

RICE UNIVERSITY

**Autonomous storage management for low-end
computing environments**

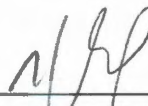
by

Ansley Post

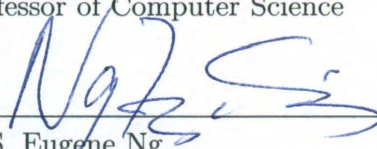
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

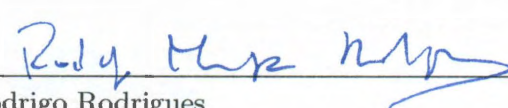
APPROVED, THESIS COMMITTEE:



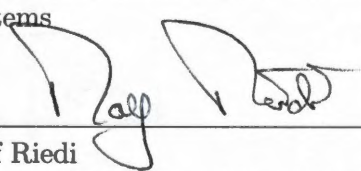
Peter Druschel, Chair
Professor of Computer Science



T. S. Eugene Ng
Assistant Professor of Computer Science



Rodrigo Rodrigues
Assistant Professor of Computer Science,
Max Planck Institute for Software
Systems



Rolf Riedi
Associate Professor of Statistics

Houston, Texas

October, 2010

Autonomous storage management for low-end computing environments

Ansley Post

Abstract

To make storage management transparent to users, enterprises rely on expensive storage infrastructure, such as high end storage appliances, tape robots, and offsite storage facilities, maintained by full-time professional system administrators. From the user's perspective access to data is seamless regardless of location, backup requires no periodic, manual action by the user, and help is available to recover from storage problems. The equipment and administrators protect users from the loss of data due to failures, such as device crashes, user errors, or virii, as well as being inconvenienced by the unavailability of critical files.

Home users and small businesses must manage increasing amounts of important data distributed among an increasing number of storage devices. At the same time, expert system administration and specialized backup hardware are rarely available in these environments, due to their high cost. Users must make do with error-prone, manual, and time-consuming ad hoc solutions, such as periodically copying data to an external hard drive. Non-technical users are likely to make mistakes, which could

result in the loss of a critical piece of data, such as a tax return, customer database, or an irreplaceable digital photograph.

In this thesis, we show how to provide transparent storage management for home and small business users. We introduce two new systems: The first, PodBase, transparently ensures availability and durability for mobile, personal devices that are mostly disconnected. The second, SLStore, provides enterprise-level data safety (e.g. protection from user error, software faults, or virus infection) without requiring expert administration or expensive hardware. Experimental results show that both systems are feasible, perform well, require minimal user attention, and do not depend on expert administration during disaster-free operation.

PodBase relieves home users of many of the burdens of managing data on their personal devices. In the home environment, users typically have a large number of personal devices, many of them mobile devices, each of which contain storage, and which connect to each other intermittently. Each of these devices contain data that must be made durable, and available on other storage devices. Ensuring durability and availability is difficult and tiresome for non-expert users, as they must keep track of what data is stored on which devices. PodBase transparently ensures the durability of data despite the loss or failure of a subset of devices; at the same time, PodBase aims to make data available on all the devices appropriate for a given data type. PodBase takes advantage of storage resources and network bandwidth between devices that typically goes unused. The system uses an adaptive replication algorithm, which

makes replication transparent to the user, even when complex replication strategies are necessary. Results from a prototype deployment in a small community of users show that PodBase can ensure the durability and availability of data stored on personal devices under a wide range of conditions with minimal user attention.

Our second system, SLStore, brings enterprise-level data protection to home office and small business computing. It ensures that data can be recovered despite incidents like accidental data deletion, data corruption resulting from software errors or security breaches, or even catastrophic storage failure. However, unlike enterprise solutions, SLStore does not require professional system administrators, expensive backup hardware, or routine, manual actions on the part of the user. The system relies on *storage leases*, which ensure that data cannot be overwritten for a pre-determined period, and an adaptive storage management layer which automatically adapts the level of backup to the storage available. We show that this system is both practical, reliable and easy to manage, even in the presence of hardware and software faults.

Acknowledgments

I would like to thank all of those who contributed towards making this thesis possible. First and foremost, I would like to thank my advisor, Peter Druschel, without whom this would not have been possible. I would also like to thank the members of my thesis committee for their valuable feedback. Finally, I would like to acknowledge all of the people I have collaborated with in my graduate career. Specifically, in the context of this thesis, I owe a debt to Petr Kuznetsov and Juan Navarro for their help formalizing the problem of storage management.

On a personal level, I would like to thank my girlfriend, Alex Kimsey, for her patience throughout the entire Ph.D. process, and my family for their support. I would also like to thank all of the friends I made along the way, who made the whole process more enjoyable. In particular, I would like to thank Marcus Perlman for traveling to visit me every year, Alan Mislove for going through the whole process with me, and Arjun V. Reddy for giving me a place to stay when visiting Houston and submitting this thesis.

Contents

Abstract	ii
Acknowledgments	v
List of Illustrations	xii
List of Tables	xvii
1 Introduction	1
1.1 PodBase	2
1.2 SLStore	5
1.3 Contributions	7
1.4 Thesis Overview	8
2 Related work	9
2.1 Storage Devices	9
2.1.1 Technology Overview	10
2.1.2 Storage Device Deployments	13
2.2 File Systems	14
2.2.1 Local File Systems	14
2.2.2 Distributed File Systems	16

2.2.3	Personal Device File Systems	19
2.3	Backup Systems	22
2.3.1	Individual Backup	22
2.3.2	Enterprise Backup	24
2.3.3	Archival Systems	25
2.4	Synchronization	26
2.5	Operations research techniques	28
2.6	Networking techniques	28
2.7	Extending storage device functionality	29
2.8	Contributions beyond related work	30
I	PodBase	32
3	Feasibility	34
3.1	Trace Collection	34
3.1.1	Limitations	35
3.2	Results: Feasibility	36
4	System Description	41
4.1	System Goals	41
4.1.1	Target environment	41
4.1.2	Desired system behavior	42

4.1.3	Definitions	44
4.2	Design of PodBase	47
4.2.1	Overview	47
4.2.2	User interaction	49
4.2.3	Device interaction	50
4.2.4	Plug-ins	54
4.2.5	Security	56
4.3	Replication	56
4.3.1	Greedy replication	57
4.3.2	Adaptive Replication	59
4.3.3	Adaptive Replication Formulation	63
5	Experimental evaluation	74
5.1	Implementation	74
5.2	Controlled experiments	75
5.2.1	Computation and storage overhead	75
5.2.2	Pairwise transfer throughput	77
5.2.3	Data restoration	78
5.2.4	Replication	78
5.2.5	Adaptivity	79
5.2.6	Partial metadata reconciliation	79
5.3	User study methodology	80

5.4	User study 1	81
5.4.1	Replication results	83
5.4.2	Availability	85
5.4.3	Replication latency and throughput	85
5.4.4	File conflicts	88
5.5	User Study 2	89
5.5.1	Replication results	91
5.5.2	Availability results	94
5.5.3	Replication latency and throughput	95
5.5.4	File Workload	97
5.6	Summary	99
II	SLStore	100
6	System Description	102
6.1	System Overview	102
6.2	Storage leases	104
6.2.1	Motivation	104
6.2.2	The storage lease abstraction	107
6.3	Implementing storage leases	114
6.3.1	Firmware implementation	114
6.3.2	Driver implementation	119

6.3.3	Storage enclosure implementation	120
6.3.4	Cloud storage implementation	120
6.3.5	Applications of storage leases	121
6.4	Planner	123
6.4.1	Backup Policy	123
6.4.2	Planning Process	124
6.5	Backup Agent	126
6.5.1	Snapshots	126
6.5.2	Determining which files have changed	127
6.5.3	De-duplication	128
6.5.4	Restoring a snapshot	128
6.5.5	Snapshot verification and repair	129
7	Evaluation	132
7.1	Storage lease evaluation	132
7.1.1	SLStore evaluation	142
7.1.2	Implementation details	142
7.1.3	Methodology and traces	143
7.1.4	Backup evaluation	145
7.1.5	Simulated long-term results	150
7.2	Summary	151

8 Conclusion	152
---------------------	------------

Bibliography	153
---------------------	------------

Illustrations

3.1	Number of devices connected by user.	36
3.2	Amount of used and free space by user.	37
3.3	Amount of time before generated data can be replicated in trace. . .	39
4.1	Data flows through the system whenever connectivity occurs. (a) When the top two devices are connected, they replicate files. (b) Later, a new connection occurs, and data flows from the first node to the third node through the second node.	48
4.2	A small device that periodically connects to different devices has the potential to act as “communications device”, transporting replicas between devices that cannot directly connect or are connected via a slow link.	59
4.3	Summary of system state and environment, which is used to generate a linear programming problem. This is the input that is needed in order to generate the initial planning step, which then generates a set of linear programming problems.	70

