` might contain bugs that can be exploited by hackers. However, this risk is no greater than the risk of running a peer-to-peer file sharing client. A study by ITIC of online activity for November, 2004 found that there were nearly 10 million file sharers on-line at any moment, an increase of just over 2% from November, 2003 [62]. Security worries have not deterred users from running file sharing software and they should not deter Pastiche users from running `pclientd` either.

Unfortunately, Pastiche users do incur opportunity costs for contributing their excess disk space. Disks are mostly empty, but granting access to unused storage may interfere with normal disk IO, especially during queries. Furthermore, although most network bandwidth goes unused, the likelihood that a Pastiche data transfer will interfere with normal network use is also non-zero. Unless storage and bandwidth capacity are infinite, users will always incur a positive opportunity costs due to the resource contention caused by congestion. It is in rational users' interest to minimize these costs by contributing as little as possible to secure their backup state.

Strategic behavior is not merely an academic invention. Computer users have demonstrated strategic behavior in a number of settings. A four month study of the Mirage [83] auction-based resource allocator for sensor network test-beds found that many users tried to game the auction protocol. Many on-line game players run altered clients that tilt the field in their favor. This form of cheating was one of the primary motivations for the Terra secure computing platform [49].

The examples of strategic behavior most germane to Pastiche come from peer-to-peer file sharing, where users must also decide whether or not to contribute their own excess resources. A study of Gnutella [1] found that two thirds of the participants provide no files for other users. The most generous 1% of hosts serviced nearly half of all requests. A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

later study of Gnutella and Napster [101] confirms this general trend, though with a less dramatic fraction of free-loaders. This study also found that almost one third of all Napster users under-reported their available bandwidth to avoid being selected as a download site. These results demonstrate that the opportunity costs of contribution drive file sharers to withhold their excess resources.

Despite this, file sharing gracefully accommodates free-loading. Might Pastiche be able to as well? To see why not, we must draw a distinction between *exclusionary* and *nonexclusionary* resources.

Nonexclusionary resources can be used by multiple parties at once. Files are nonexclusionary since users can access them simultaneously. File sharers can queue to download a file and then use a local copy later. Files are similar to weather reports available to farmers, research available to scientists, or advertising available to consumers.

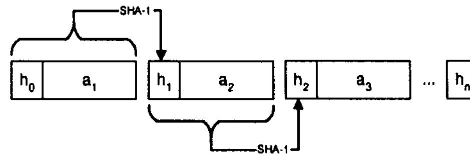
On the other hand, assuming there is no aggregation, the bandwidth used by both file sharers and Pastiche is exclusionary. This is because a unit of bandwidth can only be utilized by the source and destination of the packet it carries. However, bandwidth is also *renewable* since it becomes available again once a packet has been processed. In the presence of a deficit, bandwidth's exclusivity can curb file sharing, but its renewability will prevent a collapse.

Disk space, on the other hand, is both exclusionary and nonrenewable. Storage space cannot be occupied by multiple users' data at once and is not replenished over time. This is why the storage deficits induced by free-loading are such a threat to Pastiche. Users cannot simply queue once the storage supply is fully utilized. Thus, to be viable, Pastiche must eliminate free-loading.

4.2 The Bilateral, Equal Exchange Protocol

Pastiche cannot eliminate the opportunity costs of sharing. However, it can eliminate the possibility of cascading withdrawals by ensuring that the cost of participation is never greater than the benefit. We do this by linking the reliability of nodes' backup state to their level of contribution. The more storage a node contributes, the more replicas it is allowed to create. If a node chooses to contribute nothing, it eventually loses access to the shared

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure demonstrates how a node can respond to a query. The querying node supplies an initial value h_0, which is prepended to the first data object, a_1, and hashed to produce h_1. This is then prepended to the second object, a_2, and hashed, and so on. The storing node only needs to return the final hash h_n.

Figure 4.1: Query response construction.

storage pool.

One way to bind levels of service to contribution is to allocate excess storage through bilateral, equal exchanges (BEE). That is, node A may store data on node B if and only if B may also store the same amount of data on A. B can periodically *query* A to see if its data is still held by A, and vice versa. If a node fails a query, its data is discarded. Collectively, these pairwise checks ensure that each node contributes as much as it consumes. Furthermore, the threat of a retaliatory discard provides a strong incentive to contribute and maintain storage. Nodes cannot use anyone else's storage without giving up an equal amount of their own.

For these trades to make sense, we must assume that a unit of remote storage is more valuable than a unit of local storage. This is reasonable since recovery would be impossible without copying data to another disk.

The overhead of the protocol is also very low. Because querying is only intended to demonstrate that data is stored rather than to examine its contents, queries can be made relatively infrequently. A node could reasonably query its replica sites every few hours or even once a day. Similarly, queries need not be answered immediately. This is important because nodes might want to delay responding to a query when they are busy. There are many ways to minimize the burden of answering a query, including resource containers [6], progress-based mechanisms [43], and free disk bandwidth [71].

The network bandwidth required to satisfy a query can be reduced to a single SHA-1 hash as well. There is no need to return the entire data object to prove that it is still stored. When querying, nodes can send a unique value, h_0, along with the list of n objects they

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

wish to verify. Responding nodes prepend h_0 to the first object in the list and compute the SHA-1 hash of this concatenation. This hash is called h_1 and is prepended to the second object to compute h_2 and so on. The responding node only needs to return h_n to prove that it is storing all the objects it is responsible for. This process is shown in Figure 4.1. The overhead of querying is described in Section 5.4.1.

Querying under the threat of discard provides a strong incentive to maintain storage for others, but it does not provide an incentive to participate in recovery. This is because there is no way to reciprocate a recovery. If node A and node B exchange backup state and A fails, A will have lost all of B's data and cannot help B restore. Knowing this, B could discard A's data once it received enough data requests to indicate an in-progress recovery. Because A has already lost B's data, there is no way for A to punish B. If all nodes behave this way, recovery will be impossible.

To create an incentive to participate in recoveries, Pastiche nodes carry out randomly scheduled *fire drills*. A fire drill is false, full recovery that is indistinguishable from the real thing. If a node fails to participate in a fire drill, it risks having its data discarded. Because of this, nodes must participate in recoveries because they will look exactly like a fire drill. In practice, fire drills should be relatively rare events to be scheduled on the order of weeks or even months.

Interestingly, fire drills also allow nodes to detect when their data has been rerouted to their own disk. In this attack, whenever node A gives node B a data block to store, node B hands it right back to A to store. B could conceal its attack by encrypting each block from A before sending it back. To answer data queries, B would request A's re-encrypted data from A. A would never know it was storing its own data. However, because fire drills simulate real crashes, B may not retrieve data from A while a fire drill is in progress and B's attack will be exposed.

Another important property of bilateral, equal exchange is that it does not require certified identities or trusted third parties and does not enable the Sybil attack. Other approaches that utilize reputation systems [12], currency [61], distributed auditing [84], or micro-payments [60] require a trusted infrastructure, but these services are not free. The only way their use can scale to a large collective is to exact a fee from each user. This is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

precisely the kind of additional cost for which backup users have shown little tolerance.

It is important to note that we have not attempted to force users to use their full bandwidth capacity, nor will we. BEE is only concerned with storage. If a user wants to throttle their bandwidth to slow the rate of replication or rate of restore, Pastiche cannot stop them. Of course, if a node fails to satisfactorily participate in fire drills, it risks being dropped by its buddies.

4.3 Transient Failure

Nodes in peer to peer networks suffer from frequent transient failures [17]. Distinguishing between a node that is trying to cheat and one that experiences a transient failure is both difficult and important. Nodes should not lose all of their remote data because of a temporary loss of connectivity or failure. This is especially true for a backup service like Pastiche. When a node's disk dies, it is unable to respond to queries. If the node's remote data is then discarded because it fails a query, restore will be impossible!

One potential solution gives nodes a *grace period* for responding to a query. If this grace period is longer than a conservative estimate to recover a failed node—on the order of many days to weeks—then honest but unfortunate peers will not be penalized. This scheme tolerates transient failure, but it also leads to a straightforward grace period attack [69].

A node could choose replicas for grace periods at a time, never store anything in return, and select new peers once the grace period expires. This is easy to do in systems like Pastiche, where nodes have full discretion in selecting replica sites. It is also possible to mount this attack in a content-addressable system like a DHT. For example, nodes could encrypt their objects under a rotating key scheduled every grace period. The resulting objects have the same semantic content as their source, but different object identifiers, and will thus be stored on different replica sets.

The grace period attack exploits the fact that node only discard data at the end of the grace period and never before. Pastiche's approach retains the notion of a grace period, but clears cheaters' data through independent, *probabilistic discards*.

Each day, buddies independently discard an object with probability p, where p grows

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

exponentially with the number of consecutively failed queries. If a node never fails a query $p = 0$ and if a node fails a grace period number of consecutive queries, $p = 1$. We denote the discard probability after i consecutively failed queries, p_i. Besides i, buddies only need the normative replication factor, denoted r, to compute p_i.

Since data is replicated, probabilistic discard protects transiently failed nodes. If a node is temporarily disconnected and as a result fails consecutive queries from its buddies, the likelihood that each buddy will simultaneously discard the object is extremely low. If a subset of buddies do discard an object, these copies can be reinstated from the existing ones.

There are two complementary strategies for defeating probabilistic discard that immediately come to mind. First, as in the original grace period attack, a cheating node could create a brand new set of replicas frequently. In addition to this, a node could create many replicas to increase the expected number of consecutively failed queries before losing an object. A cheating node would have to create new replicas fast enough to ensure that there is at least one copy of each object at all times. As we will see, the bandwidth required to create enough replicas fast enough is prohibitive for most nodes.

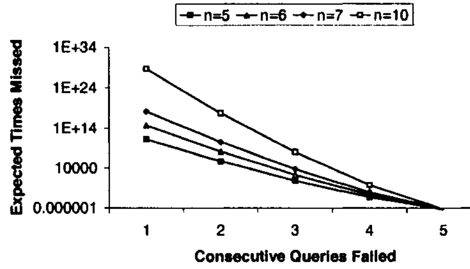
To see why p_i must grow exponentially, assume that we would like an unresponsive node to lose one replica for each consecutively failed query. For the ith consecutive failed query, each replica site computes

$$p_i = \frac{1}{r - i + 1}$$

If the unresponsive node has n replicas, the expected number of discarded replicas after failing the first query of each replica site is $D_1 = \frac{n}{r}$. More generally, $D_i = \frac{n - D_{i-1}}{r-(i-1)}$ discards are expected after failing the ith consecutive query of each site. Thus, $D_r = n - D_{r-1}$, which means that after r consecutive failed queries at each site, all n replicas should be discarded.

Unfortunately this p_i is too severe. If p_1 were $\frac{1}{r}$, a node with n replicas and one object can expect all n buddies to simultaneously discard that object after missing one consecutive query with probability $\frac{1}{r^n}$. This means that for $r = n = 5$, a node can expect to lose an object after missing a single query at each buddies only $5^5 = 3{,}125$ times.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure graphs query generation versus the expected number of times a node can miss that many consecutive queries before losing an object. The normative replication factor for the system is assumed to be five (r = 5), but the adversary populates n replicas. Each node owns 2,000,000 objects. Each series represents the number of replicas actually created.

Figure 4.2: Expected consecutively missed queries before object loss.

This is far too harsh, since nodes are likely to have many more than one object, and replica sites probabilistically discard each object stored after a query has been missed. For example, a node with 3,125 objects can expect to lose one object every time it misses a single consecutive query at all of its buddies. A typical Pastiche node might have as many as 2,000,000 objects, equal to about 32GB of state. With five replicas, this node can expect to lose 640 objects the first time it is disconnected for a full day.

Instead, we can make p_i convex with respect to i so that failures become more harmful as consecutive queries are missed. This allows p_i to be very low for small i so that honest nodes are protected and allows p_i to approach 1 as i comes closer to the grace period so that cheaters are punished. While there are an infinite number of shapes that p_i could take, we will use a simple, exponential curve, as follows:

$$p_i = \left(\frac{1}{r-i+1}\right)^{(r-i+1)}$$

For r = 5, a node with 2,000,000 objects can expect to lose an object after missing a query at each buddy once per 1.49×10^{11} times. If queries are made on a daily basis, this would require $2 \times 4.08 \times 10^{8}$ years of alternatively missing and then satisfying a query. This should protect users who are regularly disconnected for short periods of time.

Just as important, habitual cheaters are still punished. The expected number of times that specific numbers of consecutive queries can be missed before losing an object is graphed for various replication factors in Figure 4.2. The greater the replication factor,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the less likely an object will be lost. However, as the figure shows, cheating nodes can expect to lose an object eventually, even when the replication factor is double the norm.

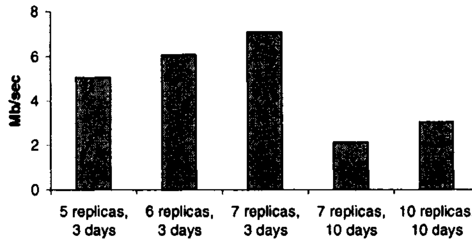
Users commonly experience one or two day periods of disconnectedness in conjunction with the end of the work day or weekend [17]. It is important that Pastiche not punish users for these normal unresponsive spells. Unfortunately, with $n = r = 5$, 2,000,000 objects, users may turn off their machines for more than two days only seven times over before they can expect to lose an object. Adding an additional replica raises the number of three day outages to 194, which is reasonable. Even a behaving node will occasionally miss three consecutive queries, but it should happen infrequently.

These curves descend quickly, however. The expected number of lost objects after missing four consecutive query jumps to 488. Because of this, for the first three days, queries should be scheduled everyday, while the fourth query should be scheduled a week later. Cheating nodes can still expect to eventually lose objects after just three days because of the number of times they will miss three consecutive queries.

Still, as alluded to earlier, a cheating node might be able to defeat probabilistic discard by creating many replicas and then replacing them before it expects to lose an object. A node with seven replicas can expect to lose an object after missing three consecutive queries 5,230 times. This is fairly risky. However, a node with ten replicas can expect to miss three consecutive queries over 100 million times before losing an object. For this node, missing three consecutive queries is probably safe, even if it expects to lose two objects if it fails four consecutive queries. Fortunately, the bandwidth required to create seven new replicas every three days or ten new replicas every ten days is prohibitive for most nodes.

Figure 4.3 shows the continuous outbound bandwidth required for various combinations of replication factors and days to create those replicas. Assume that a cheating node has 2,000,000 objects and 32GB of state. For a node with five replicas, it will need to create approximately five replacement replicas within three days to avoid losing an object. To create five new replicas within three days a node must ship 160GB of data, inducing continuous outbound bandwidth traffic of approximately 5Mb/s. If the node tries to maintain six replicas, the bandwidth requirement is about 6Mb/s. Even a node with ten replicas,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure shows the continuous outbound bandwidth required to create five, six, seven, or ten replicas every three or ten days. Nodes are assumed to have 32GB of state and a normative replication factor of five.

Figure 4.3: Required bandwidth to create replicas.

who is unlikely to ever lose any objects after missing three consecutive queries, needs over 2Mb/s to recreate ten replicas over ten days.

Bandwidth between arbitrary points on the Internet is unlikely to approach these levels anytime soon [70]. Current cable modem upload bandwidth in Ann Arbor was 384 Kbps as of August 2005. Furthermore, as we will see in Section 4.5.2 continuously paying network bandwidth more expensive than the storage an attacker is trying to save. Paying for remote storage with bandwidth instead of storage is simply not cost effective.

However, while bandwidth constraints will likely keep nodes with data sets of 32GB or larger from cheating, this might not be enough to prevent nodes with only a few objects from cheating. The less data a node needs to replicate, the less bandwidth it needs to create new replicas every three or ten days. While this is true, nodes with smaller data sets are, by definition, unlikely to consume large amounts of storage.

Nonetheless, if nodes with smaller data sets become a problem, we could further refine p_i. One possibility is to identify which nodes are capable of cheating. By using the amount of storage consumed and the rate at which it was copied, nodes could approximate the maximum number of replicas a node is capable of creating before it expects to lose an object. The discard rate could be set higher for nodes with more bandwidth and lower for nodes with less.

Finally, after the owner of the failed disk recovers, what should happen to its data? Currently, we assume that eventually an unresponsive node's backup state will be discarded, hopefully after it recovers. However, if a node truly fails, it could send its replicas

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a post-recovery message saying that it would like to store its buddies' data again. While they would be under no obligation to do so, buddies could then resend their lost data if they believe that it was the result of failure and not malice. This would save the recovered node and its buddies bandwidth or the effort of trying to find brand new replica sites.