4.4 Simplified snippet of LP program generated by example problem.

The objective function in this stage of the formulation is to minimize the total cost, given that the durability and availability goals are satisfied (the lines directly following Subject to). The next constraint enforces that the data the files stored on a device in a step never exceed the capacity of that device (the data being stored in each storage set is capture by the variables starting with f. Each letter following f is a device which the data is present on, and the following number indicate what step, and what type (0 = durability, 1 = availability)). Next, the variables which encode actions in the system are shown, and they are related to the cost. Finally, the costs are aggregated into the objective function. A variable above is always indexed by first the devices that it is / could be stored on, then the type of file (availability=1, durability = 0), and finally the step in the output plan. Action variables (above copy) also include a target device, in the case of the copy variables the step is omitted to shorten the variable names. 72

5.1 Replication over time for an example user 78

5.2 The number and type of devices present in the deployment, by household. 81

5.3	The amount of storage capacity and free space present on devices before PodBase begins replication. Additional space corresponds to the USB disks households 1 and 4 were given.	82
5.4	The initial (left bar) and final (right bar) replication status of households 1,2,3,4,5	83
5.5	The initial (left bar) and final (right bar) replication status of households 6,7,8,9,10	83
5.6	Peak daily throughput for each household	86
5.7	Replication latency for households 2,4,5,7,9	86
5.8	Replication latency for households 1,3,6,8,10	87
5.9	The number and type of devices present in the deployment, by household.	89
5.10	The amount of storage capacity and free space present on devices before PodBase begins replication. Additional space corresponds to the USB disks households 1, 4, and 5 were given.	90
5.11	The initial (left bar) and final (center bar) replication status of each household. Final results for the greedy algorithm (right bar) are shown for comparison.	91
5.12	Peak daily throughput for each household	95
5.13	Replication latency for households 1,2,4,7,9	96
5.14	Replication latency for households 3,5,6,8,10	97

- 6.1 A diagram showing 3 machines, each running one or more system component, connected via a network. A disk with a clock indicates a storage lease device, a disk without a clock indicates a conventional disk. Also shown is a connection to a remote cloud storage provider that supports storage leases. 103
- 7.1 Normalized I/O throughput (ops/sec) for synthetic workloads, with different lease placements shown on the x axis. The cache configuration is Unified (U), except for the inline (I) placement, where lease values are cached inline (I). The results are normalized to the throughput without leases. 137
- 7.2 Normalized I/O throughput (ops/sec) for trace workloads, with different lease placements shown on the x axis. The cache configuration is Unified (U), except for the inline (I) placement, where lease values are cached inline (I). The results are normalized to the throughput without leases. 137
- 7.3 Normalized I/O throughput (ops/sec) for synthetic workloads, with different cache configurations shown on the x axis. The total cache size is 16MB in all cases, and the lease value placement is S16. The results are normalized to the throughput without leases. 139

7.4	Normalized I/O throughput (ops/sec) for trace workloads, with different cache configurations shown on the x axis. The total cache size is 16MB in all cases, and the lease value placement is S16. The results are normalized to the throughput without leases.	139
7.5	Average per-operation I/O response time for the trace workloads. The traces were replayed with the recorded operation inter-arrival times (think times), and the per-operation response times measured. The placement is S16 and the cache configuration is unified. The results are normalized to the average per-operation response time without leases.	140
7.6	Normalized throughput (I/O ops/sec) for combined trace and mock backup workload with lease, with different batch sizes shown on the x axis. Crypto processor with 325 MB/s hash throughput, RSA latency 8.4ms. Unified 16MB cache, placement S16.	142
7.7	Percentage overhead in a variety of home offices	148
7.8	Absolute Overhead in a variety of home offices.	148
7.9	Effectiveness of de-duplication	149
7.10	Minimum settings required for initial capacity and rate, assuming no deduplication.	151

Tables

5.1	Transfer throughput for different connection types	77
6.1	Storage lease device interface. Additionally, the device supports normal read and write operations. However, write operations to a block with an active lease fail.	130
6.2	Replication policies in order from strongest to weakest. The planner chooses the strongest feasible policy from this table, and then chooses the best feasible snapshot schedule from Table 6.3.	131
6.3	Snapshot schedules, ordered from weakest to strongest.	131
7.1	Workloads used in evaluation	134
7.2	Placements of lease values	135
7.3	Lease cache policies	135

Chapter 1

Introduction

To make storage management transparent to users, enterprises rely on expensive storage infrastructure, such as high end storage appliances, tape robots, and offsite storage facilities, maintained by full-time professional system administrators. From the user's perspective access to data is seamless regardless of location, backup requires no periodic, manual action by the user, and help is available to recover from storage problems. The equipment and administrators protect users from the loss of data due to failures, such as device crashes, user errors, or virii, as well as being inconvenienced by the unavailability of critical files.

Home users and small businesses must manage increasing amounts of important data distributed among an increasing number of storage devices. At the same time, in these environments, due to cost constraints, expert system administration and specialized backup hardware are rarely available to manage this data. Users must make do with error-prone, manual, and time-consuming ad hoc solutions, such as periodically copying data to an external hard drive. Non-technical users are likely to make management mistakes, which could result in the loss of critical data, such as a tax return, customer database, or an irreplaceable digital photograph.

In this thesis, we show how to provide transparent storage management for home

and small business users. We introduce two new systems: The first, PodBase, transparently ensures availability and durability for mobile, personal devices that are mostly disconnected. The second, SLStore, provides enterprise-level data safety (e.g. protection from user error, software faults, or virus infection) without requiring expert administration or expensive hardware. Experimental results show that both systems are feasible, perform well, require minimal user attention, and do not depend on expert administration during disaster-free operation.

1.1 PodBase

Many households use multiple personal electronic devices, such as mobile phones, digital cameras, MP3 players, and gaming devices, in addition to desktop and notebook computers. As the number of devices in a single household increases, the task of managing data stored on these devices becomes an increasing burden on the user. We identify two aspects of home storage management that we believe are particularly problematic and time consuming for users: ensuring durability of data, and ensuring the availability of data.

Ensuring durability requires that the loss or failure of a device does not result in the loss of any of the user's data. For even a single home computer, ensuring that data is durable is an onerous task, and the situation is getting worse as the number and diversity of devices increase. A user must keep track of all devices that need to be backed up and perform the appropriate actions on a regular basis. Anecdotal

evidence suggests that many users fail to ensure the durability of their data [62, 76]. Thus, users face the risk of data loss, just as their dependence on digital information is increasing.

Ensuring availability of data on all of the devices where it may be needed is equally difficult. A user must regularly connect and synchronize devices to ensure, for instance, that changes to her address book are propagated to all communication devices, and that additions to her music library eventually are present on all devices capable of playing music. Achieving this goal manually requires foresight on the part of the user, as they must anticipate which files will be needed where, and replicate files accordingly.

With enough time and effort, a user could perform manual actions that solve all of the above problems. She could manually backup data to offline media and copy data to where it should be available. However, any one of these tasks can be laborious to accomplish, not to mention all of them. Performing these tasks correctly, regularly and in a timely manner is difficult today, and it will become increasingly difficult as users accumulate more data and devices.

Home users need tools to help manage the burden of storage management. Ideally, these tools would transparently provide durability of a user's data without any explicit action on the part of the user. As long as a user provides enough storage then their data should be durable. Likewise, the free space on each device should be managed so that data is available on the devices where it is likely to be used.

Existing storage management tools only partially address these problems, or are only applicable to particular devices and operating systems. For example, many portable devices come with vendor provided software that syncs specific file types with that particular device. Others require that the user run a specific operating system (e.g., Apple’s Time Machine [110]) or require the user to install a non-standard file system on their storage devices (e.g. Ensemblblue [77]).

Existing solutions also require expertise and attention on the part of the user. For example, a user might have to set up a dedicated server, and then configure his devices to periodically execute a task that syncs or backs up their files. Setting up such a system takes significant effort on the part of the user, even if they possess the knowledge to do so.

As such, there is a gap between the services a user would like to have, and those that current tools provide. In order to bridge this gap, we built *PodBase*. PodBase automatically manages a user’s data on existing personal devices in an automatic, decentralized, transparent, device- and operating system-independent manner. The system takes advantage of unused storage space and exploits incidental pairwise connectivity that naturally occurs among the devices, e.g., via Wi-fi, Bluetooth, or USB. PodBase uses a linear program driven planning engine, in order to replicate data through multiple disconnected devices without user intervention.

1.2 SLStore

Enterprise data management policies are designed to protect data from a variety of threats. Users may accidentally delete or overwrite critical data. Bugs in device drivers, operating system or applications may cause data to be corrupted or deleted. Hardware faults may leave storage devices unable to read part or the entirety of the information they store. Computers may be infected with a virus or attacked by an adversary, potentially resulting in the corruption or loss of data and programs stored on a disk. Finally, a catastrophic event like a fire or an earthquake could wipe out an entire site.

Enterprises guard against these threats by employing a dedicated, professional, administrative staff who use a combination of redundant storage techniques. Daily snapshots, incremental backups, periodic full backups, off-line and off-site backup storage are typically used in an enterprise environment to protect data. Such a comprehensive data protection strategy requires expertise to plan, and significant human resources and expensive equipment to execute.

Unfortunately, small businesses and home users often cannot afford the required expertise, human resources or equipment. Instead, these classes of users must resort to simpler solutions with serious shortcomings. These shortcomings may be a vulnerability to software faults or users error. Or it may be the requirement that the user diligently perform periodic manual actions (such as burning DVD's or rotating tapes), outside of their normal job, which are likely to be forgotten. These

limitations hamper data protection, just as the dependence of individuals and small businesses on digital information increases. It is essential to find automated solutions to the problem of data protection for individuals and organizations that cannot afford a professionally-managed solution, but that are nonetheless vulnerable to data loss. Many home office users and small businesses depends on irreplaceable digital information such as tax data or customer databases.

In order to address this problem, we built *SLStore*, a storage system for homes and small business environments, which offers data protection comparable to an enterprise IT infrastructure without requiring expert systems administration, significant human resources or expensive equipment.

A key technical contribution that lies at the core of SLStore's design is the concept of *storage leases*. While the concept of storage leases can be implemented by a variety of storage services, we focus on a particular storage device based implementation. A storage lease device is a storage device that protects data from accidental deletion and corruption for a pre-determined period of time, similar to an off-line storage device but without requiring mechanical action. Storage lease devices can be implemented by extending today's disk firmware, and are backward compatible so that a portion of the storage device can be used for normal data and another portion can be used for backups.

Given a set of devices that provide storage leases, SLStore still needs to implement a full data management solution in order to be useful. In particular, SLStore needs

to determine how to use the available storage resources to optimally safeguard data, while adapting to changes in the available resources and workload patterns. SLStore uses optimization techniques to determine and execute a data protection policy while requiring no expert and minimal user attention. Our approach also incorporates the use of cloud storage to protect data from catastrophic site failures. We show that SLStore is practical, reliable and easy to manage, even in the presence of hardware and software faults.

1.3 Contributions

In addition to building and evaluating the two systems, we made the following technical contributions.

- We formulate the problem of home storage management as a linear programming problem, including formalizing the concepts of availability and durability such that they can be directly optimized in for in the linear program. This allows replication strategies to be automatically derived instead of hand coded.
- We introduce a partial metadata replication protocol that allows eventual convergence of system metadata even in the presence of extremely space-constrained devices, which can not hold the full system metadata.
- We propose a new storage abstraction known as storage leases, which allow data to be protected for a specified period of time, even in the presence of operator

error, security compromises, and most software faults.

- We formulate the problem of determining a backup strategy as a linear programming problem, and produce a long-term backup policy based on available storage, and both current and predicted future storage requirements.

1.4 Thesis Overview

The rest of this thesis is structured as follows. Related work and background are discussed in Chapter 2. The body of the thesis is split into two main parts, each covering one of the two systems. In Part I, we present a feasibility study(Chapter 3), the design and implementation(Chapter 4), as well as the evaluation(Chapter 5) of PodBase. Similarly, In Part II, we present a design(Chapter 6), as well as an evaluation(Chapter 7) for SLStore. Finally, Chapter 8 presents concluding remarks.

Chapter 2

Related work

In this section we provide the basic background and related work required to put the research contained in this thesis in context. We begin by looking at the basic building block of storage, storage devices. We then look at the basic way in which data is managed, by storing it in a file system. Then, we discuss management tools and techniques which are built at a higher level on top of the file system. Finally, we enumerate the key contributions of the work of this thesis as compared to all previously discussed related work.

2.1 Storage Devices

Storage devices started out as expensive permanent storage for mainframe computers. Following the general trend in computing these expensive devices eventually became commoditized, the technologies improved, and these devices have made it in to the low end of the market. Now a flash chip that can hold many orders of magnitude more data than the first storage devices can be purchased for less than 20 dollars, and can be attached to a key ring. In this section, we will give a brief overview of how storage devices have evolved over time, and where the state of the art is currently. This will help motivate both the need for the work in this thesis, as well as set up trends and

ideas that will be used in the designs of the systems presented. The technologies discussed in this section is not meant to be an exhaustive list, rather it is meant to discuss the main storage technologies currently in use.

2.1.1 Technology Overview

Tape Storage

An early means of storing data was on magnetic tape [57, 58, 98]. Since then, tape has fallen out of favor as a primary storage media and has been replaced by other technologies which have better random access behavior. Tape still remains widely use as archival media, as it is a high capacity, convenient media for offline storage [13]. Tapes are also portable, and are often shipped to an offsite location that can serve as a backup in case of a catastrophic failure.

Hard disks

Rotational magnetic media was introduced [17] as a media that had better random access performance. Since then rotation media has served as the dominant form of secondary storage for many years. The price of disks has decreased and the capacities have increased. Currently, these are the most commonly deployed storage device, in both the home and the enterprise.

Commodity hard drives can be together to form an array of drives, known as RAID [18, 38, 75] arrays, which can be optimized for either reliability or performance.

RAID is an approach that makes multiple disks appear as a single device. RAID arrays can be configured in multiple ways that alter the cost/reliability/performance tradeoffs. In the simplest case, data is written to two underlying disks (known as mirroring), such that the failure of either disk does not result in the loss of data. More complicated RAID configurations improve performance through striping [19], allow the failure of more than one disk, or reduce the amount of data duplication using data coding techniques.

Flash Storage

In recent years, disks have become small enough to put inside of portable devices such as music players and portable hard drives. However, for some purposes the traditional rotational magnetic drives have been replaced by flash storage. Flash storage is smaller than traditional disks, has no moving parts and is more power efficient than disk. For these reasons, many common portable devices such as cell phones, cameras, and portable video game systems use flash storage. Today, some laptops are beginning to be equipped with flash storage, but these are more expensive than a comparable standard disk based system. These laptops however, enjoy advantages in weight, performance, and power consumption.

Write Once Storage

Write once storage is storage where once data has been written it is impossible to delete or modify. The most prevalent write once storage is optical media, such as CD's

or DVD's. Optical media are burned once and then can never be modified. Optical media is often cheap, and has high capacity for its size. It requires manual loading of disks into the system, and thus is not often useful for data that exceeds the capacity of a single disk. Currently commodity optical media lags well behind magnetic media in terms of total storage capacity. Write once storage has good durability properties in that it can not be overwritten or deleted. Thus, data is not vulnerable to security breaches, virii, or software faults after it has been written.

Virtual write once storage implemented in software is offered by major storage vendors [42]. This storage is used in applications where regulatory compliance requires data to be stored in a verifiable tamper proof way. These solutions are proprietary and are based upon rewritable magnetic storage. The implementation of write-once is most likely done in software in the storage appliance sold by the company for this purpose. As a consequence if there is a bug, or a security vulnerability in this code base, the write once property can be compromised.

Storage Networks

A storage area network (SAN) [37,108] is a collection of storage devices that have been networked together. On top of this collection of storage, a normal file system is run, and the storage looks exactly as a local storage device. SAN are generally deployed in clusters or data centers, as it eases the management of storage. Individual storage devices do not have to be moved around, each server requires only a connection to

the SAN. At this point it is all of the storage devices in the SAN are accessible with a block level interface, file systems at the client can then write to blocks located on any storage device. The mapping of clients to storage devices can be configured by the administrators of the system, or be done dynamically in software.

Network attached storage allows a client computing device to interact with remote storage devices. Normally it is clear to the client that it is remote device, and a network file system such as NFS is run to interact with the storage. Network attached storage generally provides both the storage and file system while a SAN is generally a block level interface. Often storage appliances deployed in SANs (e.g. [44]) implement advanced features such as snapshotting, and RAID.

2.1.2 Storage Device Deployments

The types of storage technology deployed today are dependent upon the user. Enterprises often deploy a variety of different, and more expensive technologies in order to get better performance, and reliability. Home users often have collections of commodity devices, which are cheap and less reliable. Small businesses fall in between, but rely more strongly on commodity devices than do enterprises.

While in this section we focused on different broad technologies and use cases, in the following sections we will discuss related work more closely and compare both of our systems to previous work in the field of storage.