4.4 Exchange Model and Analysis

Section 4.3 introduced the grace period attack and probabilistic discard as a way to reduce cheaters' quality of service. There is an alternative, natural "defense" against the grace period attack that relies on the relative costs of storage and network bandwidth. It may be more expensive for a node to frequently spend large amounts of network bandwidth, than to simply store what it is supposed to. To explore this further, we modeled a BEE network as an economic game.

One simplification of our model is the use of a static grace period, rather than probabilistic discard. This allows us to keep all of our model constraints linear, but also isolates the effect of relative resource costs on behavior. In future versions of the model, we hope to examine the effect of probabilistic discard.

To further simplify the analysis, we assume that all but one node obeys the exchange protocol. The well-behaved nodes give up storage equal to what they consume and maintain that storage faithfully from that point forward. The other node, denoted c, acts in accordance with the solution to an optimization problem.

Our goal is to establish when the rational node's optimal strategy is to contribute and when it is to cheat. This will depend on several model parameters, including the relative cost of resources (storage and network bandwidth), bandwidth capacity, desired level of service, and grace period length.

The optimizing node attempts to minimize the cost of achieving a desired level of service. We only considered the costs of upstream bandwidth, downstream bandwidth, and disk space. There are other costs that we ignored such as lost CPU cycles, the added security risks of being on the Internet, or the extra power consumed by running Pastiche. However, this is reasonable since CPUs are vastly under-utilized for most users and Pastiche does not require machines to be continuously running and on-line.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In our model, a Pastiche network is populated by a single rational user, c, and a set, H, of n well-behaved storage hosts, denoted $H = \{h_1, h_2, \ldots, h_n\}$. Each h_i owns a single block of excess storage, denoted $b_1, b_2, \ldots, b_n$. h_i controls access to block b_i. c owns a set of n blocks of excess storage, denoted $d_1, d_2, \ldots, d_n$. Only h_i can gain access to d_i.

c has only one block of data to backup. It can acquire storage on other hosts through *exchanges* with h_i. An exchange ϵ is a member of $H \times [1, T]^4$, where T is the number of rounds in the game. Each exchange ϵ can be written $[h_i, \alpha_c, \omega_c, \alpha_i, \omega_i]$. The function $E : H \times [1, T]^4 \rightarrow \{0, 1\}$ determines which exchanges occur. $E([h_i, \alpha_c, \omega_c, \alpha_i, \omega_i]) = 1$ if and only if c grants h_i access to d_i in round α_c and revokes it in round ω_c. In return, c gains access to b_i in round α_i and loses access in round ω_i. For example, if $E([h_1, 1, 5, 1, 1]) = 1$ then c has access to block b_1 during rounds 1, 2, 3, and 4 and h_1 never has access to block d_1. This is equivalent to c getting something from h_1 for nothing for rounds 1, 2, 3, and 4.

We modeled BEE as constraints on the mappings of E. These constraints determine which exchanges are *illegal*. If ϵ is illegal, then $E(\epsilon) = 0$. A *strategy* is a set of legal exchanges where $E(\epsilon) = 1$ for each ϵ in the strategy. To evaluate BEE, we can solve an optimization problem over our constraints to derive an optimal strategy.

4.4.1 Problem Constraints

Our first group of constraints enforce the ordering of events in exchanges. First, access to d_i and b_i must be granted simultaneously. Second, access to a block cannot be revoked before it is given. Thus, for $\epsilon = [h_i, \alpha_c, \omega_c, \alpha_i, \omega_i]$

$$\alpha_c \neq \alpha_i \;\Rightarrow\; E(\epsilon) = 0$$

$$\omega_c < \alpha_c \;\Rightarrow\; E(\epsilon) = 0$$

$$\omega_i < \alpha_i \;\Rightarrow\; E(\epsilon) = 0$$

Our next constraint uses the parameter G, the length of the grace period. G determines for how many rounds c may access b_i after c has revoked access to d_i. This constraint stipulates that h_i will not grant access to b_i G rounds after it loses access to d_i. Thus, for $\epsilon = [h_i, \alpha_c, \omega_c, \alpha_i, \omega_i]$

$$(\omega_i - \omega_c) \geq G \;\Rightarrow\; E(\epsilon) = 0$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We must also limit the number of exchanges c can make per round, which we control using the parameter B. B denotes the bandwidth capacity per round of c. c cannot make more than B exchanges per round. Thus, for all h_i, ω_c, ω_i:

$$\sum_{r=1}^{T} E([h_i, r, \omega_c, r, \omega_i]) \leq B$$

We need another set of constraints to capture the level of service that c wants to achieve. The parameter R represents c's desired replication factor. At the very least, c would like to have access to R blocks by the end of the game (round T). However, setting a specific round by which c should have R replicas is unrealistic. It would be equivalent to c's knowing when its disk will fail. More realistically, c would want to maintain R replicas for a period of P rounds. Thus, for all $\alpha_c, \omega_c, \alpha_i$, and $t \in [T - P, T]$

$$\sum_{j=1}^{n} E([h_j, \alpha_c, \omega_c, \alpha_i, t]) \geq R$$

4.4.2 Problem Objective

With these constraints, we can cast our problem as a minimization optimization. Our goal is to find a strategy with minimal cost. Exchanges cost upload and download bandwidth and potentially disk storage. The relative costs of these resources are set by the parameters U (upload bandwidth), D (download bandwidth), and S (storage space).

Anytime c ships its data to another host it will incur a cost of both D and U. It is obvious why c would incur an upload coast, but it is reasonable to assume that it will also incur a download cost. Most nodes will not have the resources to prevent blocks sent by another host from reaching its network interface. Nodes can still choose to not store data, but before it can be discarded, it must be downloaded. We are most interested in when c will store data for others. c will incur a cost of S for each round that it stores data for h_i. We modeled S, U, and D as constant values.

Lastly, for exchange ϵ we represent c's decision to not store data for h_i as $\alpha_c = \omega_c$. This leads to the following cost function for c:

$$\text{cost} = \sum_{\epsilon} (E(\epsilon)(\omega_c - \alpha_c)S) + (U + D) \sum_{\epsilon} E(\epsilon)$$

Our objective is to minimize cost.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.3 Model Equilibria and Analysis

The ratio of the cost of storage (S) to the cost of upload and download bandwidth ($U + D$) and the ratio of the length of the grace period (G) to the maintenance period (P) affect c's incentives more than any other parameters. For the moment, assume that c only needs one replica ($R = 1$) and that I is the round by when the replica must be created. Later, we will examine strategies for arbitrary values of R.

Case: One Replica

The simplest case is when the cost of storing data for a grace period is greater than the cost of creating a new replica. Then c should never store data for others, leading c's strategy consisting of $\lfloor \frac{P}{G} \rfloor$ exchanges, $\{\epsilon_0, \epsilon_1, \ldots, \epsilon_{\lfloor \frac{P}{G} \rfloor - 1}\}$ where

$$\epsilon_i = [h_1, I + Gi, I + Gi, I + Gi, \min(I + G(i+1), T + 1)]$$

The cost to c of this strategy is $\lfloor \frac{P}{G} \rfloor (U + D)$.

The more interesting case is when $U + D > SG$. When this is true, it is helpful to imagine c spending storage to avoid paying network costs. c can use less expensive *storage events* (rounds when data is stored) to avoid more expensive *network events* (rounds when data is exchanged). Let $\mathcal{S}$ be the number of storage events and $\mathcal{N}$ be the number of network events, such that c's total cost is $\mathcal{S}S + \mathcal{N}(U + D)$.

The number of network events, $\mathcal{N}$, scheduled by c is determined by its storage events, $\mathcal{S}$, and the length of the grace period, G. c always incurs an immediate network cost of $U + D$ to create its replica during round I. Our approach will be to fix $\mathcal{N}$ and find the strategy with minimum storage costs, $\mathcal{S}$, for that value of $\mathcal{N}$. Strategy $s_{\mathcal{N}}$ denotes the minimum-cost strategy with $\mathcal{N}$ network events. Once we find $s_1, s_2, \ldots, s_n$, we can compare the costs of each to determine the optimal strategy.

If the grace period is long enough to last the entire maintenance period ($1 < P \leq G$), then $\mathcal{S} = 0, \mathcal{N} = 1$, and the final cost is $U + D$, which is the least possible cost. This strategy contains a single exchange: $\{[h_1, I, I, I, T + 1]\}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now say that the $G < P \leq 2G$. c has two competing strategies:

$$\begin{aligned} s_1 &= \{[h_1, I, T-(G-1), I, T+1]\} \\ s_2 &= \{[h_1, I, I, I, I+G], [h_1, I+G, I+G, I+G, T+1]\} \end{aligned}$$

In s_1, c holds on to data long enough to avoid a second network event so that $\mathcal{S} = P - (G-1)$ and $\mathcal{N} = 1$. Any other strategy with only one network event must involve a larger $\mathcal{S}$.

In s_2, c pays no storage costs, but pays network costs twice so that $\mathcal{S} = 0$ and $\mathcal{N} = 2$. Any other strategy with two network events incurs storage costs.

The optimality of s_1 or s_2 is determined by the relative costs of storage and network events. We denote this ratio r, such that $S = r(U+D)$. In terms of r, s_2 is optimal when

$$\begin{aligned} (P-G+1)S + (U+D) &> 2(U+D) \\ (P-G+1)r(U+D) + (U+D) &> 2(U+D) \\ (P-G+1)r + 1 &> U+D \\ r &> \frac{U+D-1}{P-G+1} \end{aligned}$$

and s_1 is optimal otherwise.

Next, say that the $2G < P \leq 3G$. c has three competing strategies:

$$\begin{aligned} s_1 &= \{[h_1, I, T-(G-1), I, T+1]\} \\ s_2 &= \{[h_1, I, T-2G+1, I, T-(G-1)], \\ &\quad [h_1, T-(G-1), T-(G-1), T-(G-1), T+1]\} \\ s_3 &= \{[h_1, I, I, I, I+G], [h_1, I+G, I+G, I+G, I+2G], \\ &\quad [h_1, I+2G, I+2G, I+2G, T+1]\} \end{aligned}$$

For s_1, $\mathcal{S} = P-(G-1)$ and $\mathcal{N} = 1$, as before. Any other strategy for which $\mathcal{N} = 1$ requires spending more storage. In s_2, $\mathcal{S} = P-(G-1)-G$ and $\mathcal{N} = 2$. Here c stores data long enough to only need two network events. It schedules a cheat toward the end of the game to avoid late storage costs. The other exchange involves holding onto data just long enough to gain G rounds of free-loading. Any other strategy for which $\mathcal{N} = 2$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	$2G < P \leq 3G$		$3G < P \leq 4G$	
Strategy	$\mathcal{S}$	$\mathcal{N}$	$\mathcal{S}$	$\mathcal{N}$
s_1	$P-(G-1)$	1	$P-(G-1)$	1
s_2	$P-(G-1)-G$	2	$P-(G-1)-G$	2
s_3	0	3	$P-(G-1)-2G$	3
s_4	NA	NA	0	4

Table 4.1: Summary of $\mathcal{S}$ and $\mathcal{N}$ for $2G < P \leq 3G$ and $3G < P \leq 4G$.

requires spending more storage. In s_3, c pays no storage costs, so $\mathcal{S} = 0$ and $\mathcal{N} = 3$. Any other strategy involving three network events must have non-zero storage cost.

If we apply similar reasoning to a network in which $3G < P \leq 4G$, we arrive at

$$\begin{aligned}
s_1 &= \{[h_1, I, T-(G-1), I, T+1]\} \\
s_2 &= \{[h_1, I, T-2G+1, I, T-(G-1)], \\
&\quad [h_1, T-(G-1), T-(G-1), T-(G-1), T+1]\} \\
s_3 &= \{[h_1, I, T-3G+1, I, T-2G+1], \\
&\quad [h_1, T-2G+1, T-2G+1, T-2G+1, T-(G-1)], \\
&\quad [h_1, T-(G-1), T-(G-1), T-(G-1), T+1]\} \\
s_4 &= \{[h_1, I, I, I, I+G], [h_1, I+G+1, I+G+1, I+G+1, I+2G+1], \\
&\quad [h_1, I+2G+2, I+2G+2, I+2G+2, I+3G+2], \\
&\quad [h_1, I+3G+3, I+3G+3, I+3G+3, T+1]\}
\end{aligned}$$

Table 4.1 summarizes $\mathcal{S}$ and $\mathcal{N}$ for strategies when $2G < P \leq 3G$ and $3G < P \leq 4G$. A pattern is apparent. When $iG < P \leq (i+1)G$ and $1 \leq j < i+1$ for strategy s_j, $\mathcal{S} = P-(G-1)-(j-1)G$ and $\mathcal{N} = j$. For strategy s_{i+1}, $\mathcal{S} = 0$ and $\mathcal{N} = i+1$.

This pattern holds for arbitrary i. To see why, consider how each strategy has been constructed. For strategies where $\mathcal{S} > 0$, we worked backwards from round T. The final exchange is a cheat to ensure that the last $G-1$ rounds do not incur any storage costs. This is why $G-1$ rounds are always subtracted from P (the length of the maintenance period).

Before this final exchange, each strategy schedules as many rounds of free-loading as possible. Each of these exchanges is for a grace period G. The more exchanges a strategy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

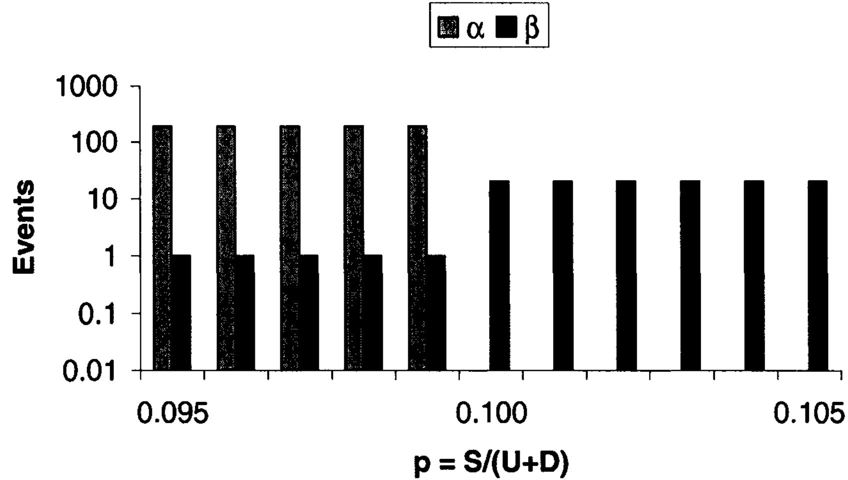


Figure 4.4: Optimal values of $\mathcal{S}$ and $\mathcal{N}$ for $P = 200$ and $G = 10$.

is allowed, the more of these can be scheduled. Hence, the subtraction of $(j-1)G$ from P. Only the first actually incurs any storage overhead and it is as short as possible.

Deciding which strategy is optimal requires the values of P, G, and r. Our original goal was to find the conditions when c would contribute storage. This is captured by the variable $\mathcal{S}$. In particular, we would like to know when $\mathcal{S}$ approaches P.

Figure 4.4 shows values for $\mathcal{S}$ and $\mathcal{N}$ for optimal strategy $s_{\mathcal{N}}$ under various values of $r = \frac{S}{U+D}$ in games where $P = 200$ and $G = 10$. The optimal strategy is to never store data until $r < \frac{1}{G}$. After that, obeying BEE is optimal with $\mathcal{S}$ approaching P. Thus, for $r < \frac{1}{G}$ in each of these games, c's optimal strategy is s_1. This is intuitive since larger values of of r indicate a larger relative cost of storage, which will make paying no storage costs (i.e. cheating) more attractive.

Case: Arbitrary Replicas

We will try to understand strategies under arbitrary values of R. Let I_i denote the round when replica r_i must be created. This is a generalization of starting round I from when $R = 1$. We want each I_i to be as late as possible to avoid unnecessary costs.

We can compute c's strategy by decomposing it into a set of individual replica strategies, $s_1, s_2, \ldots, s_R$. For each replica r_i with strategy s_i, its first exchange must occur in round $I_i = T - P - \lfloor \frac{i}{B} \rfloor$. This ensures that as many replicas are initiated as close to round $T - P$ as possible. It also ensures that all have been initiated by round $T - P$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

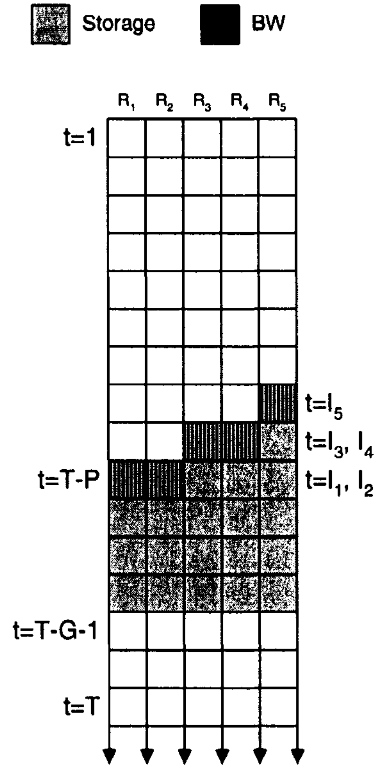


Figure 4.5: Schedule of strategies for $R = 5$, $G = 3$, $B = 2$, and $\mathcal{N} = 1$.