2.2 File Systems

A file system provides the user with a view of their data organized so that it can be retrieved later. As such, we first discuss different file system designs, beginning with vanilla local file system, and moving towards those that more directly deal with the problem of management such as distributed file system, and file systems custom designed for personal devices.

2.2.1 Local File Systems

File systems provide a metaphor by which a piece of data is stored in a file, and groups of files are stored in directories. Directories, can also include other directories allowing a user to impose a hierarchical structure on their data.

A local file system [15, 43, 63, 71, 125] provides a mapping of the logical units of files and directories on to the basic block interface that storage devices export. The main work in local file systems has gone in two directions, the first is optimizing the data structures used in the file system to improve performance, and the second is to augmenting the file system with additional features, such as copy-on-write snapshots, or extensible file metadata. The most common file systems deployed today are HFS+ [43], NTFS [71], ext3 [15], and ZFS [125]. These are the default file systems shipped with the most popular operating systems. ZFS has gone the furthest towards adding manageability and reliability to the file system. It includes per block hashes, and allows mirroring and snapshots at the file system level. Below we discuss a few

research file system which have added additional features for the purpose of improving reliability or management.

Envy [5] is a system that protects users from file system bugs. It does this by implementing three separate file systems in parallel on top of a general block store interface. It uses majority voting when the results returned by each of the file systems disagree. In order to be space efficient, it stores multiple copies of file system metadata but not data blocks. Envy reduces the trust required of the file system by using redundancy. Envy still resides inside of a normal, insecure system and is thus subject to any attack that compromises the host operating system.

The Elephant File System [97] is a file system that keeps old versions of files for possible retrieval by the user. This allows the user to recover from software failures which might accidentally delete data or from human error. In order to be practical Elephant keeps only as many versions of data as is possible within the storage constraints. In order to do this Elephant uses a user-defined retention policy.

Venti [81] is a centralized storage system that implements a write-once read only interface. In Venti all data is written once to the underlying storage and from that point can no longer be modified. Venti is intended to be run on top of a RAID group in order to prevent the failure of a storage device resulting in the loss of data. Since the data in Venti can not be overwritten and is resilient to the failure of physical devices it is well suited for reliably backing up data. In its intended deployment, Venti included support for daily snapshots of data written to it which could then be

offloaded to archival media such as CD/DVD or magnetic tape. Venti assumes that the amount of storage provided is enough so that the underlying storage will never fill completely. It is unclear whether Venti, would be cost effective for a home or small business, given that storage can never be reclaimed.

Self Securing Storage (S4) [104] implements the abstraction of a storage device that should be used by systems that want to detect and recover from network based intrusions. S4 provides an NFS like interface that keeps a complete record of writes to blocks on the device for a specified period of time. While targeted towards network intrusions, the interface provided by S4 is sufficient to build other applications.

2.2.2 Distributed File Systems

A distributed file system allows a centralized logical view of all data, which eases management for the user. The data in the file system may be accessed by multiple client devices, and distributed across one or more storage nodes. In this section we discuss the proposals for general distributed file systems.

In a client-server distributed file system, the client is connected to a server and issues a request to the server to perform operations on a particular file. An example of such a traditional distributed file system is the Network File System [96] (NFS). In NFS, all requests go to the server, and when the server is not available, then the client does not have access to the files.

The Low Bandwidth File System (LBFS) [67] is designed for situations where

clients connect to the server over low bandwidth connections. It is based on the traditional NFS protocol, but includes modifications for more aggressive caching. It also leverages the redundancy of data that is shared between different files. If two files share a common chunk of data, this chunk can be cached and used whenever either file is read by the client. While LBFS uses aggressive caching, it is not designed for disconnected operation.

The file system included in the Locus distributed operating system [116], allowed files to be replicated on a per-file basis, and stored on any number of physical devices. This capability was used to allow flexibility in the placement of files. Since some physical devices may be offline when a file is updated, a version vector is used on a per file basis to keep a history of modifications.

The Andrew File System [45,52] caches data locally at the client to minimize the amount of communication necessary with the file server. In case of disconnection or disk failure this provides additional availability of the users data. If the server fails, the data in the cache may be recovered, preventing a disk failure from causing the loss of all data.

Up to this point, the distributed file system have expected clients being able to reach the server as the common case. However, some distributed file systems, e.g., [56,79,109] were specifically designed for disconnected operation as the common case. For these systems to function correctly, data must be available on nodes that wish to manipulate it offline, and there must exist mechanisms to prevent two users from

making conflicting disconnected edits, or to handle them when they do occur. We will discuss these file systems next.

Bayou [109] is a file system that allows clients to read and write any data objects that are locally cached. Bayou is not meant to be a general purpose file system, it is meant to be used by applications that are aware of the weakly consistent semantics that it provides, and which are capable of reconciling data that has been edited in a conflicting manner.

Coda [56], is a distributed file system that allows clients to hoard data locally to make files more available. This capability is used for both performance and to allow disconnected access. Disconnected operations are logged, and this log is replayed when a client is in contact with a server. When a conflict has occurred then the replay is aborted, and it is left to the user to resolve the conflict. In [56], the authors showed that conflicting edits were rare in a trace of users who were also developers of Coda.

The Ficus [79, 83] file system uses replication of data across many devices to improve the scalability, performance, and availability of data access for users. The main problem it solves is propagating updates, possibly conflicting, to files, which were made during periods of disconnection. Although data is replicated across many machines, it is not explicitly used for durability, and the optimistic semantics of the file system provide no guarantees about the consistency of replicas.

Farsite [3, 10, 11, 26] attempted to build a large scale enterprise file system from the

unused resources available in workstations. In order to do this, they had to develop techniques which allowed the use of untrusted and unreliable nodes in the system.

WinFS [122] is an unreleased file system for the Windows operating system. WinFS allows concurrent writes to shared files on different (possibly offline) computers and attempts to reconcile the resulting versions [70]. WinFS uses epidemic techniques to propagate metadata about replicated files.

Up to this point, none of the file systems discussed were specifically designed for personal electronic devices. Distributed file systems have been tailored to allow the participation of constrained personal electronic devices [51, 77, 80, 95, 100, 101, 126], and these are discussed in detail next.

2.2.3 Personal Device File Systems

Personal device file systems are meant to be run by a collection of devices, including some of which are portable special purpose devices. These special purpose devices include things like MP3 players, flash drives, and smart phones. These devices could be used by either home or business users. Some of the systems below are general, while some are more targeted towards the home environment.

Ensemble [77] allows devices with limited capabilities to participate in a more general distributed file system. It uses cache coherence techniques to notify clients when a replica has changed. Ensemble allows the specification of triggers, which can cause a particular action to be executed when a file of certain type is added to the

system. For example, if a new music file is added to the file system, it can be copied to an MP3 player automatically.

The Few File system [80] is a distributed file system designed to support offline collaboration between a group of users. These users may make their edits on different types of devices, and the connectivity of these devices may be limited. Files are divided into containers, where a container represents a group of files pertaining to a project. Whenever edits are made to a file, the update is propagated to all reachable devices via a pro-active event propagation mechanism. Events that can not be propagated via this mechanism, are spread epidemically whenever pair-wise connectivity exists. In addition, to the mechanism by which edits are propagated, the authors a new method of reconciliation based on operational transformation [8, 80]

Segank [101] is a file system that is designed for devices with heterogeneous connectivity abilities. It optimizes the placement of data so that each node will have access to fresh data, and so it is possible to consistently share data across users. It does this by utilizing a multicast primitive for locating data, and a lazy peer-to-peer mechanism for invalidation messages. The system can cope with completely disconnected operation, but is targeted towards an environment where all nodes have network connectivity so that users always have a consistent view of the file system.

PersonalRAID [100] is a system where the user carries a portable storage device between all of the disconnected devices that they have. The portable device acts as a conduit by which updates to the filesystem are propagated between disconnected

devices, and by which replicas may be transported. Using this mechanism provides devices a view of the same filesystem, and allows recovery from a limited number of failures. When using a machine that is running the PersonalRAID file system, the portable storage device should be connected while using the system (the system has mechanisms to cope when it is not, but this usage mode is considered exceptional).

Perspective [92–95] is a storage system that is designed for the home environment. It departs from the standard file system concepts such as files and directories and is instead based on a semantic store of a collection of objects. Each device specifies a view over the semantic store, which determines what objects are stored at each device. Whenever a new object is introduced, it is propagated to devices that have a view that contains that object. In addition to proposing a new basis for organizing data, the members of this project have performed several user surveys which capture the home user’s experience with their storage devices [91,92]

The Roma system [107] provides a shared metadata service for user data. While not a file system, Roma can be used to build higher-level services that provide synchronization, consistency and increased availability. In theory, the metadata service could be used as a building block for a more general personal device file system. Roma relies on a centralized metadata server, which represents a possible single point of failure.

2.3 Backup Systems

In this section, we look at systems that target the problem of ensuring the durability of user data through backup. These systems are built on top of the file system layer, and often run as user level applications.

2.3.1 Individual Backup

Commercial remote backup systems [16, 47, 48, 65, 102, 123] allow the user to upload their data to a commercial server, or group of servers, where it is stored on their behalf. These services provide client software that uploads files to the service, and allows the restoration of data from failed devices. These services require the user to pay a monthly subscription fee. Many of these services provide the user some additional functionality, such as remote access to backed up data via the web, or the ability to share photos that are stored on their servers. Cumulus [114] is a system that uses general purpose cloud storage [88] as the data store for backup. Instead of relying on existing cloud backup services it uses its own specialized backup data structures, which are stored in the cloud. In this thesis, we will leverage cloud services, to increase the safety of data.

Pastiche [21] removes the reliance on a commercial service by providing cooperative backup to other user's machines over a wide area peer-to-peer network. Pastiche automatically copies data to other machines in the network, and minimizes the amount of data that must be copied by leveraging overlapping data present on the

participating devices. In Pastiche, the other nodes are not trusted to retain data, so the system must periodically check to make sure backups are being stored. The authors developed an additional system, [22] in order to enforce fair sharing of storage resources.

Friendstore [111] allows users to form a backup network among a group of users who are mutually trusting. They use this network to store two additional copies of each piece of data. Friendstore does not protect the user from bugs or malicious code deleting data. Friendstore does not vary the backup policy depending on the amount of data to be stored, or the available capacity in the system. In contrast to Friendstore, PodBase and SLStore do this automatically.

Time Machine [110] is an automatic backup utility included in Apple's OS X 10.5 operating system. It allows all Apple machines in a household to automatically backup to a single disk over the network. Windows 7 includes a basic backup and restore utility [119] that allows the user to backup their data to a secondary storage device. Windows Home Server [120] includes more advanced functionality that allows a single computer with potentially several hard drives to act as a home storage server. All machines can be configured to back up to the server. Additionally, users can access data that is explicitly shared via this server remotely. As part of the operating system, these systems can be compromised by user mistakes, virii, or bugs in the operating system.

Drobo [27] is a network attached storage server targeted to the home user. It is

meant to provide the user with a single device, which contains multiple disks, where all of their data is stored in a fault tolerant way. It allows the flexible addition of additional storage, and redundantly stores the users files across the different storage devices contained within it.

2.3.2 Enterprise Backup

In an enterprise, data protection is a core IT task. To minimize chances of data loss or implement legally mandated data retention policies, a combination of techniques are employed. Often enterprises use a tiered storage model where data is stored on backup servers, and then eventually moved to offline tape after a certain period of time. Servers often use some form of RAID configuration [75], and on-line snapshotting (e.g., NetApp WAFL [44], Sun's ZFS [125], or Windows VSS [115]). Off-line copies of the snapshot data are then created by copying the data onto tapes, and archived manually or using a tape robot. Lastly, tapes or disks are transported periodically to a safe location to create off-site copies. However, this professionally managed, expensive tiered system of backups is not practical for home and small business users.

Most large storage companies sell desktop backup clients which are intended to be used with that company's storage infrastructure. For example, storage companies such as EMC [30], NetApp [68], Sun [106], and HP [46] sell a variety of backup products that are compatible with their storage appliances, as well as offering offsite backup for corporate customers. These solutions attempt to minimize the adminis-

trative burden of keeping data backed up for a company with a large number of client machines. These solutions are expensive and out of reach for most homes or small businesses.

In addition to proprietary enterprise solutions there exist free open source alternatives that allow the setup of a backup server to which all clients back data up to, e.g. [4, 41, 127]. These solutions often include features that rival the proprietary solutions, but are less well supported and more complex to set up and administer. Significant expertise is required for setting up, configuring, and adequately provisioning such a system.

2.3.3 Archival Systems

While backup is concerned with the ability to recover from a failure, archival is concerned with the long term integrity of data. These two concerns are related, but some systems focus on the latter problem rather than the former.

NetApp's SnapVault [42] allows the creation of compliance volumes which allow data to be retained for a specified time period for regulatory compliance. It comes in two types, one which only allows deletion of entire volumes after a specified period of time, or one that allows deletion of files after a specified period of time. These features are implemented as part of NetApp's file server OS.

The LOCKSS [60, 61] project aims to preserve digital data stored in libraries by creating and maintaining replicas across different sites. The goal of the LOCKSS

project is to prevent the corruption of replicas over a long period of time using active maintenance and checking of replicas. LOCKSS, is targeted at a different environment then either of the systems in this thesis.

2.4 Synchronization

Synchronization tools attempt to ensure availability of data, and reconcile data stored on two or more devices. Synchronization systems can be divided into three different types, the first is general synchronization, the second is device or data type specific, and the final is replication frameworks on which synchronization can be implemented.

General synchronization tools, e.g. Unison [78, 113] or rsync [112], synchronize directories stored on different storage devices. Unison attempts to reconcile replicas that have diverged due to concurrent edits, which is orthogonal to the problem of availability addressed in this work.

There are commercial services (e.g. [28, 105]) that provide synchronization of personal data at a remote server. Like the remote backup services already discussed, a user's data is stored on a remote cluster of machines, and a small piece of software running on clients synchronizes files in specially designated directories. Apple's MobileMe [64] provides a small amount of storage that appears as a network drive on all of the user's machines running Apple's operating system. In addition, it allows the storing and sharing of photos and movies, as well as automatic synchronization of a user's contact and calendars. Microsoft operates a competing service [99], which gives

users access to a fixed amount of Internet accessible storage, as well as the ability to share files that are in a special folder. Windows Live Sync [121] and Live Mesh [59] allow users to sync folders across their machines.

There are many software packages [49, 72, 128] that manage and synchronize particular data types with specific devices. For example, iTunes [49] manages a users music library and synchronizes connected devices, such as iPods, with that music library. PDAs are bundled with software [72] that will synchronize particular content such as calendar entries, and contacts with the users desktop machines. Groove [40] provides a collaborative workspace for office documents that propagates file edits automatically among a group of users.

Finally, there is a class of systems which allow custom replication strategies to be specified. On top of this, synchronization can be implemented. Cimbiosys [82], is a system that provides selective replication of data between a set of disconnected devices. It allows each device in the system to define a filter, which specifies the data that it would like to receive. Applications are then built on top of this system that perform higher level functionality such as synchronization. Cimbiosys places constraints on connection patterns, ruling out the use of certain replication topologies. PRACTI [7] provides a framework for partial replication in large scale distributed systems. On top of this framework different replication algorithms can be implemented. Oasis [87] is an SQL-based data management system for pervasive computing applications, which can then be used as a building block for specifying replication policies.

The authors of [103] propose an approach called device transparency. Device transparency means that a unified view of all personal data is presented to the user. In order to achieve this they propagate metadata to all devices in they system. On top of this base, they include a design for rules which control data propagation.

2.5 Operations research techniques

In both systems presented in this thesis, we use operations research techniques, namely linear programming, to formulate storage management problems. This general approach his been proposed in other contexts, but not to the application presented in this thesis.

Keeton et al. [53] advocate the use of operations research techniques in the design and implementation of systems. Both systems in this thesis take this approach and use linear optimization to adapt to their environment. Other examples of this approach include: Rhizoma [124] and Sophia [117], which use logic programming to optimize cloud computing and network testbed environments, respectively. Conductor [118] is a recently proposed system that use linear optimization to optimize cloud computing based map-reduce computations in a competitive market.

2.6 Networking techniques

While the primary focus of this thesis is storage, the design of the systems, particularly PodBase, required an implementing an underlying networking layer. In this section

we describe networking related work.

The Unmanaged Internet Architecture [33–35] (UIA) provides a naming and routing service for personal electronic devices. It requires no configuration and allows rich sharing semantics between users. PodBase addresses the complementary problem of data management for personal devices. UIA could be used by PodBase to provide naming and connectivity among a household’s devices regardless of their present location.

In PodBase, replicas and metadata propagate through the network of devices in an epidemic manner [24]. Unlike epidemic multicast algorithms (e.g. [9]), PodBase is not trying to deliver the replicated data to a specific subset of the devices: since our primary goal is durability, it is often sufficient to create replicas of each data item on a specific *number* of devices.

Since PodBase shares data among a set of intermittently connected devices, it implements a form of delay tolerant network (DTN) [25, 31, 50]. PodBase can be viewed as a data management application on top of a specialized DTN.

2.7 Extending storage device functionality

Storage leases are an enhancement to storage device functionality. There are other projects that have looked into extending the functionality of disk drives. In particular, self-encrypting hard disks [1] protect data from being accidentally leaked when a device is lost or stolen by encrypting data before it is stored. Active disks [2, 84, 85]

and IDISks [54] allow application-specific functionality to be executed on storage devices. The goals and the type of functionality these projects are adding to storage devices differ from those of storage leases.