Our constraints require c to maintain R replicas from round $T - P$ to round T. The values of R and B affect when network events can be scheduled. If $B \geq R$, network events for different replicas will not interfere with each other. In this case, our analysis for $R = 1$ is still applicable since each s_i can be computed independently.

When $B < R$, scheduling strategies can be difficult. Computing each s_i as we did when $R = 1$ may result in conflicts. For example, to minimize storage costs, each s_i may include the exchange $[h_j, I, T - (G - 1), T - (G - 1), T + 1]$. Unfortunately, because $B < R$, only B strategies can include this exchange. Other replicas must be maintained some other way. Ideally, $\mathcal{N} = 1$ for all strategies, in which case the staggered start times is enough to avoid conflicts. As long as $r < \frac{1}{G}$, this is optimal. Figure 4.5 shows such a schedule.

However, there may be s_i for which $\mathcal{N} > 1$. In this case, strategies that are optimal in isolation may be infeasible in combination with others. Computing a globally optimal schedule from a set of locally optimal schedules is difficult. One way to resolve conflicts

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is to inject storage events into strategies to delay conflicting network events. c is forced to contribute storage to maintain its replicas. Resolving conflicts this way may not be optimal, but may be near optimal when storage events are inexpensive relative to network events. Resolving conflicts this way is also attractive since it is similar to our earlier analysis of bandwidth constraints in Section 4.3.

4.5 Expected Costs and Behavior

The final question we would like to answer is how would a typical Pastiche user behave? This depends on the relative cost of its resources. In the rest of this section we will see how observed resource costs affect nodes' strategic behavior as well as the cost of running Pastiche.

4.5.1 Strategic Behavior

Strategic behavior largely depends on the ratio of S to $U + D$, which we denoted r. Suppose that a typical user spends $0.64 per GB for storage [1] that will last 290 days [17]. This means that access to a GB of data for a day costs $0.002.

Also suppose that a typical Pastiche user pays $60/month for cable modem service that provides 4 Mbs downstream and 384 Kbps upstream.[2] This means that our user can download data for approximately $0.05 per GB/day and upload data for approximately $0.50 per GB/day with a maximum capacity of 42 GB/day downstream and 4 GB/day upstream. However, most bandwidth is unused and any excess resources used by Pastiche are a sunk cost. Thus, the opportunity cost of using bandwidth is much less expensive.

To characterize typical bandwidth consumption, we can use the observations of a network usage study of the University of Washington campus [100]. This study found that the average web user downloaded 4 MB/day and that the average Kazaa user downloaded 40 MB/day and uploaded 400 MB/day. It was surprising to find that the Washington students uploaded more data than they downloaded, but it conforms to suspicions that campus networks essentially act as file sharing servers for the rest of the Internet. Nonetheless, if we

[1] As of August, 2005, a 250GB disk costs $160.
[2] As of August, 2005, this is what Comcast's Internet service cost in Ann Arbor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

assume that this usage is typical of a Pastiche user, then users have 41.9 GB/day of excess downstream bandwidth and 3.6 GB/day of excess upstream bandwidth.

If we also assume that a user has 32 GB of state, then the opportunity cost of creating a *single* replica is approximately \$0.20 per day for upstream bandwidth and zero for downstream bandwidth over eight days. This is because 400 MB/day is 10% of the total upstream capacity and interfering with this used capacity would cost 10% of the total upstream cost of \$2 per day.

Since it takes eight days to upload 32 GB, the cost of uploading a single replica would be approximately $8 \times \$0.20 = \$1.60$. The ratio of S to $U + D$ would thus be $r = (32 \times \$0.002)/\$1.60 = 0.04$. As we found in Section 4.4.3, if $r < \frac{1}{G}$ contributing storage is optimal. This means that as long as grace periods are shorter than 25 days, we can expect Pastiche nodes to contribute storage.

4.5.2 Monthly Cost of Running Pastiche

One of Pastiche's primary goals is to provide a low-cost backup service. Using this model we can calculate a conservative estimate of a user's network and storage costs to run Pastiche under BEE. As with our model, our estimate ignores resources other than storage and network bandwidth.

We will assume that the cost to upload a 4 GB of data costs \$0.20, the cost to download 4GB of data is free, and the cost to dedicate 4GB to Pastiche over the lifetime of the disk is $4 \times \$0.64 = \$2.56$. As before, the cost of storage is based paying \$160 for a 250GB disk.

Suppose that nodes use five buddies at any moment in time. We must also account for other hosts failing. If we assume a constant failure rate, then, on average, approximately two buddies will fail before a node itself fails. This means that over the lifetime of a disk, nodes will *create* about seven replicas, on average. This does not affect a node's storage costs, which remain constant, but does affect bandwidth its costs.

Also, we must account for the cost of participating in other users' recoveries (including fire-drills). This depends on the lifetime of the disk and node availability. The longer the disk lasts, the more fire-drills a node will have to participate in. If we conservatively

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
\text{cost to upload 4GB} &: \quad c_u = 0.20 \\
\text{cost to store 4GB} &: \quad c_s = 2.56 \\
\text{buddies at a time} &: \quad b_{ins} = 5 \\
\text{buddies over time} &: \quad b_{tot} = 7 \\
\text{mean time to failure} &: \quad m = 290 \\
\text{time between fire--drills} &: \quad f = 60 \\
\text{node availability} &: \quad a = 0.70 \\
T(d) &= \frac{d}{4}\left(c_s + c_u b_{tot} + c_u \left(\frac{m}{fa}\right)\right)
\end{aligned}
$$

This shows a formula to compute the cost of running Pastiche and realistic parameters. The formula computes the cost to store data, the cost to create replicas, and the cost to participate in buddies' fire-drills and restores.

Table 4.2: Cost formula.

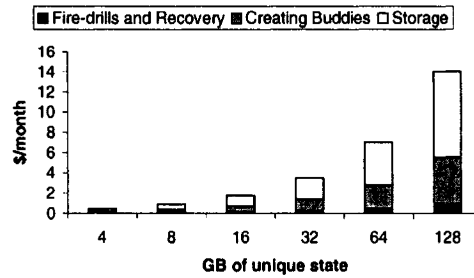
estimate that a disk will last 290 days [17] and that fire-drills are scheduled every 60 days, then a node can expect to participate in 5 fire-drills per buddy.

The greater node availability is, the less burden is placed on each available buddy during a recovery. If all five buddies are available, then each individual's burden is one-fifth. A study of the Overnet peer-to-peer file sharing system found that the median node availability was approximately 70% [11]. Thus, we will assume that $0.7 \times 5 = 3.5$ buddies will be available for each recovery. These assumptions lead to the function T, for the total cost, $T(d)$, of running Pastiche with d GB of unique backup state, shown in Table 4.2. Figure 4.6 plots $T(d)$ for several state sizes using the parameters in Table 4.2.

Figure 4.6 shows that the cost of running Pastiche is low. Backing up 10GB of unique state costs less than one dollar per month for a period of 290 days. While this number is low, it also represents a conservative estimate. In particular, we assumed that disks last only 290 days before failing. If we assume that disks last two years, users can backup up to 24GB for less than a dollar per month. This is because doubling the time to run Pastiche cuts the per month cost of running Pastiche in half, since the initial storage investment can be amortized over a longer period.

This demonstrates that initial storage investment is by far the largest cost. This is very encouraging. The cost of disk storage has been falling exponentially for twenty years and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure plots the cost of running Pastiche per month for several backup state sizes for the parameters in Table 4.2. Pastiche costs less than one dollar per month for state sizes of less than about 10GB.

Figure 4.6: Cost of running Pastiche.

should continue to do so for as least the next five [78]. At the same time, storage hardware represents less than ten percent of the overall cost of centralized solutions [78]. Since Pastiche has no administrative overhead, it will continue to benefit from falling storage costs while centralized services will not.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

STORAGE CLAIMS

Pastiche must ensure that nodes contribute in proportion to their consumption. Under bilateral, equal exchange (BEE), node A provides storage for node B if and only if B also provides storage for A. B can periodically check its data to see if it is held by A, and vice versa. Collectively, these pairwise checks ensure that each node consumes as much as it contributes.

BEE requires what economists call a *double coincidence of wants* [87]. Nodes can only use the storage of those who want an equal amount of theirs. Double coincidences of wants may be rare in Pastiche since overlap is asymmetric and users have a wide range of storage demands. Because of this, BEE could over-constrain storage allocation.

Consider two nodes with different storage needs. Node A might want to store 1MB on B and node B might want to store 1GB on A. One way to solve this problem is for B to trade a 1MB subset of its state for A's 1MB. Unfortunately, because Pastiche currently only replicates data at a file system granularity, such exchanges are not supported [1].

Thus, under the assumption that backup state is replicated monolithically, Pastiche addresses problems of asymmetric demand with *storage claims*. Storage claims are incompressible storage placeholders. They are *content-immaterial* objects whose sole purpose is to enforce equal exchange by occupying space. Claims allow nodes to manufacture equal exchange when it does not arise naturally.

Once stored, claims are treated like any other data object under BEE; nodes can query

[1] As discussed in Section 3.6, while state-splitting is not impossible, it requires additional information to be added to Chunkstore meta-data chunks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and discard their content just as before. Claims need two properties to act as placeholders. First, they must be incompressible. This forces claim recipients to expend the intended amount of storage space. Second, claims' owners must be able to easily recreate them from a small amount of state for querying. Otherwise, claims could add to their owners' storage burden.

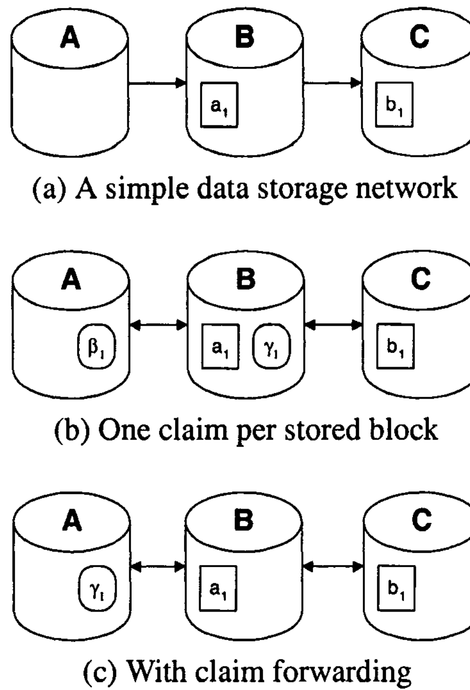
Returning to our example, A can pad out its 1MB of actual data with 999MB of claims to meet B's demand for 1GB. This assumes that A is willing to give up more of its disk than it has data. A's incentive to agree to this trade depends on the value to A of creating a replica at B relative to other options. The value of storing 1MB of data and 999MB of claims at B in exchange for 1GB of local storage must be greater than the value of any alternative exchanges, possibly including those with lighter storage requirements. Understanding how nodes weigh the benefit of a host's network proximity and data overlap versus the cost of its storage demand is beyond the scope of this document. From now on, we will remain agnostic about how *preferred* hosts are chosen and will simply assume that they present the best combination of attributes.

Extreme asymmetry occurs when one host wants to store data on a host with no reciprocal interest. In the worst case, storage interest is never mutual. Because Pastiche nodes want most of their buddies to be collocated, this worst case is unlikely. However, finding far away buddies may resemble this worst case.

When there is nothing to gain from storing data at another host, there is no incentive to negotiate an exchange. For example, if node B wants to store 1GB on node A, but A does not want to store any data on B, what incentive does A have to send B storage claims? Sending claims to B would cost A bandwidth and maintenance overhead for little or no value in return. If Pastiche cannot give nodes value in return for shipping storage claims, only nodes with mutual interest will trade storage. This prevents most nodes from storing at their preferred sites and over-constrains storage allocation.

Alternatively, a node could accept storage requests from arbitrary hosts and ship its backup state instead of claims. This may be an attractive option when willing buddies are difficult to find as a way to create temporary replicas. Unfortunately, if a node cannot find any willing buddies, it will be left using unsatisfactory storage.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) A simple data storage network

(b) One claim per stored block

(c) With claim forwarding

This figure illustrates a simple data storage network. The top panel shows only the blocks stored by each node. Node B stores block a_1 for node A, and C stores b_1 for B. The middle panel shows the network with the addition of storage claims. Node B must store claim γ_1 for node C, and A must store β_1 for B. In the bottom panel, node B forwards claim γ_1, rather than generating its own. This reduces storage overhead, but creates a dependency chain; B's data now depends on the faithfulness of A.

Figure 5.1: Storage claims

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ideally, a node could identify its preferred replica sites and use that storage. To do this, nodes can overwrite claims they own with claims they receive. This is called *forwarding*. Forwarding allows nodes to treat claims as a *store of value*. Claims are a kind of credit for contributing storage that can be cashed in during later exchanges. The ability to hold credit gives claims value to their owner and creates an incentive to own them.

Forwarding requires creating data dependencies. For example, in Figure 5.1(c), forwarding creates the dependency chain $C \rightarrow B \rightarrow A$. This has implications for data reliability, because nodes who forward claims are still responsible for them. When a downstream node fails, upstream nodes are punished. However, if a claim is forwarded back to its creator, it forms a dependency cycle. Cycles tolerate single failures without cascading loss.

The rest of this chapter is organized as follows. Section 5.1 describes how and when claims are constructed. To create a claim, a node only needs a secret key K and the on-disk location of the object it was traded for.

Section 5.2 describes the forwarding protocol and how it enables flexible storage allocation. Forwarding takes advantage of two abstractions. The first is *storage rights*, which allow data to be overwritten. The second is claims' content-immateriality, which allows them to be over-written without consequence.

Section 5.3 discusses the data dependencies created by forwarding and how they can threaten overall reliability. The longer a dependency chain becomes, the more likely that much of the data along the chain will be discarded. However, if a claim is forwarded back to its original owner to form a cycle, nearly all data can be preserved in the event of a failure.

Finally, Section 5.4 evaluates our prototype for managing claims and shows the results of simulations of forwarding under various failure and storage utilizations settings. We found that the overhead incurred by our prototype is low and that forwarding only leads to data loss during large-scale, correlated failures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1 Claim Construction

Storage claims transform each storage request into an equal exchange. To preserve these terms, nodes that are given claims must only be able to produce them if they are physically stored. Claim owners, however, must not be forced to store local copies of claims in order to verify that they are being honored. This would double the storage burden of the owner.

Before a node joins the system, it must initialize its storage space by logically filling it with storage claims. Later, when data is placed in the storage space, the node can return any claims that were "overwritten."

Computing a claim requires two values— a private, symmetric key K, and a location in the storage space. To initialize the space, nodes first logically fill the storage space with hash values. In the first 20 bytes, nodes compute the SHA-1 hash of the concatenation of the symmetric key K and the number 0, denoted $h_0 = SHA - 1(P, 0)$. For each 20 byte chunk that follows, the hash $h_i = SHA - 1(P, i)$ is stored in the ith chunk.

Claims are fixed-sized blocks formed from consecutive hash values. For example, suppose that claims are 512 bytes long. The initial claim, denoted C_0, is computed by concatenating the first 25 hashes with the first 12 bytes of the 26th hash, and then encrypting it using the symmetric key K. The second claim, C_1, is computed by encrypting the concatenation of the next 25 hashes and the first 12 bytes of the hash after those, and so forth. Thus, the $i$the claim, C_i, is :

$$C_i = \{h_j, h_{j+1}, \ldots, h_{j+24}, h_{j+25}[0], \ldots, h_{j+25}[11]\}_K$$

where $j = i \times 26$.

Claims do not have to be computed during initialization, and never need be stored on the originating host. They can be computed on the fly as needed using K and the associated storage location.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Forwarding Claims

When node A stores data for node B and B stores claims for A, we say that B is *downstream* of A; claims always move downstream. Trades between A and B represent an exchange of *storage rights*. Each node is entitled to store anything it chooses on the remote disk blocks of the other. A can replace its object with anything and B can replace its claims with anything. Replacement does not affect equal exchange since all nodes still consume and contribute equally.