2.8 Contributions beyond related work

We now discuss the contributions of the two systems presented in this thesis beyond the existing related work.

PodBase differs from related work, in that it provides a comprehensive, transparent, and autonomous solution for home storage management. PodBase is built upon a novel adaptive replication algorithm, which successfully replicates data in arbitrary scenarios, even those it was not explicitly designed for. Unlike other proposals, such as distributed file systems, backup, and synchronization, PodBase addresses both availability and durability. Additionally, PodBase does not require changes to the operating system or file system, and operates completely transparently to the user.

SLStore provides a comprehensive and autonomous backup system for small business and home office users. It makes use of similar linear programming planning techniques as PodBase in order to select a backup strategy automatically based on the current resources, storage configuration, and workload. It is based on a novel abstraction known as storage leases, which allow data to be reliably stored even in the presence of software faults or user error.

Storage leases are the building block that allows SLStore to provide enterprise

class storage management. Storage leases protect data from operator error, security breaches and most software errors. Storage leases provide flexibility in reuse than existing off-line or write once media, and are more general than previously proposed systems, e.g., [104]. The design for storage leases includes facilities for securely embedding storage lease devices in untrusted environments, and still having guarantees about stored data.

Part I

PodBase

In this part, we describe and evaluate PodBase. We begin by presenting a feasibility study that was conducted before PodBase was built, which motivates the design of PodBase. We then present the design of the system. Finally, we present the evaluation of the system.

Chapter 3

Feasibility

In this section, we explore whether the goal of automatic storage management is feasible, given the set of devices, available storage, rate of data generation and frequency of device connections that occur in a given household. We focus on whether there are sufficient resources in terms of number of devices, connectivity and free storage space for the users to achieve the most basic goal of a management system: durability of data against the loss of any one device.

3.1 Trace Collection

In order to study the feasibility of building a system on top of a user's spare resources, we need data about the intended deployment environment. Unfortunately, there is little available data on the disk usage of personal devices such as desktops, notebooks and PDAs. Moreover, in addition to disk usage, our analysis also requires information about the frequency, duration, and bandwidth of connections between devices.

Since such data is not readily available, we gathered a trace by deploying a data collection program on the devices of a small group of participating users for approximately two months. During this time period, each machine reported all storage devices that were attached either directly or via USB. For each device, we recorded

the used storage, free space and total capacity at one minute intervals. We also periodically crawled the file system of each storage device to discover which files were stored and how often files were added, deleted, and modified in the file system. Additionally, we collected information about what type of network each computing device was connected to. The measurements were taken between August 18, 2007 and October 18, 2007 and included 11 households with 23 computing devices and 48 storage devices. There were 1,568,773 samples taken, and 971 disk crawls performed during this period.

3.1.1 Limitations

Ideally, we would like to have collected a trace from a large and diverse user community over a long period of time. However, logistical and privacy issues make this a difficult undertaking. We instead relied on the generous support of a small community of volunteers, namely the members of our research group and their friends and family.

In our group, at least one member of each household was a computer science researcher. Thus, there is a likely bias towards users who have an interest in technology and tend to surround themselves with electronic devices. They may also be more likely to accumulate a large amount of data. In the absence of better data, we believe that our trace data nevertheless gives a useful indication of the feasibility of our approach.

3.2 Results: Feasibility

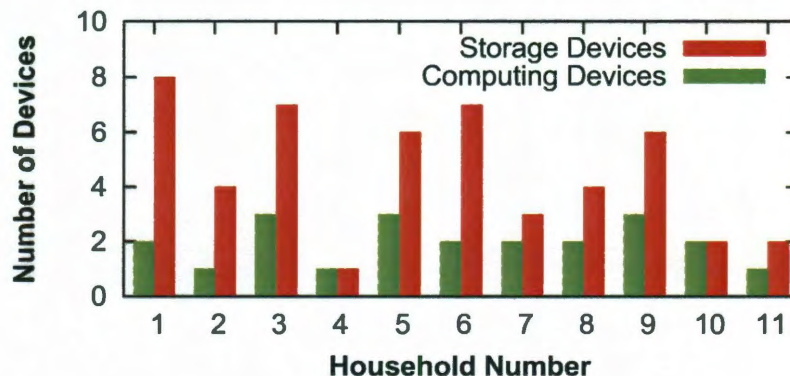


Figure 3.1 : Number of devices connected by user.

In order to build a distributed storage system, a user must have at least two devices capable of storing data. Figure 3.1 shows how many computing devices (desktop or notebook computers) and how many storage devices (hard drives, MP3 players, memory sticks) were present in each household during our trace collection. The results show that there are many households with more than one computer. Also, most households have additional storage devices beyond a single internal hard drive per computer. This shows that many households could benefit from automatic storage management.

Figure 3.2 shows the amount of free storage space in each household. Some households have significant amounts of free space, others have very little. In particular, household 3 has over 70% of its storage space used. For eight of the eleven households there is enough free space available to provide data durability (replicating all data on

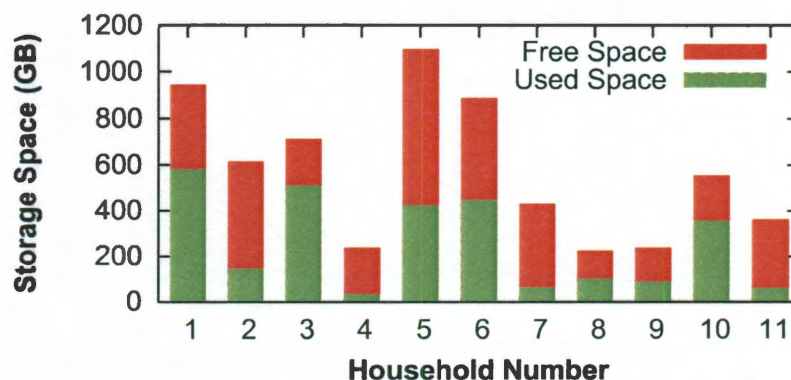


Figure 3.2 : Amount of used and free space by user.

2 devices); the remaining households would have to add storage in order to get the full benefit from the system. These households could simply purchase an inexpensive USB disk¹ and plug it in; the system would automatically use this new space to provide durability and availability.

Our result on the availability of free space is conservative, because it ignores data redundancy that exists on a device and across multiple devices. For example, there are nine MP3 players in our trace. An MP3 player typically contains a subset of the files that are already stored on one of the users' computers. Also, it has been shown that two machines running the same operating system have a significant overlap in data [21].

Next, we wish to gauge if there is enough connectivity to ensure that new data can be replicated in a timely manner. Adding more storage is relatively easy, since a user

¹320GB is the amount of space required for the most constrained user in this study. At the time of the study was conducted, 1TB disks were widely available for under 200 US Dollars

can simply purchase an inexpensive disk. However, if a storage management system required users to connect devices often, this would place a considerable burden on users.

To address this concern, we use the free space, data growth rates, and connectivity measured in the trace to perform a simulation. We begin with the system in stable state, and then have each device generate new data at the average rate from the trace. When two devices can connect to each other (either directly or via the network), they transfer data at the nominal rate of their connections. When a device generates data, it attempts to replicate the new data by transferring it to another device that has free space. We wish to measure the amount of time it takes for generated data to be replicated once created.

To do this, we replay² the connectivity among the devices of a single household from our trace. When data is generated it is put into a queue; the head of the queue is transferred first when a connection is present. We measure the time (in hours) that data waits in this queue before being replicated. If, at the end of the experiment, data has not been transferred, it is included as having waited since its creation time. We repeat this experiment for all households and aggregate the data.

The lower curve in Figure 3.3 shows the results. Approximately 30% of data can be replicated within one hour. This case occurs when the creating device is connected to another device with free space at the time when data is generated. Around 50%

²We begin the simulation at the time when all of the users' devices that generate data appear in the trace.

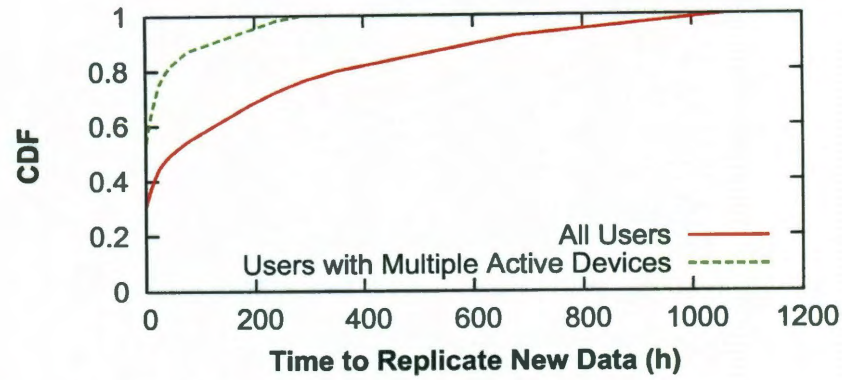


Figure 3.3 : Amount of time before generated data can be replicated in trace.

of data can be replicated in under 48 hours. Some data could not be replicated for a much longer period of time; most of these cases are due to three of our households who either only had one device, or had multiple devices but only activated them to install our monitoring application and then did not use them again.

The second, higher, curve shows the results with these three users removed from the trace. For the remaining nine households in our trace, it is possible to get quick replication for newly created data.

In summary, our study indicated, for most of our user community, there are enough devices, storage space, and sufficiently many connections to achieve durability in a timely manner. The remaining households would be able to achieve durability if they were to purchase and connect an additional storage device. In the following sections, we design a system, PodBase, that is motivated by the results of this feasibility study. In actual deployment, the simplifying assumptions made in this section do not hold, but we will show that building an automated storage management system is indeed

feasible.

Chapter 4

System Description

4.1 System Goals

In this section, we describe the target environment for PodBase. We then informally describe the goals of PodBase. Finally, we present definitions for the goals of durability and availability, and describe what it means to meet these goals.

4.1.1 Target environment

PodBase is intended for a household environment with one or more users and a set of shared personal electronic devices. The users are not necessarily technically sophisticated and would prefer not to be concerned with managing data and storage on their devices. Based on the result of our feasibility study, we characterize the environment as follows:

- Devices are periodically connected, such that any pair of devices can eventually communicate via a series of sequential pairwise connections.
- A device may fail or be lost at any time. However, the failure or loss of many devices during a short period of time is unlikely.

- Devices may be turned off when not in use; it cannot be assumed that any one device is always online.
- The system must be able to handle a wide range of usage patterns and device configurations, without attention from an expert system administrator.

An important aspect of the target environment is that most users don't have the expertise, interest or time to manage data and storage on their devices. They expect the system to do something reasonable automatically. Unlike a system designed for expert users (like the authors and readers of this paper), PodBase must be able to achieve its goals with little user expertise and attention.

4.1.2 Desired system behavior

PodBase aims to relieve users from having to worry about the *durability* and *availability* of their data. Durability requires that the failure or loss of a device not result in the loss of user data. Availability requires that each device store the latest collection of data *relevant* to that device. For example, each communication device should store the latest version of the address book and, subject to available storage space, a shared music collection should be available on all devices capable of playing music.

As an example, Alice and Bob share a household. Alice has a notebook, an MP3 player and an external USB hard drive. Bob has a notebook and a desktop computer at his office. Their home has a wireless network connected to the Internet via a broadband connection. During the day Alice and Bob bring their notebooks to their

offices and perform their daily work, such as writing documents and using email.

At night both return home with their notebooks and use them to surf the web, play games, or listen to music. Although they have important data stored on their notebooks, they rarely back up their data.

PodBase can automatically perform the following tasks without any explicit action by Alice or Bob:

- Every night, new or modified files are replicated, in cryptographically sealed form, between Alice and Bob's notebooks via the wireless network. (This works even when they are on vacation, e.g., when the pictures Alice uploads from her camera are replicated on Bob's notebook.)
- When Bob purchases a new CD and rips it to his hard drive, a replica of the mp3 file is later moved to Alice's notebook. When Alice connects her MP3 player to charge, it also receives the new music.
- Whenever Alice or Bob edit their personal address books, the changes are automatically propagated to their other communication devices.
- Whenever Alice's USB hard drive is connected to her laptop, additional replicas of the files and replicas on her laptop are made.
- Bob's office desktop is connected to his home via a broadband connection. Rather than transfer data using the slow connection, the system uses Bob's notebook disk to rapidly replicate data between home and work.

- When Bob’s notebook is running low on disk space (after removing any replicas), the system asks Bob if it should move not recently accessed movie files to Alice’s USB drive, which has plenty of space.

PodBase can recover from otherwise costly incidents. For example, imagine Alice’s laptop is stolen. With PodBase, she is able to restore the data on the lost device’s hard drive to her replacement notebook. When she connects over the wireless network to Bob’s notebook, some files from her stolen notebook are restored on the new device. When she later connects her new notebook to the USB drive, the remaining files are restored. Thanks to the replication between home and Bob’s office, they could recover all data even after a total loss of the home or office devices.

An important goal for PodBase is transparency: the system’s background activity should not affect users’ experience during normal operation. By default, the system does not remove user files, automatically propagate changes to user files or attempt to reconcile conflicting versions of concurrently modified files. Instead, PodBase maintains all versions of a file along with their modification history. Optional plug-ins can define file type-, device-, or application-specific consistency semantics.

4.1.3 Definitions

We now formally specify the properties of availability and durability that PodBase attempts to maintain. The primary goal of PodBase is to ensure durability. We want to guarantee that each file is replicated on as many devices as possible. As a

secondary goal, we want to maximize availability by placing copies of each file on devices where they are potentially useful, subject to available space. This ordering is not inherent to the system, rather, it is how we chose to prioritize them, based on what we believe are a typical user's priorities.

Let Π be the set of participating devices and F be the set of files that are managed by the system. For each device $i \in \Pi$, let S_i denote the amount of space available at i for storage of user files and replicas. For a set of files $S \subseteq F$, $size(S)$ denotes the amount of storage required to store S .

We assume that a map $\nu: \Pi \rightarrow 2^F$ is given which assigns, to each device, the set of user files stored in that device. The goal of a storage management system is to maintain a *file assignment* function $\psi: \Pi \rightarrow 2^F$ that maps each device to the set of both user and replica files that the device should store. At any time, the file assignment must satisfy

- $\nu(i) \subseteq \psi(i)$, user files are never moved or deleted from devices;
- $size(\psi(i)) \leq S_i$, the files stored on a device may not exceed the capacity of that device.

Given such a file assignment ψ , we define its *replication factor* as the maximum value k such that, for every file $f \in F$,

$$|\{i \in \Pi : f \in \psi(i)\}| \geq k ;$$

that is, ψ places each file in at least k devices. We moreover say that the replication factor of ψ is *optimal* if there is no other file assignment ψ' with a higher replication factor.

We also assume an *availability map* $\varphi: \Pi \rightarrow 2^F$ that assigns to each device $i \in \Pi$ a set of files that i should *preferably* store. The *availability score* of a file assignment ψ is defined as

$$\alpha = \sum_{i \in \Pi} |\varphi(i) \cap \psi(i)| ,$$

i.e. the number of file copies that match the preference expressed by φ .

In a desired *goal state*, a data management service places at each device $i \in \Pi$, a set $\psi(i)$ of file copies so that the following properties are satisfied:

Durability. The replication factor of ψ is optimal, i.e. files are maximally replicated in participating devices of the system.

Availability. From all the file assignments with optimal replication factor, ψ also has a maximal availability score α ; i.e. files are replicated in devices where they are useful.

For the purposes of PodBase, we define default availability mappings between well-known file types and devices that are capable of interacting with these file types. For example, devices capable of playing MP3 files will be more likely to store such kind of files. Advanced users can modify φ to more finely control replica placement.

4.2 Design of PodBase

In this section, we present the design of PodBase. We start with an overview of the system, then describe how users interact with PodBase and how it propagates its metadata. Finally, we show how optional plug-ins can modify the behavior of the system, and describe the security mechanisms used to protect user's data. In Section 4.3, we describe how PodBase replicates data and formally define the desired goal state of the system.

4.2.1 Overview

PodBase is implemented as a user level program. It keeps track of user data at the granularity of files. PodBase is oblivious to file and device types. However, PodBase supports a plug-in architecture, by which file type and device specific data management policies can be added.

PodBase distinguishes between *active devices* and *storage devices*. Storage devices include hard drives, media players and simple mobile phones. Active devices run the PodBase software and provide a user interface. An active device contains at least one storage device; additional storage devices can be connected internally or via Bluetooth or USB. The set of devices in a household form a PodBase *pool*. In each pool, there must be at least one active device, which runs the PodBase software.

Active devices communicate via the network and handle the exchange of data. Whenever two active devices communicate, a storage device is attached to an active

device, or two storage devices are attached to the same active device, we say that these devices are connected. Data propagates during these pair-wise connections.