To authenticate replacement, the two nodes negotiate a Diffie-Hellman key [38] during the initial trade. This is unauthenticated because we do not assume that nodes have certified identities. It is therefore subject to man-in-the-middle attacks. However, if the initial exchange is not intercepted, future replacement requests can be checked for authenticity. Pastiche decreases the likelihood of interception by routing negotiation messages via IP rather than the overlay.

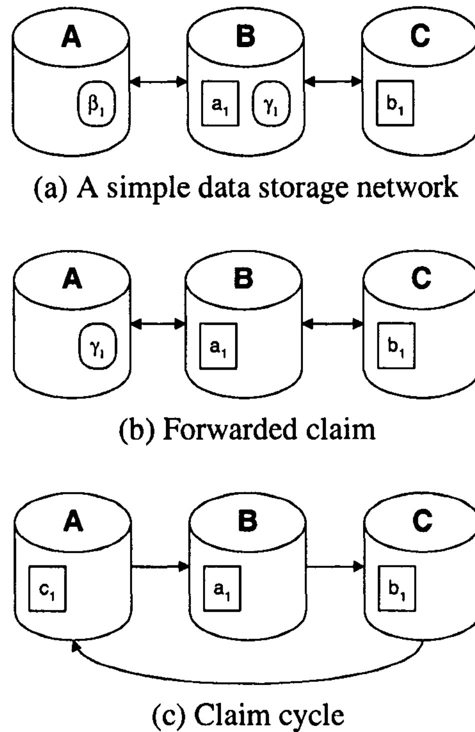
Unfortunately, claims ownership is not valuable enough to encourage trade between arbitrary nodes. Claim content itself has no inherent value. They are *content-immaterial* because they do not help during recovery. Claims are only useful insofar as they verify that their owner's storage rights are being respected.

Because of this, there is no incentive for a node to spend valuable local storage or bandwidth to ship data that cannot help it recover. Worse, creating claims may actually weaken the nodes ability to recover from disk failure. Consider a node whose disk is one-sixth full and who wants five backup buddies. Accepting data requests in return for claims limits the node's ability to swap data with its buddies. Given these trade-offs, nodes should only store claims from their buddies. This limits allocation to transactions between hosts with mutual interest.

To enable more flexible allocation, Pastiche must compensate nodes for the disincentive to ship claims to hosts they have no interest in. It does this by allowing nodes to overwrite claims they own with claims they are responsible for. Moving claims further downstream this way is called *forwarding*.

Forwarding makes claims useful to their owner. With forwarding nodes can treat

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) A simple data storage network

(b) Forwarded claim

(c) Claim cycle

This figure illustrates how a claim is forwarded in a simple three node storage network. Arrows represent query lines. Objects are boxes and claims have rounded corners. Node C is storing object b_1 for node B. B is storing claim γ_1 for C in return. B is also storing object a_1 for node A, who is in turn storing claim β_1 for B. Because B wants to free up space so that it can create a second replica of b_1, it replaces β_1 on A with γ_1 and forwards any queries or storage requests for γ_1 from C to A. Say that C wants to create a replica of object c_1 on A. This allows A to replace whatever new claim it would have returned to C with claim γ_1, which removes γ_1.

Figure 5.2: Forwarding claims.

claims as a *store of value*. This is similar to how fiat money operates. Nodes will be willing to accept data requests in exchange for claims if the storage occupied by claims can be leveraged in later transactions. Used this way, claims function as a kind of currency, freeing storage allocation and eliminating the need for direct, mutual interest.

Forwarding has the additional advantage of clearing the storage overhead of claims. When one claim is overwritten with another, their is one less claim occupying space in the storage pool. This overhead can be significant. In the simple case where nodes never replace the claims they create, the storage overhead of Pastiche is equal to the data stored in the storage system, doubling global storage requirements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider the network shown in Figure 5.2(a). Node A stores its object a_1 on node B, which in turn stores claim β_1 on A. B also stores object b_1 on node C, which in turn stores claim γ_1 on B. B only has space for two blocks—both of which are currently occupied—but wants to create a second replica of b_1. Creating a second replica is possible only if B has enough space to store another node's claims. Thus, B needs to free a block of storage.

B can free a block by using the storage rights it received for storing a_1, so B replaces β_1 with γ_1. B no longer needs to query for β_1. Whenever C queries B for γ_1, B will simply forward the query to A. This network is shown in Figure 5.2(b).

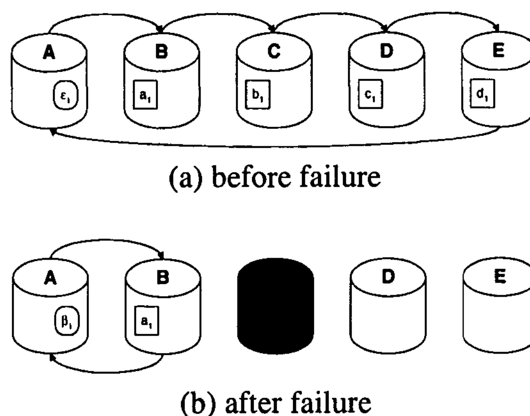
Allowing nodes to forward claims does not enable the Sybil attack [41]. Suppose a dishonest node creates an alias for itself. When it receives a claim, it can "forward" the claim to its alias, but the node gains no advantage. It must still produce the claim when asked, and must therefore still store it.

B was able to eliminate one claim, but another claim, γ_1, still exists. Now say that C wants to create a replica of its object c_1 on A. A first stores c_1 for C and returns claim γ_1 to C, rather than creating a new claim. When γ_1 is forwarded back to its owner, C, a cycle is created, and γ_1 can be removed. This network is shown in Figure 5.2(c).

Even when a cycle is created, each node's storage rights remain intact. C still has the right to a block on B, B still has the right to a block on A and A has the right to a block on C. In a cycle, every node can forward a replacement request to a downstream node until the request comes back to the node that originally made it.

The cycle example, as presented, uses the simple optimization of telling nodes where their claims are physically stored. Without the optimization, C would initially receive a brand new claim, α_2, from A. C would then use its storage rights on B to forward α_2. B would in turn forward α_2 to A, who does not need to store the claim because it can compute it. The result is the same—all claims are removed. When C knows that A is storing γ_1, all of this forwarding can be eliminated. C can simply tell A to return γ_1 instead of creating new claim α_2. This optimization has the additional benefit of eliminating query forwarding since nodes can query claim storers directly.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) before failure

(b) after failure

This figure illustrates a five-node dependency chain of claims, both before and after the middle node fails. Arrows represent query lines. Objects are boxes and claims have rounded corners. Upstream nodes lose all data, while downstream nodes retain it.

Figure 5.3: Failure in dependency chains.

5.3 Forwarding and Reliability

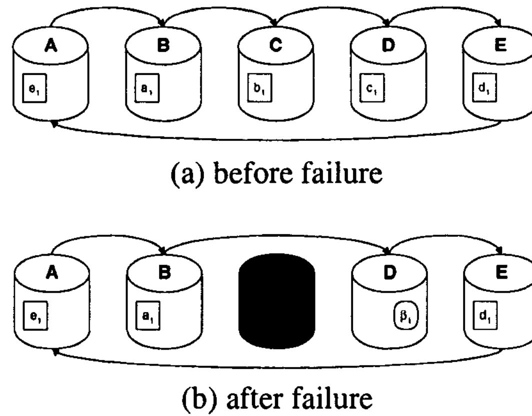
When a node forwards a claim downstream, it remains responsible for it. Without authenticated identities, nodes cannot prove to the claim owner that the claim was forwarded. To see why, consider a dishonest node that can create aliases for itself. Every claim received by this dishonest node could be "forwarded" to one of its fictitious identities. If other nodes believed these assertions, the dishonest node would never be held accountable for discarding the claims it should have been storing.

Pastiche assumes that nodes respect the storage rights they have promised other nodes when their own data depends on it. In the case where claims are never forwarded, the rules of BEE apply. Whichever node's query failed must be storing data owned by the failed node. The querying node simply discards the failed node's data.

When claims are forwarded, however, the node that detects the failure might not be storing any data owned by the failed node. Consider the five node dependency chain in Figure 5.3(a). In this network, a dependency chain exists via claim ϵ_1. Node A is downstream of node B, B is downstream of node C, C is downstream of node D, and D is downstream of node E.

Suppose that C fails B's query. B is not storing c_1, so it cannot discard C's data.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) before failure

(b) after failure

This figure illustrates a five-node dependency cycle of claims, both before and after the middle node fails. Arrows represent query lines. Objects are boxes and claims have rounded corners. Upstream and downstream nodes retain all data.

Figure 5.4: Failure in dependency cycles.

However, B still has storage rights on A, so it uses those rights to replace ϵ_1 with its own claim, β_1. A must replace ϵ_1 with β_1 to protect its object, a_1.

Meanwhile, E holds D responsible for ϵ_1. When E queries D for ϵ_1, the query fails. E then discards D's object, d_1. In response, D attempts to use its storage rights at C to replace ϵ_1 with a claim of its own. When this request fails, D discards C's object, c_1. The resulting network is shown in Figure 5.3(b). All data upstream of the failure is lost, but downstream data is preserved.

In our example, even though D did not fail, it lost d_1 as a result of C's failure. Forwarding a claim makes the data traded for that claim only as reliable as the weakest downstream node. The longer the dependency chain, the less reliable each upstream node's data becomes.

This weakens claims' function as a store of value, since they are valuable in proportion to their reliability. The further a claim is forwarded the less valuable it becomes. Nodes could agree to limit chain lengths, although this is not enforceable. Capping chain lengths also means that the node at the end of a chain will lose some incentive to create claims of its own. Even though dependency chains weaken reliability, as we show in Section 5.4.4, object loss is still rare, unless both the permanent failure rate and storage utilizations are very high.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Surprisingly, while dependency chains weaken reliability, cycles actually restore it. This is because each node in a cycle has storage rights on another node in the cycle. In a dependency chain, the node farthest downstream does not have any storage rights. When the farthest node does have storage rights, data can "wrap around" any failure. Consider the five node dependency cycle in Figure 5.4(a). In this network, a cycle exists where node A is downstream of node B, B is downstream of node C, C is downstream of node D, D is downstream of node E, and E is downstream of A. Suppose, as before, that C fails B's query. B again uses its storage rights to store claim β_1 on A.

Before, A replaced ϵ_1 with β_1. Now, however, A has storage rights on E, because A is storing E's object e_1. A can use those rights to forward β_1 to E. E has storage rights on D and thus forwards β_1 to D. D has storage rights on C and attempts to store β_1 on C. This fails, so D discards C's object, c_1. However, because D wants to protect its object, d_1, on E, it stores β_1. The resulting network is shown in Figure 5.4(b). This time all data upstream and downstream of the failure remains intact.

This puts Pastiche in a strange predicament. Forwarding weakens data reliability as long as the claim remains active in the system, but if the claim can be retired to create a cycle, reliability is restored. It is thus in nodes' interest to create cycles, but not chains. We will return to this issue in Chapter 6.

5.4 Evaluation

We have built a prototype claims management application in C, called `samsarad`. The name `samsarad` is a remnant of the original name of this research, Samsara [35][2] The daemon is composed of three layers—a messaging layer, a replica manager, and a storage layer.

The messaging layer is responsible for sending and receiving all network and local messages. Local messages are passed via Unix domain sockets. This is how Pastiche communicates with the claims management software. There are four messages that can be sent to `samsarad`—`store`, `retrieve`, `query`, and `callback`. The arguments and

[2]Samsara is the name of the cycle of good and bad karmas in Hinduism.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Message	Arguments	Return Value
store	*pathname, objectid, location, sync*	*error*
retrieve	*objectid, location, sync*	*pathname*
query	*location*	*error*
callback	*pathname, event*	*id*

(a) samsarad messages.

Event	Description
store_req	a remote node requests storage
retrieve_req	a remote node requests data
query_res	a query succeeds or fails

(b) Callback events.

This table shows the messages that can be passed to the samsarad daemon by higher level peer-to-peer storage systems, like Pastiche. Each message is passed to samsarad via Unix domain sockets. The callback message allows other processes to register a socket to be notified when an event occurs. The events that can be registered for are listed and described in the bottom table.

Table 5.1: samsarad interface.

return values for these messages are summarized in Figure 5.1. All network communication uses the RPC2 package [102].

Once the messaging layer receives a message, it forwards the request to the replica manager. The replica manager is responsible for authentication and maintaining replica locations. We used the openssl0.9.7 library for all authentication including a 2048-bit Diffie-Hellman key exchange with SHA-1 hashes.

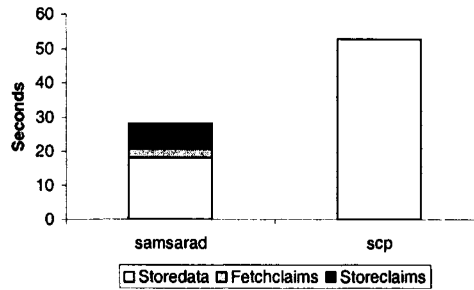
The storage layer is responsible for knowing who owns any stored data and where that data lies; it also handles claims generation, using a claim size of 4096 bytes.

Claims, like chunks, are stored in the underlying ext2 file system. Claims are computed from a user-defined password and the i-node number of the chunk it was traded for. Chunk locations are stored in a database that maps object identifiers to a ext2 path. Additionally, all stores are whole-object grained.

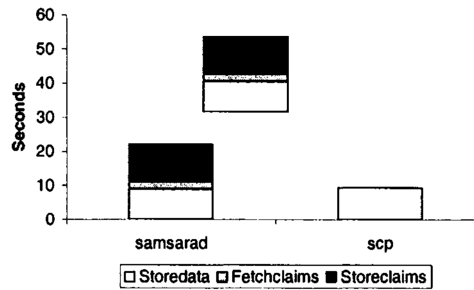
In evaluating claims forwarding, we set out to answer the following questions:

- What are the data transfer and query performance of the prototype?
- How much processor and disk overhead does querying induce?
- How does storage utilization affect the need to forward?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Time to complete tree benchmark.



(b) Time to complete archive benchmark.

This figure shows the time to transfer both the tree and archive loads for samsarad and the copying utility scp. Our prototype requires three phases to complete the transfer—storing the data at a replica site, receiving claims in return, and then storing those claims.

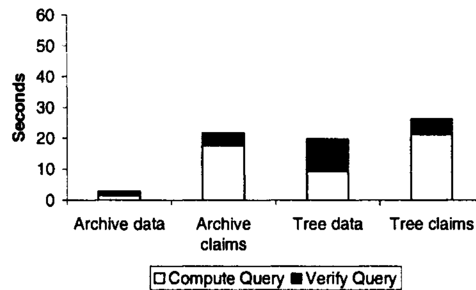
Figure 5.5: Data transfer performance.

- How often do dependency cycles arise naturally?
- How long do dependency chains get as utilization increases?
- How do dependency chains affect reliability?

5.4.1 Prototype Micro-benchmarks

We were interested in measuring two aspects of our prototype, samsarad. We wanted to know what its data transfer and querying performance were and we wanted to characterize the load of responding to and verifying queries. To explore both questions, we used two representative data loads. The first load, referred to as the "tree" load, consisted of 1676 files totaling 13MB. The other load, referred to as the "archive" load, consisted of a single 13MB file. We chose these two loads to examine the effect of transferring and querying many objects versus just one. The data used for both was taken from the openssl-0.9.7a source tree. The tree load was all regular files in this source tree,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure shows the time to compute and verify a query under both the tree and archive loads. Times to compute and verify queries for both the data and the claims exchanged are shown.

Figure 5.6: Query performance.

while the archive load was all regular files concatenated into a single file.

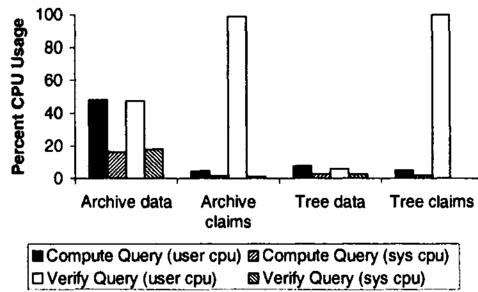
All experiments were run on machines with a 550 MHz Pentium III Xeon processor, 256MB of memory, and a 10k RPM SCSI Ultra wide disk. The disk has an average 4.7 ms seek time, 3.0 ms rotational latency, and 41 MB/s peak throughput. Test-bed machines are connected by a switched, 100Mb/s Ethernet fabric.

We first measured data transfer performance. How does `samsarad` perform relative to the commonly-used secure copy utility `scp`? In this experiment, `samsarad` first copied data to a replica site. It then retrieved and stored the claims exchanged for that data. We ran ten trials per experiment with both `samsarad` and `scp`, being careful to flush the system's buffer cache between trials. The performance results are in Figure 5.5.

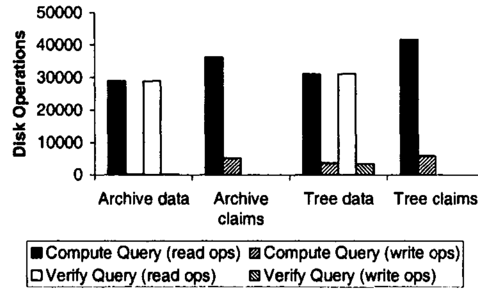
`samsarad` outperformed `scp` for both loads during the store data phase, with `scp` performing much worse under the tree load. Storing claims, however, is quite slow, taking nearly 10 seconds to complete. The reason for this is that `samsarad` needs to store each claim, and then update the claim's entry in the storage location database. While slow, this performance is still reasonable since most peer-to-peer storage systems ship data asynchronously.

We evaluated query performance and overhead next. We queried the data and claims associated with the tree and archive loads. This involves two phases—the query response computation and verification. The results are shown in Figure 5.6. These measurements were taken after the data had been copied and the buffer caches at both machines had been flushed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) CPU load during querying.



(b) Disk operations during querying.

This figure shows processor and disk load during query computation and verification.

Figure 5.7: Query processor and disk load.

For both data queries, the time to compute the response was the same as the time to verify it. This is because both the querying and storing node performed the same tasks. Each needed to read the data from disk to compute the response. It took much more time to compute and verify the tree data, because it required more disk reads. For claims queries, the time to compute a response took much longer than the time to verify. This is because verifying a claims query is done entirely in memory, which is much faster than reading from disk.

The processor and disk loads of computing and verifying queries are shown in Figure 5.7. The processor and disk loads of the querying and responding nodes are the same for data, but the overhead of responding to a claims query is very different from verifying it. The responding node uses many more disk operations to read the claims from disk, while the querying node requires more processor time to recompute the claims.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4.2 Space Overhead

In the worst case, when nodes never forward claims, Pastiche doubles the global storage burden. As we saw in Section 4.5.2 this would have a significant impact on the cost of running Pastiche, nearly doubling the monthly cost of participation. Furthermore, this could be problematic when storage utilization is high. To explore how effectively Pastiche can reduce its overhead under high utilization, we built a Pastiche simulator on top of SimPastry. We populated our simulator with 5,000 nodes, and set the replication factor to four. Each node was assumed to have one object that was equal in size to a claim.

We were also interested in the relationship between replication strategy and overhead. Objects can be replicated as individual objects in a distributed hash table (DHT), or can be replicated as collections on nearby nodes. The former strategy is employed by CFS [36] and PAST [98], while latter strategy is similar to that used by Pastiche. For both replication strategies, we varied system-wide space utilization from 50% to 100%. In each experiment objects were forwarded to other nearby nodes, as determined by replication strategy, if their "natural" replication sites were full. Each experiment continued until all objects were placed or storage was exhausted.

The results are in Table 5.2. Nodes were almost always able to replicate their objects, even at 100% storage utilization. In the 100% utilization case, both collection-based and DHT replication reduced claim overhead to less than 0.02% of global storage.

Even though collection-based and DHT replication pursue very different strategies, claim overhead can be effectively reduced in both. The biggest difference between collection-based and DHT replication is that collection-based generated more cycles than DHT. The reason for this is that collection-based nodes are more likely to choose one another. The collection-based strategy targets replica sites that are close in the network, which is generally symmetric. This leads to clustering of nodes that are close to each other, and makes cycles more likely. In DHT replication, nodes replicate along a random range of the nodeId space. The randomness makes it highly unlikely that nodes within a range of the nodeId space will choose to replicate their own objects on reciprocal peers.

Still, for both replication strategies, we observed very few cycles. Because chains

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Collection-based replication					
	50%	57%	67%	80%	100%
replicas	20000	20000	20000	20000	19997
claims	13626	11488	8619	4740	3
cycles	86	129	173	180	152
forwards	6326	8528	11942	18913	131360
DHT replication					
	50%	57%	67%	80%	100%
replicas	20000	20000	20000	20000	19998
claims	15207	12753	9168	9168	2
cycles	5	7	4	6	37
forwards	4788	7269	11332	18050	119833

This table shows the storage overhead in two simulated Pastiche networks of 5000 nodes, each needing four replicas. Each column represents the percent of global storage needed to store all replicas.

Table 5.2: Simulated space overhead.

Collection-based replication			
	80%	50%	100%
total claims	14971	10671	9098
% with length $\leq$4	1	.88	.68
% with length $\leq$8	1	.97	.79
DHT replication			
	80%	50%	100%
total claims	16529	11126	9775
% with length $\leq$4	1	.81	.62
% with length $\leq$8	1	.98	.80

Table 5.3: Chain length distributions.

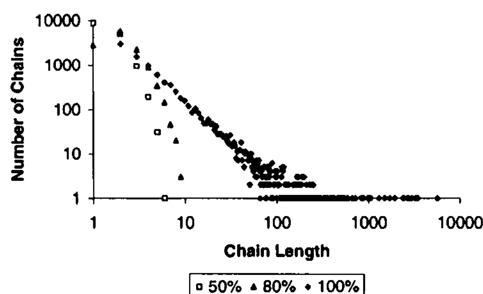
rarely become cycles, forwarding claims will almost always weaken data reliability.

5.4.3 Chain Length Analysis

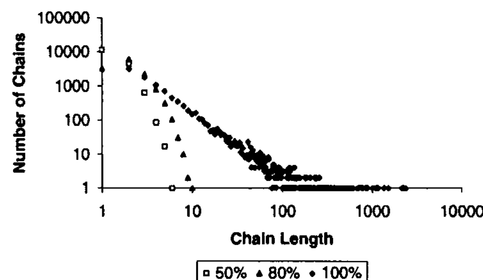
Pastiche can remove enough claims through forwarding to ensure that all nodes are able to create replicas, no matter the storage availability, but how long do chains become? Furthermore, at what length do they threaten hosts' ability to restore their files? Figure 5.8 shows the distribution of chain lengths under both collection-based and DHT replication. These results are summarized in tabular form in Table 5.3.

Chain lengths grow beyond eight only when approaching 100% utilization. In the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Distribution of chain lengths for collection-based replication.



(b) Distribution of chain lengths for DHT replication.

This figure shows the distribution of chain lengths for networks using collection-based and DHT replication. Each series represents a percent storage utilization required to create all replicas.

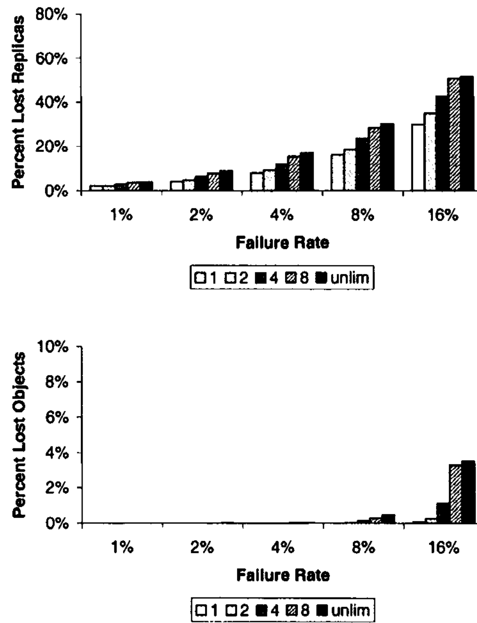
Figure 5.8: Chain length distributions for forwarding claims.

100% utilization case, 69% of chain lengths are less than four and 79% are less than eight. Interestingly, chains can become extremely long when utilization requirements are 100%. In one experiment, for example, we witnessed a chain that was 3,822 nodes long out of 5,000.

We must know two things to explain why this happens—how and under what circumstances chains become longer. When a claim is forwarded, its chain lengthens by the size of the chain it replaces. The sum of all chain lengths increases when claims are introduced, remains unchanged when claims are forwarded, and declines through cycles.

Few new claims are introduced beyond a certain point, cycles are rare, and storage is cleared almost exclusively by forwarding claims. What this means is that the sum of all chain lengths among active claims remains relatively constant. Because this sum remains stable, the fewer active claims there are, the longer their chains necessarily become. In addition, when storage utilization is 100%, there are between four and five times more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure shows the percent of lost replicas and objects when chain lengths are capped using aggressive claims forwarding. An object is lost only when all of its replicas are.

Figure 5.9: Reliability of data with capped chain lengths.

forward operations then when utilization is 80%. Forwarding reduces the number of claims without changing the sum of all lengths, which leads to longer chains.

5.4.4 Reliability

Chain length is important because it has implications for reliability. The reliability of data stored along a chain depends on downstream nodes not failing. Because of this, it is important to explore the relationship between storage utilization, chain length, and data reliability.

To evaluate reliability in the face of permanent node failure, we constructed a 5000-node network using DHT replication. Once constructed, we assumed that nodes *permanently* and *simultaneously* failed with some probability. This eliminated the failed nodes' objects along with any objects stored on other nodes in exchange. If any lost objects were forwarded claims, all upstream data was lost, as described in Section 5.3.

Our first set of experiments isolated the effect of chain length on reliability. To do this, we gave nodes unlimited storage capacity and forwarded claims aggressively, up to a limit on the number times claims could be forwarded. The results of these experiments are in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 5.9. It shows both the fraction of lost replicas and the fraction of lost objects; an object is lost only when all of its replicas are.

We are interested in the percent of lost replicas beyond those lost in the base case, where forwards are capped at one. The number of lost replicas for the base case reflects both the objects stored *at* the failed node and the objects stored *in exchange* for the claims stored at the failed node.

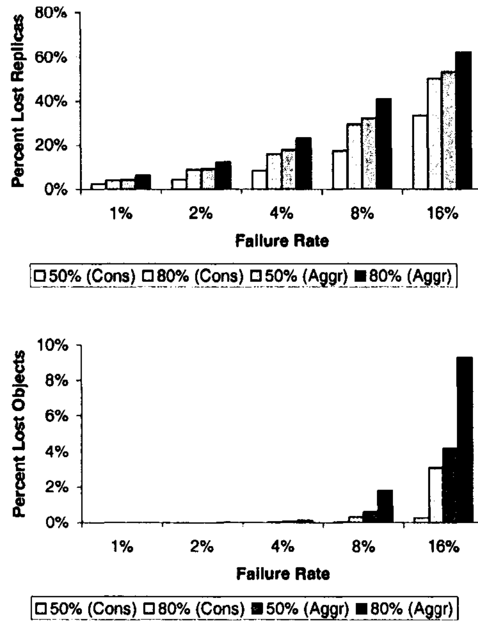
There is a significant drop in reliability between chains of length four and eight. For a permanent failure rate of 8%, when chain lengths are limited to four, only 0.13% of objects are lost, whereas when chain lengths are capped at eight, 0.30% of objects are lost. When the permanent failure rate is 16%, 1.1% of objects are lost when chain lengths are four or less, while 3.3% are lost when chain lengths are eight or less. Because forwarding claims only when necessary keeps almost all chain lengths below four, it will be reliable even with widespread failure.

Our second set of experiments focused on the effect of storage utilization on reliability. When storage utilization is low, nodes hold onto all claims they receive, rather than endanger their data by forwarding them. When storage utilization is high, nodes are forced to forward them. To understand this interaction, we examined a range of failure rates for storage utilizations of 50% and 80% and forwarded claims both conservatively and aggressively. The conservative trials represent expected reliability, while the aggressive trials represent worst-case reliability. The results are in Figure 5.10.

These figures show that nodes should always forward claims conservatively. Forwarding claims aggressively is no more effective at clearing space for replicas at high utilizations, but increases the probability that an object will be lost several-fold. The figures also show that excess storage can be adaptively occupied to increase reliability, or cleared to create space for more replicas. When utilization is at 50%, forwarding claims conservatively and using that excess space to store claims keeps object loss rates low for all failure rates. Even when 16% of all nodes permanently fail, only 0.28% of objects are lost. The effect of utilization on reliability gives individual users an incentive to dedicate more local storage to the system.

It should be noted that our simulations are pessimistic. We have assumed that all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



This figure shows the percent of lost replicas and objects for utilizations of 50% and 80%, paired with both aggressive and conservative claims forward schemes.

Figure 5.10: Reliability of data under various storage utilizations.

failures are permanent, because we provide mechanisms for preserving data in the face of transient ones. Studies have shown that while transient failures are common, permanent ones happen far less frequently [17]. It is unlikely that even 1% of a large population of nodes would all permanently fail in such a short period of time that the affected nodes are unable to find replacement replica sites.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 6

CYCLIC EXCHANGE

Storage claims allow nodes to manufacture equal exchange and function as a store of value for future transactions. As long as permanent node failure is rare, the transitive arrangements created by forwarding storage claims allow trades between arbitrary nodes. However, the further downstream a claim is forwarded, the more likely it will be discarded and jeopardize the data for which it was exchanged.

This threatens claims' function as a store of value. A claim's value is inversely proportional to the risk that it will be discarded. It is only valuable to its owner if it can be overwritten later. Unfortunately, the risk of a claim not being available for overwriting accumulates as it is forwarded. Forwarding causes claims' value to depreciate. Low confidence in claims' ability to hold their value leads to a network in which nodes will only store claims from their preferred replica sites. Nodes will not be willing to circulate claims among arbitrary hosts, and only swaps between pairs of mutually interested nodes will be possible. This over-constrains storage allocation.

One way to limit the risk of forwarding claims is to guarantee that they will always form dependency cycles. Dependency cycles are one fault tolerant and do not jeopardize claim value to the degree that dependency chains do. Unfortunately, this is difficult to coordinate in a decentralized network of independent nodes. Instead, Pastiche can impose a new cyclic constraint on allocation decisions.

Our approach is to build a light-weight, distributed graph among Pastiche nodes based on storage demand and swap storage within cycles in the graph. We call this graph the *demand graph*. Edges in the demand graph are directed from a Pastiche node to its preferred

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

replica sites. In addition to avoiding the disincentive issues of forwarding claims, maintaining and searching the distributed demand graph require a tiny fraction of the bandwidth needed to forward claims.

Nodes must execute a distributed cycle search to create replicas. If a node finds a cycle, it nominates a cycle *coordinator.* The coordinator holds a vote among members of the cycle to determine the amount of storage to change hands and create dependencies between adjacent edges. BEE falls out as a special case of this new approach.

Cyclic allocation consists of two stages—cycle detection and cycle instantiation. To detect cycles, nodes send search messages along all incoming and outgoing edges. If a cycle exists, then each node in the cycle will receive the search message along at least one incoming edge and at least one outgoing edge. Nodes deterministically elect a member of the cycles as coordinator. The coordinator must then notify other members of the cycle and set the terms of the transaction, including the amount of storage exchanged and the dependencies.

A danger of this approach is that nodes will not be able to create replicas if cycles are difficult to find. This can happen if cycles do not form in the demand graph. Even when cycles do exist, they still must be found. Low node availability and high membership churn can impede searches. Nodes can address this problem by computing a risk-analysis function. This function helps nodes decide whether they should fall back on BEE or wait for a cycle to emerge.

6.1 Searching for Cyclic Demand

There is a large body of work exploring cycle detection in distributed graphs [18, 39, 64, 77, 86, 107]. Most of these algorithms are ill-suited to the dearth of global information in peer-to-peer networks, though our approach is similar to Chandy and Misra's [77]. Our strategy is to launch two parallel breadth-first searches—one in the same direction as the directed edges and one against the directed edges. If both searches reach a node, then it is in a cycle with the source.

Our graph consists of Pastiche nodes and edges defined by preferred replica sites. If A wants to store its data on B, then there is a directed edge from A to B, which we denote

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$A \Rightarrow B$. To establish the edges of the graph, each node contacts its preferred replica sites soon after entering the system. To initiate a search, nodes send search messages along all incoming and outgoing edges. Search messages originating from A are of the form $\{A, dir, time, TTL\}$, where $dir \in \{IN, OUT\}$. The *time* field enables repeat searches and the TTL field limits network flooding.

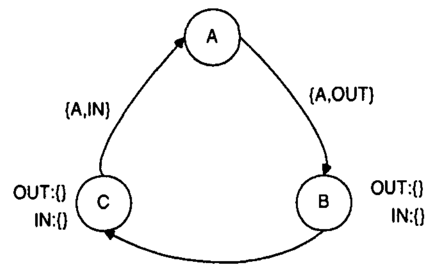
The Misra and Chandy algorithm assumes that the set of all vertices is known at each node. Because we cannot make this assumption, we use Bloom filters [14] to remember which search messages have been seen with high probability. A Bloom filter is a bit array of size j that summarizes set membership in constant space. Members of the set in question are hashed using k different hash functions. The hash results are then used to uniquely set bits in the bit array. The bits set by inserting an element into an empty filter are called the element's entries.

To later determine whether an element is in the set, the filter hashes the element and examines its entries. Depending on j and k, if the entries are set, then the element is in the set with very high probability. One important feature of Bloom filters is that they produce no false negatives. If an element has been added to the filter, any future check of that element will be positive. Bloom filters risk generating false positives, but for large enough j and k these should be rare.

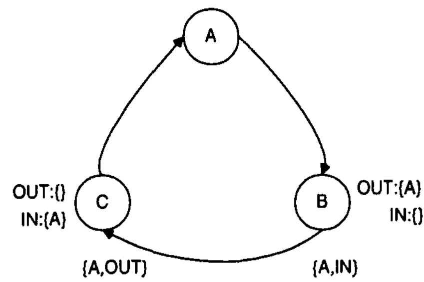
Each node maintains two Bloom filters similar to the *succeeding* and *preceding* functions of Misra and Chandy. One filter is for messages received along incoming edges called *outsearch* and one for messages received along outgoing edges called *insearch*. When a node receives a search message, it first checks to see if the source node has already been inserted into the Bloom filter associated with the message direction. If it has, then the node has already received the search message and does nothing more. If the node is not in the Bloom filter, then the recipient inserts it into the filter and forwards the message in the direction of the message field *dir*.

Next, the recipient checks whether the source node is in the opposite filter. If the source nodeID is in the opposite filter, then the recipient has received the search message from both directions and it is in a cycle with the source. Figure 6.1 shows an example search. When this happens, the recipient sends the source a message containing the edge along

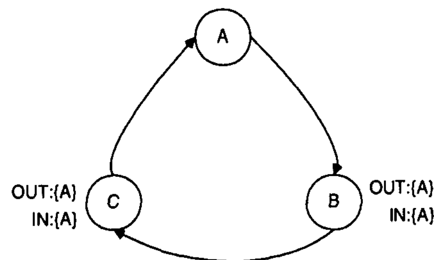
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) A begins a search



(b) B and C receive and forward a message from A



(c) B and C receive two messages from A

Figure 6.1: Example Search

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which the search message was received. A response message originating from A is either of the form $\{C, size, A, time\}$ when $C \Rightarrow A$ or $\{A, size, B, time\}$ when $A \Rightarrow B$. $size$ is the amount of storage required and $time$ is the value of the last received search message.

Source nodes can determine which cycles they are in from the response messages they receive. If node A is in a cycle with nodes $B_1, B_2, \ldots B_n$, it sends out search messages along all incoming and outgoing edges, and no nodes fail, each B_i will return a response message to A.

Say that node A is in a demand cycle with $B_1, B_2, \ldots, B_n$ such that $A \Rightarrow B_1, B_n \Rightarrow A$, and $B_i \Rightarrow B_{i+1}$ for $0 < i \leq n$. The first round of the search starts when A sends search message $m_{OUT} = \{A, OUT, time, TTL\}$ to B_1 and search message $m_{IN} = \{A, IN, time, TTL\}$ to B_n. After receiving m_{OUT}, B_1 adds A to $outsearch_{B+1}$. Similarly, after receiving m_{IN}, B_n adds A to $insearch_{B_n}$. In the next round of messages, B_1 forwards m_{OUT} to B_2 and B_n forwards m_{IN} to B_{n-1} who also update their Bloom filters, and so on. Assuming no failures, eventually all nodes in the cycle receive both m_{OUT} and m_{IN}. Thus, each node sends A a response message.

Because Bloom filters sometimes generate false positives, searching nodes may receive spurious response messages. The edges described by these messages may create a cycle in the searcher's internal graph that does not exist in the demand graph. While inconvenient, these inconsistencies will not affect allocation correctness. The searching node verifies each edge during the instantiation phase, which we will discuss in more detail in Section 6.2.

6.1.1 Search Efficiency and Flooding

Our search requires $O(e+n)$ messages per search, where there are e edges and n nodes in the graph. This is because search messages are only copied along an edge once in each direction and because each node sends at most one message back to the source. It uses $O(1)$ space per node for the Bloom filters.

It is likely that many nodes will search for cycles at once. This could cause $O(n(e+n))$ search messages to flood the network. We can reduce the likelihood of flooding by using the TTL field of the search message. Before a message is forwarded, TTL is decremented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When TTL reaches zero it no longer needs to be forwarded.

6.1.2 Repeatable Searches

Because of transient failure and membership churn, nodes will likely need to search more than once. Unless we are careful, repeat searches may be blocked by regularly connected nodes. For example, say that B's first search reaches A and B initiates a second search an hour later because it was unable to find a cycle. Unfortunately, when B's new search message reaches A, A will check its filter, find B, and not forward the message. The Bloom filters remember which messages a node has received, but block multiple searches from the same source.

To fix this, we use the *time* field to maintain logical clocks [68] between nodes and replace our Bloom filters' bit array with a counter array. Now, whenever a new element is added to the Bloom filter, rather than just setting bits in the array, nodes can use the search message's *time* field to update the filter. This allows nodes to distinguish between receiving the same search message twice and a new search by the same node.

Before, when a search message arrived, nodes checked to see if the source had been inserted in the Bloom filters. Now, nodes compute the minimum time over each entry of the source and compare it to the *time* field of the search message. If the *time* field is greater than the values from the filter, the node has not seen the search message before. It should update its filter and forward the message. Otherwise, the node has already seen the search message and doesn't need to do anything more.

6.2 Instantiating Cycles

If a node finds a cycle, it will receive response messages from other nodes. Using these messages, it can build a local, in-memory representation of the demand graph and apply a breadth-first search for additional cycles whenever a new edge is added. If this local search succeeds, the node must establish a cycle *coordinator*.

Pastiche nodes elect the cycle member with the lowest numeric id to be coordinator. Each node that discovers a cycle forwards the edges and the amount of demand along those edges to the coordinator. A node may be the coordinator of multiple cycles at once and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

must be careful that each cycle is disjoint.

Instantiating cycles is similar to the two-phase commit protocol [51]. In the first phase, the coordinator asks each member of the cycle to vote on its *terms*. The cycle's terms include the amount of data exchanged and all data dependencies. Currently, the amount traded is just the maximum demand among all edges. If a node accepts the terms of the cycle, it must temporarily withdraw its edge from the graph. If a node is asked to instantiate another cycle later using the same edge, it must must vote against it.

Nodes may vote against a cycle for other reasons. The amount of storage may be more than the node is willing to contribute. For example, node A may be asked to participate in a cyclic exchange of 100GB even though it only has 1MB of data. A may decide that 100GB of local storage is too much to pay to store 1MB of useful data. Also, a node may reject the terms of the cycle because of its length. The smaller a cycle is, the more stable it is likely to be. Additionally, messages may be lost or nodes may become disconnected during the first phase. If the coordinator does not receive a vote from a node within a time-out period, instantiation is aborted.

During the second phase, the coordinator informs each member of the cycle of the vote's outcome. The cycle can only be instantiated by consensus. If any node rejects the terms, the coordinator will send all nodes an "abort" message and no data changes hands. If a node receives an abort message, it can free its edge and use it to find other cycles. Otherwise, the coordinator sends all nodes a "commit" message, who in turn permanently withdraw their edges the demand graph. Once instantiation has been committed, data maintenance is regulated by BEE.

6.3 Augmenting Cyclic Allocation

In some environments, allocating storage along cycles may not be sufficient. This will be true when nodes are frequently disconnected or there are high rates of churn. High unavailability makes it difficult to maintain and search the demand graph.

Nodes who are unable to satisfy their storage demand through cycles are faced with a choice: they can either wait for a cycle to emerge or they can settle for the storage offered to them. For example, say that node A is having trouble creating replicas at its preferred

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sites. At the same time, node B has identified A as one of its preferred sites. A can either wait for a cycle to emerge or it can simply swap storage with a less preferred, but available, site like B.

There are risks associated with each choice. The longer A waits to create its replicas, the greater its risk of not recovering from a failure. This risk grows over time. However, the risk of waiting is inversely proportional to the number of replicas already in place. If A already has four replicas, adding a fifth has only a small effect on its ability to recover. If existing replicas are highly available or trusted because of an out-of-band relationship, the risk of waiting is even less.

On the other hand, adding the first or second replica or an available replica to an existing set of unavailable ones can greatly increase a node's ability to recover. However, settling for a less preferred replica site may compromise the independence of a replica set.

Because Pastiche does not support strong identities, an attacker can compromise the independence of other users' replicas. The attacker could create multiple identities and request storage under each identity from a recently joined node. To the victim, each request would appear to come from a different source. By accepting these requests, the victim would unwittingly store all of its replicas on a single host.

Nodes can capture these competing forces with a risk analysis function, R. This function uses a disk's age, how many replicas it has created, and how available those replicas are to compute the probability of not being able to recover from a failure in in the near future.

The risk of disk failure is modeled as a Poisson process with mean-time-to-failure (MTTF) of 290 days. This MTTF is based on observed machine lifetimes at Microsoft [17]. We model machine availability by periodically pinging hosts to see if they are alive. Using the ratio of successful to total pings, we assign an uptime ratio to each host, $0 \leq u_i \leq 1$, for $i \leq U$, the total number of replicas. This leads to the following risk function R, where t is the time:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$PFail(t) = \sum_{i=1}^{t} e^{-MTTF} \times \frac{MTTF^i}{i!}$$

$$R(t) = PFail(t) \times \prod_{i=0}^{U} (1 - u_i)$$

Each node has a risk threshold with which they are comfortable. Whenever R drops below its threshold, a node is willing to settle for whatever storage it can access, even though it is otherwise not preferred. Nodes can use R to monitor their risk and sustain a high level of recoverability if cyclic exchange fails them.

6.4 Evaluation

In evaluating cyclic storage allocation, we set out to answer the following questions:

- How many and what kinds of cycles in the demand can Pastiche nodes expect?
- How does node availability affect searching for cycles?
- Under what conditions is cyclic allocation insufficient?
- How does node churn affect recoverability?
- How much more bandwidth is required to cope with churn?

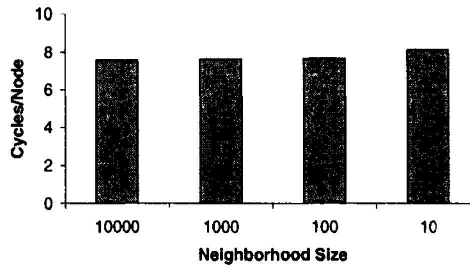
6.4.1 Characterizing Cycles

There must be cycles in the demand graph for cyclic allocation to be feasible. How many cycles emerge from the demand graph is a function of how nodes chose their preferred replica sites. Pastiche nodes prefer that most of their replicas have network proximity. These collocated nodes should create short, stable cycles among each another. We expect cycles among nodes with no network locality to be longer, if they exist at all.

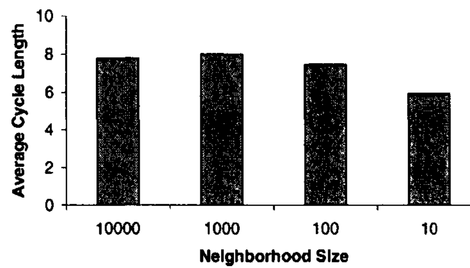
We simulated several Pastiche networks to characterize cycles among both collocated nodes and distant nodes. Each simulated network contained 10,000 nodes. The size of the set of collocated nodes is the *neighborhood* size. We varied the neighborhood size to understand the quality of the cycles Pastiche nodes can expect over a range of networks.

Each node in the simulation was assigned a unique identifier and chose ten preferred replica sites from the rest of the network. We assume that each node wanted 60% of

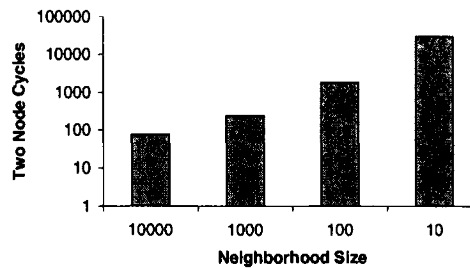
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Expected cycles per node



(b) Expected average cycle length



(c) Expected number of two node cycles

Figure 6.2: Neighborhood Size and Cycle Membership.

its replica to be nearby and 40% to be distant. Nodes selected distant replicas randomly from the entire network. They chose their nearby replicas by randomly sampling their neighborhood. A node's neighborhood consisted of the $2k$ closest nodes in the identifier space—the$k$ immediately greater than its identifier and the k immediately less than its identifier. Neighborhoods wrapped around the identifier space. When the neighborhood size was equal to the network size, nearby hosts were chosen the same way as distant ones.

To find cycles, we first created a graph based on each node's preferred replica sites. We then iterated through each node, using that node as the root for a breadth-first cycle search. As soon as the search found a cycle, we recorded the cycle's characteristics, removed its edges from the graph, and moved on to the next node. If we did not find a cycle, we simply

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

moved on to the next node. This process guarantees that each cycle is disjoint and that we will find the shortest cycles first. We continued this process until we iterated through each node without finding a cycle.

We were interested in answering two questions: how many disjoint cycles can nodes expect to be a member of and how long will they be? We looked at networks with neighborhood sizes of 10,000, 1,000, 100, and 10. Neighborhood size controls how much locality nodes can exploit. In the worst-case the neighborhood size is equal to the network size of 10,000. More likely, nodes will be able to identify a set of collocated hosts, although the size of this set will vary. Figure 6.2 shows the results for each network.

Figure 6.2 (a) shows that nodes can expect to be members of between seven and eight cycles regardless of the neighborhood size. This result was somewhat surprising. We expected cycles to be much more common as neighborhoods size shrank. Instead, they cycles were only slightly more common as neighborhood sizes shrank. However, 40% of preferred replica sites were chosen independently of the neighborhood size. This greatly dampened its overall effect on the number of cycles found.

While neighborhood size did not have much of an effect on the number of cycles, it did have an effect on the length of those cycles. Figure 6.2(b) shows that the average cycle length decreases by nearly 25% from a neighborhood size of 10,000 to 10. The primary reason for this decline is apparent from Figure 6.2(c). Here we see that the number of two node cycles grows linearly with falling neighborhood size. This makes sense since double coincidences of wants are more likely within smaller networks.

These results are encouraging because they demonstrate that nodes can expect to be members of a significant number of cycles for a range of networks. The availability of collocated nodes will shorten the length of the cycles nodes find, but it has little effect on the number of cycles nodes find.

6.4.2 Node Availability

If nodes are always available, they should find cycles easily. This may not be true under more realistic modes of availability. To explore the effect of availability on searching, we reran our simulations, but varied each node's availability. We used a study of machine up-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

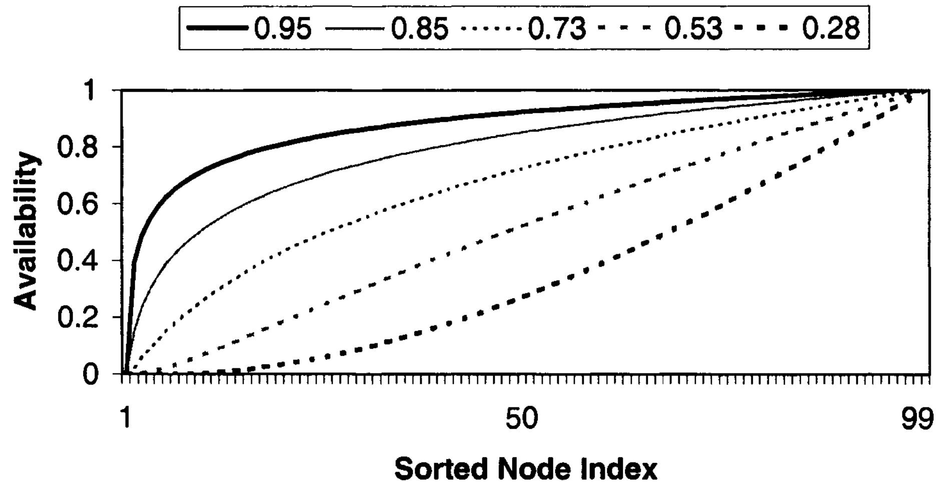


Figure 6.3: Availability Distributions.

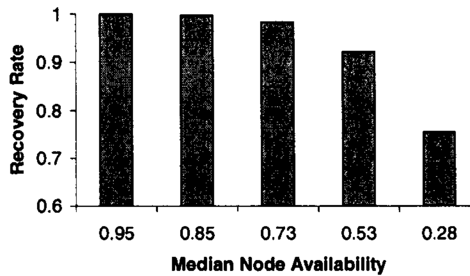
times at Microsoft [17] as a starting point. In this study, researchers periodically pinged machines at Microsoft over several months to determine if a machine was running or not. The study found that half of all machines were up 95% of the time and that up-times followed a logarithmic distribution.

Of course, a machine could be running without being available to the Pastiche network, but machine uptime is a reasonable approximation to Pastiche availability. The overhead of routing search messages and issuing queries is low and nodes have a strong incentive to remain available to queries as long as the machine is on.

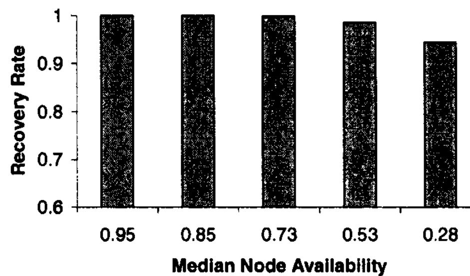
Nonetheless, to fully understand the effect of availability on searching for cycles, we used several availability distributions with different median values. For our experiments, the median node availability was .95, .85, .73, .53, or .28. Figure 6.3 shows each of these distributions. Before an experiment, we computed a discrete representation of the distribution curve with as many points as the network size. We then assigned each node a unique point on the curve as its availability.

Simulations were composed of discrete rounds and iterated through every node for each every round until the simulation finished. Ten simulated rounds were approximately equal to one day of physical time. At the beginning of its turn within a round, each node probabilistically went off-line or failed. The probability that a node failed during a round was computed using a Poisson process with a MTTF parameter and a node's disk age as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Recovery Rate without R



(b) Recovery Rate with R

Figure 6.4: Effect of Node Availability on Recovery Rate.

input. Nodes started with random disk ages at the start of the simulation so that the failure rate was constant. Failures made nodes unavailable for ten rounds, at the conclusion of which they reentered with no replicas and a reset disk age. We set the MTTF to 2900 rounds and ran the simulation until each node failed twice.

On a turn within a round, each node could transmit a new search or response message and check its message queues for messages received since its last turn. A search or response messages sent in one round was not received until the recipient's turn in the next round. If a recipient failed or was unavailable when a message was sent, the messages was never received.

If a node found a response message in its message queue, it added the edge to its internal graph and looked for a cycle. If it found one, the node tried to instantiate the cycle. If any member of the cycle was unavailable or had failed, instantiation was aborted. Otherwise, each node created its replica immediately.

When a node failed, it tried to recover using its replicas. If none of its replicas were available at the moment it failed, recovery failed. This condition for not recovering was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

overly conservative. A node's replicas may have been off-line the moment it failed, but they might help with recovery if they returned in a later round. However, for the purposes of our simulations, we assumed that recovery was only possible during the same round and turn as the failure. We were also only interested in failures that occurred in steady-state. Thus, our results do not include failures from the first 100 rounds of the simulations, when the network was still unsettled.

Figure 6.4(a) shows how availability affected recovery rate in steady state without the use of the risk-analysis function R. Nodes only used cyclic allocation and never settled for less preferred storage sites. Under the observed median uptime of .95, Pastiche demonstrated a perfect recovery rate. The recovery rate remained above 99% for a median availability of .85. However, as the median availability decreased so did recovery rate. When the half of all machines are available just 28% or less, Pastiche's recovery rate was 75%.

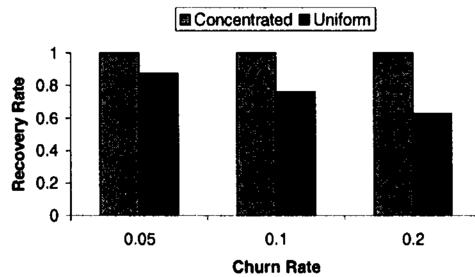
We next wanted to see how see how much the risk-analysis function R could help nodes understand when to use whatever storage was available before they failed. Figure 6.4(b) shows these results. Using R, Pastiche demonstrated a perfect recovery rate with the median availability as low as .73. With median availability .28, Pastiche's recovery rate was 94%. This was much better than the 75% without R.

6.4.3 Node Churn

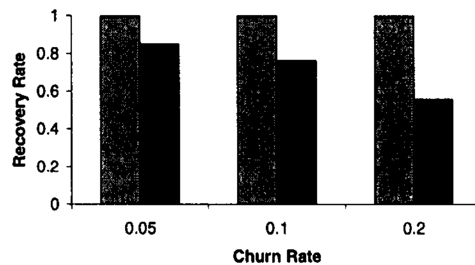
One of the most difficult aspects of building peer-to-peer services on the Internet is membership *churn*. Churn is the rate at which nodes enter the system for the first time or permanently leave. A study of the Overnet peer-to-peer file sharing system [11] found that new members comprised approximately 20% of the system membership. Existing hosts permanently exited at the same rate.

To explore the effect of churn on Pastiche we reran our simulations using churn rates of .20, .1 and .05 and median availabilities of .28, .73, and 95. For each experiment, the total membership size remained constant. However, the churn rate determined what percent of the network dropped its data at the beginning of each ten round period. This simulated the effect of nodes leaving and being replaced by new nodes.

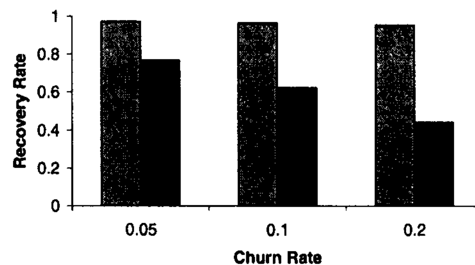
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) .95 Median Node Availability



(b) .73 Median Node Availability



(c) .28 Median Node Availability

Figure 6.5: Effect of Churn and Availability on Recovery Rate.

One aspect of churn that has not been well studied is *which* nodes come and go. One possibility is that membership churn is uniformly distributed—any node is as likely to leave at the end of the day as any other. A more likely scenario is that churn is more *concentrated* in certain parts of the network. In this scenario, give a 20% churn rate, 80% of nodes maintain their membership over the long-term while the remaining 20% of the network turn over on a daily basis. We simulated churn under both uniform and concentrated distributions.

If churn is concentrated, Pastiche punishes nodes with only short-term contributions. These nodes should have lower recovery rates by design. Thus, we presented the simulated recovery rate only for nodes who maintained their membership over the long-term.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 6.5 shows the results of these simulations.

When churn was concentrated, Pastiche performed nearly as well as with no churn at all. For a median availability of .95, Pastiche demonstrated a perfect recovery rate for all three churn rates. For a median availability of .73, recovery rate was approximately 99.7% for all three churn rates. For a median availability of .28, Pastiche demonstrated a recovery rate of 94%.

As expected, recover rates were low when churn was uniformly distributed. In the best simulated case, when the churn rate was only .05 and the median availability was .95, only 87% of nodes were able to recover from failure. When the median availability was .20 and the median availability was .95, only 63% of nodes recovered. In the worst simulated case, when the churn rate was .20 and median availability was .28, Pastiche demonstrated a recovery rate of only .44.

These low recovery rates reflect Pastiche's ability to bind a node's contribution to its level of service. For example, when the churn rate is only .05, nodes expected to turn over after only 140 rounds, or approximately two weeks. When the churn rate is .1, nodes only expect to last one week and when the churn rate is .2, replicas only expect to last four days. If a node is only willing to maintain storage for others for two weeks, its recovery rate should be low.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 7

RELATED WORK

Pastiche's goal is to provide convenient, low-cost backup. Pastiche is convenient because it only requires the user to remember a password. As we showed in Section 4.5.2, Pastiche is low-cost, because a user with 32GB of unique state will spend only $1.50 per month in opportunity costs over two years to create and maintain five replicas. To understand why users need Pastiche, we can compare it to existing backup options.

In Section 7.1, we evaluate other backup options using four criteria: cost, convenience, local safety, and catastrophic safety. Cost is how many dollars a user spends to backup data per month. Convenience measures the effort required to backup data. Local safety measures whether or not data can be recovered from a local disk failure and catastrophic safety measures whether or not data can be recovered from a local catastrophe.

In addition to addressing the problem of backup, Pastiche provides a platform for dealing with self-interested users. Pastiche addresses self-interest through bilateral, equal exchange, storage claims, and cyclic exchange. We arrived at these techniques after making several conservative assumptions, including the absence of a trusted authority and the uniformity of strategic behavior. Other research projects have made different assumptions and in Section 7.2 we examine how they affected various system designs.

7.1 Approaches to Backup

In general, there are three kinds of backup systems: centrally-managed repositories, user-managed repositories, and peer-to-peer repositories. Pastiche falls into the last category while most academic work outside of the peer-to-peer literature has focused on cen-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tralized backup of large installations [75, 90]; Chervenak provides a survey of a number of different backup systems [25].

Systems like Veritas' NetBackup [33] and IBM's Tivoli [34] are typical of the commercial systems available to organizations and users. These systems use a large storage infrastructure combining both rotating media such as a RAID for more recent snapshots and off-site tape archives of older data. These are large, expensive software and hardware packages that require expert administration. They are neither convenient nor low-cost for normal PC users. However, they do provide very high quality service. Users of these systems are safe from both simple disk failures and correlated, catastrophic failure.

AFS [57], Plan 9 [91], and WAFL [56] expose a snapshot primitive for a variety of purposes, including backup. Typically, snapshots are used to stage data to archival media other than disk. SnapMirror [88] leverages WAFL's snapshot mechanism to provide fine-grained, remote disk mirroring with low overhead.

Other centrally-managed repositories copy data over the Internet and are designed specifically for PC users and small organizations. These include services such as @backup [5], LiveVault [32], and ibackup [58]. These services are easy to use, do not require any extra hardware (although LiveVault offers a local TurboRestore appliance to speed recovery), and only require user involvement during setup to choose which file system subtrees to backup and how often.

The prices of these services vary widely, depending on the number of snapshots provided. At one end of the spectrum, LiveVault charges nearly $140 per month and provides realtime, continuous data backup of 5GB of state. On the other end, for just $10 per month, ibackup will store 5GB of data, but no snapshots. For all on-line services, data will be safe from local disk failures and larger catastrophic failures. However, these services force users to choose between the quality of their service and the price they are willing to pay. Pastiche is equally convenient, and provides high-quality service at a cost less than the lowest quality alternative.

Of course, users could also maintain their own backup state locally if they are willing to manage extra hardware. A simple scheme might use a tool such as the Rsync [108] utility to create nightly file system snapshots before copying the snapshots to a second

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

hard disk. After some non-trivial initial setup this scheme would protect users' data from local hardware failures. However, if the extra hardware is also stored locally, as is likely, users would be vulnerable to catastrophic failures such as a fires or floods.

A similar approach is to use inexpensive network-attached storage (NAS) devices such as the Linksys Network Storage Link disk drive [31]. This and other NAS devices attach to a PC either via USB or the local Ethernet network and require minimal initial setup. This is essentially the Rsync solution with a friendlier user interface. Thus, while they may be more convenient than setting up Rsync, they also leave data vulnerable to local catastrophes.

The third category of backup solution is a decentralized, peer-to-peer approach, like Pastiche. Several projects have suggested the use of peer-to-peer routing and object storage systems as a substrate for backup, including Chord [105], Freenet [26], and Pastry [97]. File systems built on them, such as PAST [98] and CFS [36], provide protection against machine failure. However, they do not protect against human error, nor do they provide the ability to retrieve prior versions of files once replaced. OceanStore [95], does provide these benefits, but the decision of which versions to retire rests with the utility, not its clients. This can lead to slower restores.

The pStore cooperative backup system [7], built on top of Chord, stores individual objects on a number nodes, rather than storing the entire set of objects. However, it does not exploit inter-host sharing, nor does it address the problem of self-interested hosts. Elnikety presents a cooperative backup scheme [45] that requests random blocks from partners, but assumes that partners either drop all or none of the archived state.

Pastiche's use of duplicate data is unique among peer-to-peer systems, although a number of other systems have used this technique in different contexts. The Single Instance Store [16] detects and coalesces duplicate files, while Venti [92] divides files into fixed-size blocks and hashes those to find duplicate data. Neither of these approaches can take advantage of small edits that move data within a file, as content-based indexing does [74, 79]. Other sophisticated techniques for detecting such changes exist [2, 108], but must be run on pairs of files that are assumed to have overlap.

Broder provides a mathematical foundation for detecting similarity and inclusion based

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on sketches [20], similar to Pastiche's abstracts. Sketches of a few hundred bytes are able to find similarities among single documents on the web [19]. Pastiche extends this result, to find similarities between entire disks.

Rather than exploit redundancy, one can instead turn to the use of erasure codes [89] to stripe data across several replicas. Such codes allow for low-overhead replication, and are tolerant of the failure of one or more replicas; they are employed by Myriad [22], OceanStore [95], and Elnikety [45]. Their main shortcoming, compared to our simpler scheme, is that they complicate overwriting chunks, since overwriting a single chunk can change multiple remote blocks. Furthermore, they require the participation of more than one node during restore.

7.2 Strategic Behavior

The application of economic analysis to computer systems is a growing area of research. It has been used in a wide range of systems areas, including multicast [46, 85], routing [12], sensor networks [73, 83], file sharing [112, 3], cluster computing [61, 48], storage [60, 35, 84], and wireless mesh networks [72].

The two primary sources of concern for these systems are how to encourage nodes to contribute resources and how to efficiently allocate those resources. Most research has focused on the latter, although the former has also received attention.

One way to approach the problem of inefficient allocation of finite resources is to place limits on the amount any individual can consume. Several peer-to-peer storage systems have considered the problem of enforcing storage quotas in some way, but all assume the presence of either a trusted third party or strong identities. Two of the first to address this problem were CFS [36] and PAST [98].

In CFS, each storage node limits any individual peer to a fixed fraction of its space. These quotas are independent of the peer's space contribution. CFS uses IP addresses to identify peers, and requires peers to respond to a nonce message to confirm a request for storage, preventing simple forgery. This does not defend against malicious parties who have the authority to assign multiple IP addresses within a block, and may fail in the presence of network renumbering [47].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PAST provides quota enforcement that relies on a smartcard at each peer [98]. The smartcard is issued by a trusted third party, and contains a certified copy of the node's public key along with a storage quota. The quota could be determined based on the amount of storage the peer contributes, as verified by the certification authority. The smartcard is involved in each storage and reclamation event, and tracks the peer's usage over time.

The grace period attack is also discussed by Lillibridge et al [69] in the context of a cooperative backup system. This system provides for challenges between peers, but requires symmetric storage relationships, restricting data placement. They propose two solutions to the grace period attack. In the first, peers must prove themselves reliable by storing random data for some time before benefiting from the system. In the second, a centralized third party is used to impose a storage "fine" immediately after a restore request.

In addition to these specific storage systems, there have been several efforts to produce a general framework that might be applied to peer-to-peer storage. Fileteller [60] proposes the use of micropayments to account for storage contributed and consumed. Such micropayments can provide the proper incentives for good behavior [50], but must be provisioned and cleared by a third party and require secure identities. Cooper [28] proposes *data trading*, the exchange of equal amounts of data, as a way to ensure fairness and discourage free-loading. This is similar to Pastiche's bilateral, equal exchange protocol, but suffers from the same problem of over-constrained allocation.

Ngan [84] has proposed a distributed accounting infrastructure. Each peer maintains a signed record of every data object it stores, and every object stored on its behalf elsewhere. Each node periodically chooses another random node to audit. For each file the node claims to hold for some other peer, the auditor retrieves the corresponding peer's record, and compares the two. If the auditor finds a node claiming an object it doesn't actually have, that node can be ejected from the system. This framework differs from Pastiche in several ways.

First, it requires certified identities to prevent false accusations and to ensure that aliases cannot claim to occupy storage on a node. Second, it tracks only actual usage, not capacity; it detects inflated claims of capacity only when the storage system is nearly

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

fully utilized. Pastiche detects inflated claims by any individual node as soon as it exceeds its contribution. Finally, a random audit may catch a cheater, but it is unlikely that the perpetrator claims the auditor itself as a victim. In other words, random audits benefit the group as a whole, but cost the auditor directly. Unless users are punished for not auditing or benefit directly, it is unclear why a rational node ever would from performing this task. In contrast, Pastiche nodes benefit directly from their police-work.

The problem of decentralized resource allocation on shared testbeds is also beginning to receive significant attention. SHARP [48] is a framework for the secure, distributed allocation and consumption of resources. Like Pastiche, SHARP assumes no globally certified identities. However, the mechanisms in SHARP require principals that exchange resource rights to first authenticate one another off-line to establish faith in one another's public keys. Furthermore, resource exchanges in SHARP may be asymmetric, and potentially rely on non-local means to detect and punish nodes that advertise and later withhold services. This is appropriate for platforms like the PlanetLab [8] Internet testbed, where resources are directly controlled by larger administrative entities like corporations and universities. It is not as appropriate for Pastiche, which is designed for independent, unaffiliated users.

Several systems, such Cereus [61] and Mirage [83] provide users with monetary budgets and allocate resources via auction. Cereus builds on SHARP, but uses auctions and *self-recharging currency* rather than bartering. Mirage applies a similar approach in the context of a sensor network testbed. The main difference between the two systems is in how their respective currencies are managed. Cereus expires and refreshes user budgets after a fixed amount of time. This provides a steady flow of currency over time and prevents cycles of inflation and deflation. Mirage, on the other hand, allows users to save and accrue currency, although savings are taxed to discourage hoarding. Both systems rely on a certified fiat currency, which introduces administrative costs eschewed by Pastiche.

In addition to peer-to-peer storage, several systems [12, 85] have looked at eliminating free-loading in peer-to-peer routing. Routing is different than storage in that long-term, bilateral relationships are rare. This can make Pastiche's punishment model difficult to apply.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[12] models the routing problem as a prisoner's dilemma game in which a trusted, global authority punishes free-loaders by assigning them a poor reputation. Users with poor reputations are more likely to have their packets dropped. While this approach is somewhat limited because it assumes the presence of a trusted authority, node strategies converge toward compliance under this assumption.

[85] takes an approach to routing that is similar to Pastiche's to storage. It attempts to create long-term relationships between nodes so that if they observe cheating they can retaliate later. Rather than using a global reputation list, users maintain local lists and record which paths result in packet loss. It is in nodes' interest to forward packets because forwarding paths are periodically reordered. Reordering the routing paths places nodes that may have been cheating behind nodes who have experienced packet loss. This gives victims an opportunity to retaliate by dropping packets from nodes they have observed misbehaving.

In some environments it may be reasonable to assume that a majority of users are altruistic, even though we have not in Pastiche. Catch[72] does make this assumption and uses it to address free-loaders in wireless mesh networks. Catch is populated by a majority of altruistic *watchdogs* who are willing to do extra police work for the greater good. Watchdogs target free-loaders by forwarding them anonymous messages and waiting for them to be rebroadcast. Free-loaders are identified based on the relative rebroadcast rate of these anonymous packets and actual data packets. If the rebroadcast rate of anonymous messages is higher than of data packets, the target is flagged as a free-loader.

Once a watchdog identifies a free-loader, it must inform the nodes around the cheater to stop forwarding packets on its behalf. This is tricky when the free-loader lies between watchdogs in the network topology. Thus, to notify others the watchdog chooses a random token and broadcasts its cryptographic hash. Other watchdogs listen for rebroadcasts of the hash followed by a rebroadcast of the token itself. If the watchdog is trying to notify others of a free-loader, it will withhold the token. If the other watchdogs do not receive the token, they assume the target is a free-loader and cease forwarding its packets.

Finally, our use of cyclic exchange in Pastiche is very similar to work in peer-to-peer file sharing [3]. The goal of this system is to align the incentive to contribute content

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with download performance. Nodes looking to download a file can improve their wait queue position by finding a cycle of which both they and a file owner are a member. The primary difference between this work and Pastiche's cyclic exchange is the stability of the transactions. While Pastiche reroutes data around failed nodes, file sharers can only receive data as long as every other member of the cycle does. Once a download finishes or is cut short, the entire cycle collapses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 8

CONCLUSIONS

This dissertation set out to demonstrate the feasibility of a convenient, low-cost collaborative backup service using unreliable, untrusted, and self-interested hosts. Pastiche provides such a service. Pastiche provides high-quality backup by storing users' backup state in a self-organizing collective of excess disk storage. It can recover from both local hardware failures such as disk crashes and large-scale correlated failures such as fires, floods, and power surges.

Pastiche is convenient because it requires no extra hardware and no administration beyond remembering a password. Snapshots and restores are handled by the system automatically and require minimal user interaction. During recovery, users do not need to locate any application or operating system files from online sources or CDs. Users only need to authenticate themselves to the system to restore their disk.

Pastiche also costs less than conventional alternatives. Users can backup up to 24GB of state over two years for under one dollar per month. This is significantly less than the tens or even hundreds of dollars per month charged by centralized or online services. Pastiche achieves this low-cost by taking advantage of excess PC disk and network capacity to render sunk the normal power, cooling, and administrative costs of using a data center.

Pastiche faced three major challenges: providing highly available storage on hosts that can come and go without warning, ensuring data privacy and integrity while storing on untrusted hosts, and encouraging storage contribution from self-interested users.

Because members of the collective are independent, Pastiche data must be replicated at multiple hosts. Unfortunately, replication can strain precious system resources. Pastiche

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

eases this strain by identifying common inter-host data. If users only ship the data that is unique to them, the collective can accommodate higher replication factors and provide greater availability. First, Pastiche looks for local data redundancy by using content-based indexing to break files up into content-specific pieces called chunks. These chunks are, in turn, named by the cryptographic hash of their content. Hosts can then use the list of these chunks to locate disks with similar data. A survey of disks in the EECS department at the University of Michigan found that machines with common installations could easily identify multiple disks with between 30 and 70 percent of their data data in common.

Data privacy and integrity are easy to provide through encryption. However, Pastiche users cannot simply choose their own encryption keys. If this happened, identical chunks would be encrypted to different values, which would destroy data sharing. Instead, users apply convergent encryption to chunks. Pastiche encrypts chunks using keys derived from the contents of the chunk itself. This allows holders of the same block to compute the same encryption key and preserves inter-host data sharing.

Encouraging storage contribution in a network of self-interested hosts was the greatest challenge of Pastiche. If Pastiche experiences too many free-loaders, the collective may collapse from cascading resource withdrawals. To prevent this, Pastiche transformed the opportunity costs of sharing from a cost of production into a cost of consumption.

It did this through a barter protocol called bilateral, equal exchange (BEE). Hosts are only allowed to store data on those for whom they are storing an equal amount in return. Once data has been swapped, hosts query one another to ensure that their's is intact. If a host fails a query, its data is discarded in retaliation. This simple protocol ensures that all hosts contribute as much storage as they consume. More important, it provides a strong incentive to contribute storage since discarding another host's data will result in the loss of one's own. Our analysis of a formal model of BEE predicts that under realistic resource prices, Pastiche users will contribute storage.

The disadvantage of BEE is that it over-constrains resource allocation. Users are only allowed to store data on hosts that want to store data with them. To allow more flexible storage allocation, Pastiche augments BEE with uncompressible, junk placeholders called storage claims. Storage claims allow hosts to manufacture equal exchange when it does

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not arise naturally. Because claims do not participate in recovery, they can circulate and be overwritten. This allows them to act as a store of value for future transactions. Forwarding claims allows arbitrary hosts to exchange storage, but can create volatile dependency chains. Simulations show that dependency chains do not threaten recoverability unless there is large-scale simultaneous failure.

Nonetheless, Pastiche can provide more stable, cyclic arrangements through cyclic exchange. Under cyclic exchange, hosts build a distributed demand graph based on preferred replica sites. Users then search the graph for cycles and allocate storage along the edges of found cycles, creating dependencies between adjacent edges. Cyclic arrangements are more stable than chains and searching the demand graph does not incur the cost of transferring large collections of claims between hosts. Simulations of cyclic exchange show that it is robust in the face of low host availability and observed rates of churn. Together, BEE, storage claims, and cyclic exchange provide incentives for users to contribute without introducing any significant new overhead to the basic Pastiche architecture.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

[1] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.

[2] M. Ajtai, R. Burns, R Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, to appear.

[3] K. G. Anagnostakis and M. B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *Proceedings of the 24th International Conference On Distributed Computing Systems*, Tokyo, Japan, March 2004.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI-home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[5] @Backup. @Backup Service.
http://www.backup.com.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[7] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Unpublished report, MIT Laboratory for Computer Science, December 2001.

[8] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of the First Symposium on Network Systems Design and Implementstion (NSDI)*, San Francisco, CA, March 2004.

[9] K. Belson. Dial-up internet going the way of rotary phones. The New York Times, June 21, 2005.

[10] Y. Benkler. Sharing nicely: On shareable goods and the emergence of sharing as a modality of economic production. *Yale Law Journal*, 114(273), 2004.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[11] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[12] A. Blanc, Y. K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proceedings of the IEEE Infocom Conference*, Miami, FL, March 2005.

[13] M. Blaze, J. Ioannidis, and A. Keromytis. Offline micropayments without trusted hardware. In *Proceedings of the Fifth Annual Conference on Financial Cryptography*, pages 21–40, Cayman Islands, BWI, February 2001.

[14] B. Bloom. Space time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[15] B. Bolosky. `personal communication`, July 2005.

[16] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000.

[17] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.

[18] A. Boukerche and C. Tropper. A distributed graph algorithm for the detection of local cycles and knots. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):748–757, August 1998.

[19] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World-Wide Web Conference*, pages 391–401, Santa Clara, CA, April 1997.

[20] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES*, pages 21–29, Salerno, Italy, June 1997. Published in 1998.

[21] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Submitted for publication.

[22] F. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 103–116, Monterey, CA, January 2002.

[23] J. S. Chase, D. C. Anderson, P.N. Thakar, , A. M. Vahdat, and R.P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[24] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto '82*, pages 199–203, August 1982.

[25] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.

[26] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting fee expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[27] Connected Corporation. The 60% you're missing: Preventing data loss through PC management. White paper, Farmingham, MA, 2002.

[28] B. F. Cooper and H. Garcia-Molina. Peer-to-peer resource trading in a reliable distributed system. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 319–327, Cambridge, MA, March 2002.

[29] Boston Computing Network Corporation. Data loss statistics. `http://bostoncomputing.net/consultation/databackup/statistics/`, 2003.

[30] Iomega Corporation. Nearly one in four coputer users have lost content to blackouts, viruses, and hackers according to new national survey. Iomega press release, February 2001.

[31] Linksys Corporation. Nslu2. `http://www.linksys.com`.

[32] LiveVault Corporation. Livevault. `http://www.livevault.com`.

[33] Veritas Corporation. Netbackup 6.0. `http://www.veritas.com`.

[34] IBM Corportation. Tivoli. `http://www.ibm.com`.

[35] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[36] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.

[37] J. Daemen and V. Rijmen. AES proposal: Rijndael. Advanced Encryption Standard Submission, 2nd version, March 1999.

[38] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–54, November 1976.

[39] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[40] divetomark. How to install windows xp in 5 hours or less. `http://diveintomark.org/archives/2003/08/04/xp`, August 2003.

[41] J. R. Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 251–260, Cambridge, MA, March 2002.

[42] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 59–70, Atlanta, GA, May 1999.

[43] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260, Kiawah Island Resort, SC, December 1999.

[44] P. Druschel and A. Rowstron. PAST: a large-scale, persistent peer-to-peer storage utility. In *Proceedings of 8th Workshop on Hot Topic in Operating Systems*, pages 75–80, Elmau, Germany, May 2001.

[45] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. Cooperative backup system. In *The USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002. Work-in-progress report.

[46] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 2001(63):21–41, 2001.

[47] P. Ferguson and H. Berkowitz. Network renumbering overview: Why would i want it and what is it anyway? Internet RFC 2071, January 1997.

[48] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[49] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[50] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proceedings of the Third ACM Conference on Electronic Commerce*, pages 264–267, Tampa, FL, October 2001.

[51] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, pages 394–481. Berlin: Springer-Verlag, 1978.

[52] P. Gupta and P.R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46, March 2000.

[53] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Cryptology*, 3(2):99–111, 1991.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[54] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.

[55] V. Henson. An analysis of compare-by-hash. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[56] D. Hitz, J. Lau, and M. A. Malcom. File system design for an NFS file server appliance. In *Proceedings USENIX Winter Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[57] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[58] iBackup Corportation. ibackup. `http://www.ibackup.com`.

[59] Harris Interactive. News release. `http://imation.com/storagesurvey`, September 2002.

[60] J. Ioannidis, S. Ioannidis, A. D. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proceedings of the Sixth Annual Conference on Financial Cryptography*, pages 282–299, Bermuda, March 2002.

[61] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *Proceedings of the Third Workshop on the Economics of Peer-to-peer Systems*, Philadelphia, PA, August 2005.

[62] Corporation ITIC. P2P and music statistics for November 2004. `www.itic.ca`, November 2004.

[63] J.Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[64] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, March 1975.

[65] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA, February 1999.

[66] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Maziéres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.

[67] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[68] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[69] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proceedings of the USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, June 2003.

[70] S.H. Low, F. Paganini, J. Wang, S Adlakha, and J.C. Doyle. Dynamics of TCP/RED and a scalable control. In *Proceedings of IEEE/INFOCOM'02*, New York, NY, June 2002.

[71] C.R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 117–129, Monterey, CA, January 2002.

[72] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[73] G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[74] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Conference*, pages 1–10, San Francisco, CA, January 1994.

[75] E. Melski. Burt: the backup and recovery tool. In *Proceedings of LISA'99*, pages 207–217, Seattle, WA, November 1999.

[76] Microsoft Corporation. SimPastry. http://www.research.microsoft.com/~antr/Pastry/ download.htm.

[77] J. Misra and K. M. Chandy. A distributed knot algorithm. *ACM Transactions on Programming Languages and Systems*, 4(4):678–686, October 1982.

[78] R.J.T. Morris and B.J. Truskowski. The evolution of storage systems. *IBM Systems Journal*, 42(2):205–217, 2003.

[79] A. Muthitacharoen, B. Chen, and D. Maziéres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Candada, October 2001.

[80] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 33–43, Seattle, WA, May 1989.

[81] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[82] National Institute of Standards and Technology. Secure hash standard. FIPS Publication #180-1, April 1997.

[83] C. Ng, P. Buonadanna, B. N. Chun, A. C. Snoeren, and A. Vahdat. Addressing behavior in a deployed microeconomic resource allocator. In *Proceedings of the Third Workshop on the Economics of Peer-to-peer Systems*, Philadelphia, PA, August 2005.

[84] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[85] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems*, Cambridge, MA, June 2004.

[86] R. Obermarck. Distributed deadlock detection algorithm. *Transactions on Database Systems*, 7(2):187–208, 1982.

[87] J. M. Ostroy and R.M. Starr. The transactions role of money. In *Handbook of Monetary Economics Volume 1*. Elsevier Publishing, November 1990.

[88] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 117–129, Monterey, CA, January 2002.

[89] W. W. Peterson and E. J. Weldon. *Error-correcting Codes*. The MIT Press, 1972.

[90] W. C. Preston. Using Gigabit Ethernet to backup six Terabytes. In *Proceedings of LISA'98*, pages 87–95, Boston, MA, December 1998.

[91] S. Quinlan. A cache WORM file system. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.

[92] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 89–102, Monterey, CA, January 2002.

[93] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[94] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference*, San Diego, CA, August 2001.

[95] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[96] R. Rivest. The MD5 message-digest algorithm. Internet RFC 1321, April 1992.

[97] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001.

[98] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Banff, Canada, October 2001.

[99] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Kiawah Island Resort, SC, December 1999.

[100] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[101] S. Saroiu, G. P. Krishna, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, pages 156–170, San Jose, CA, January 2002.

[102] M. Satyanarayanan. *RPC2 User Guide and Reference Manual.* School of Computer Science, Carnegie Mellon University, October 1991.

[103] B. Schneier. *Applied Cryptography.* John Wiley and Sons, New York, second edition, 1996.

[104] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[105] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, San Diego, CA, August 2001.

[106] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the ACM SIGCOMM 2000 Conference*, pages 309–319, Stockholm, Sweden, August 2000.

[107] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[108] A. Tridgell. *Efficient algorithms for sorting and synchronization.* PhD thesis, The Austrailian National University, 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[109] G. Tsudik. Message authentication with one-way hash functions. In *IEEE Conference on Computer Communications*, pages 2055–2059, Florence, Italy, May 1992.

[110] J. D. Tygar, A. Gupta, O. Shmueli, and J. Widom. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*, pages 1–12, New York, NY, August 1998.

[111] University of Wisconsin. Backup Cost Calculator. http://www.doit.wisc.edu/backup/bbcalc.asp.

[112] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for p2p resource sharing. In *Proceedings of the First Workshop on the Economics of Peer-to-peer Systems*, Berkeley, CA, June 2003.

[113] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[114] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Proceedings of the 24th Annual Eurocrypt Conference*, pages 19–35, Aarhus, Denmark, May 2005.

[115] A. Westerlund and J. Danielsson. Arla—a free afs client. In *Proceedings of 1998 USENIX, Freenix track*, New Orleans, LA, June 1998.

[116] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division (EECS), University of California, Berkeley, April 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.