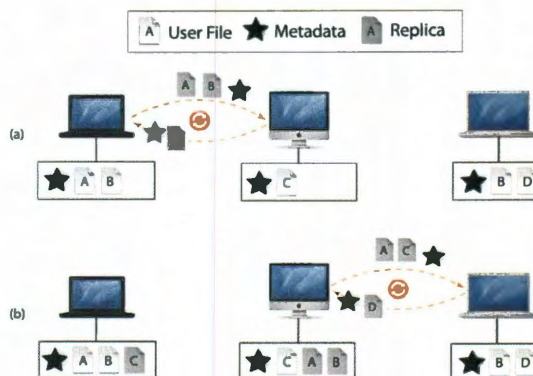


Figure 4.1 : Data flows through the system whenever connectivity occurs. (a) When the top two devices are connected, they replicate files. (b) Later, a new connection occurs, and data flows from the first node to the third node through the second node.

There are three different types of data on each storage device: (1) regular *user data*, (2) PodBase *file replicas*, and (3) PodBase *metadata*. Although logically separate, all of these data are stored in the device's existing file system. The PodBase replicas and metadata are cryptographically sealed and stored under a single directory.

Metadata describes a device's most recent view of the pool's state. Included in the metadata is the set of known devices and their capacities, a logical clock for each storage device and a list of all user files that PodBase manages, along with their replication state. Capacity constrained devices may store only a subset of the system's metadata, as described in Section 4.2.3.

Some of the space on a device not occupied by user data or metadata is used

to replicate files for durability and availability. User data has priority over replicas. PodBase continuously monitors its storage use and seeks to keep a proportion f_{min} of the device’s capacity free at all times.

When a file is modified by an application or the user, PodBase creates a new version of the file and replicates both the old and new version independently. Plugins (see Section 4.2.4) can be used to automatically apply consistent file updates or reconcile conflicting versions in a file type-specific manner. Users can manually retrieve copies of old versions or even deleted files.

4.2.2 User interaction

Next, we describe how users typically interact with PodBase. Though PodBase is designed to minimize user involvement, some interaction is required. Moreover, interested, tech-savvy users have the option to change its policies.

Device Registration. When a new device is connected for the first time, PodBase asks the user if the device should be added to the storage pool.

Device Deregistration. A storage device may permanently disappear due to loss, permanent failure or replacement. If a device has not been connected for an extended period (i.e., a month), PodBase prompts the user to connect the device or else deregister it.

Data Recovery. When a storage device fails, PodBase can recover the files it stored. The user informs PodBase that she wishes to recover the data from a particular lost

device onto a replacement device or onto an existing device. The PodBase software on the recovery device then obtains copies of the appropriate files during each connection.

Externalization. By default, users and applications cannot directly access replicas stored on a device. However, users with the appropriate credentials can *externalize* replicas, that is decrypt and move the cleartext of a replica into the user file portion of the device. Alternatively, externalization can be automated using a plug-in.

Warnings. PodBase warns the user when it is unable to replicate files because there is insufficient storage space or connectivity, with specific instructions to buy an additional disk or connect certain devices.

4.2.3 Device interaction

When two devices are connected, they reconcile their view of the system and exchange data. First, the devices reconcile their metadata. Then, PodBase determines if any of the replicas on either device should be moved, copied or deleted. Next, we detail these steps.

Metadata contents

Metadata contains the following items:

1. **Vector Clock:** A vector clock [73], consisting of the most recent known logical clock values for each device in the pool. A device's logical clock is incremented upon each metadata change. When a device is removed from the system, its logical clock is set to a special tombstone value. Also, the metadata includes the most recently

observed vector clock of each device in the storage pool.

2. Connection History: A list of the past 100 connections that have been observed between each pair of devices, their time, duration, their average and maximum throughput, as well as the network addresses used by the devices.

3. Policies: The current policy settings. Policies can be modified by sophisticated users. Installed plug-ins (Section 4.2.4) can also modify the policies.

Items 1–3 are included in the metadata of all devices.

4. Set of user files: Keeps track of the user files stored on each device in the pool. The content hash¹ value, size and last modification time are recorded for each unique file. In addition, the content hashes of the last ten versions of each file are included (modification history).

5. Set of replicas: Keeps track of the replicas stored on each device in the pool. For each replica, its size, content hash value, and replica id are recorded.

6. Reverse map of unique files in the pool: Maps a content hash value to the set of files whose content matches the value. This mapping is used to determine the current replication level for each unique data file, considering that different files may have identical content. (PodBase de-duplicates files prior to replication.)

Each record in items 4–6 contains a version number, which corresponds to the device's logical clock at the time when the record was last modified. A small device may include only a subset of the records in items 4–6.

¹A second preimage resistant hash function is used

Metadata reconciliation

MMetadata reconciliation is straightforward in the common case when two devices that carry the full metadata are connected. They compare their vector clocks to determine which has the more recent metadata for each device in the pool. For each such device, the more recent metadata is then merged into the reconciled metadata.

PodBase also supports devices too small to hold the full metadata. (In practice, devices smaller than about 100 MB are excluded. This is a mild limitation, since smaller storage devices are increasingly rare.) Such devices hold the full metadata for the files and replicas they store, plus some amount of partial metadata about other devices.

PodBase ensures progress and eventual consistency of metadata, even if some devices are only ever sequentially connected via small devices. To this end, PodBase places metadata on a small device that are needed to update other devices. For this purpose, it checks the last known vector clocks of all devices. PodBase selects partial metadata subject to the available space on the small device, while ensuring that (i) metadata needed by more devices are more likely to be chosen, and (ii) a roughly equal number of metadata items are included for each device that the small device may encounter.

When reconciling any device L with a small device S , PodBase checks if the metadata on S can be used to update L . For a given device d whose partial metadata appears in a small device, all metadata are included that have changed within some

range of versions $i < j$ of d 's metadata. This metadata can be used to update L if L 's current metadata version for d is at least i and less than j . If so, PodBase merges the metadata about d from S into L 's metadata.

Replication

Once the metadata is reconciled, PodBase determines the actions, if any, that should be performed on the data. PodBase may *copy* a replica of a file, in which case the file is stored on the target device with a new random replica id (used to distinguish between replicas), while the original replica remains on the source device. A device may also *move* a replica, in which case the replica is stored on the target device with the same replica id and then deleted from the source device. Finally, a device may *delete* a replica, to make room for another replica that it believes is more important. During replication, data is transmitted in a cryptographically sealed form, and a hash of each replica's content is attached to ensure data integrity. How PodBase determines the actions that should be performed is described in Section 4.3.

Data recovery

After a device loss or failure, data can be recovered onto a replacement device at users' request. During each connection to another device, the replacement device restores as many files as possible, guided by the reconciled metadata. The most recent available version of each file is restored. Users can speed up the recovery process by connecting appropriate devices under the guidance of PodBase. The system notifies the user

when the restoration is complete.

Replica deletion

PodBase removes replicas when the free space on a device falls below f_{min} , the minimal proportion of a device's storage that PodBase keeps available at all times (by default, $f_{min} = .15$). When PodBase frees space, it considers the most replicated files first. Among files with the same replication level, PodBase first deletes replicas that have the lowest (randomly assigned) replica id among the replicas of a file, then the second lowest id, and so on. This policy ensures that different devices delete replicas of the same file only when a shortage of space dictates it. (PodBase never deletes the original or any externalized replica.)

When space is available, PodBase also stores partial replicas from aborted transfers, so that they can be later resumed. When space runs low, these partial replicas are cleared before any complete replicas are deleted.

4.2.4 Plug-ins

Plug-ins can be used to implement policies and mechanisms that are specific to particular file types, collections of files, device types or specific devices. Following are some examples of plug-ins.

Consistency: PodBase replicates each version of a file independently. A plug-in can be used to automatically propagate changes or reconcile concurrent modifications under a given consistency policy. There is a large body of work on consistency, and

powerful tools exist for reconciling specific file types, e.g. [32, 66, 90]. Such tools can be integrated as plug-ins in PodBase.

Digital Rights Management (DRM): Media files stored on a user's devices may be protected by copyright. Usually, copyright regulations allows users to maintain copies on several of their personal devices. However, if restrictions apply, then the policies appropriate for a given media type can be implemented as a plug-in.

Archiving: A plug-in can automatically watch for large, rarely accessed user files (e.g. movies). If such files occupy space on a device that is nearing capacity, the plug-in suggests moving the collection to a different device with sufficient space. If the user approves, PodBase automatically moves the files.

Content-specific policy: A content-specific plug-in can, for example, replicate and automatically externalize mp3 files on devices capable of playing music. Moreover, the plug-in can select a subset of the music collection for placement on small devices. For instance, when replicating music on a device with limited space, a plug-in may select the most recently added music, the most frequently played music, and a random sample of other music.

As a proof of concept, we developed a plug-in that automatically externalizes replicas of mp3 files and imports them into iTunes. The plugin required around 100 lines of Java code, and two simple OS specific AppleScript scripts to interact with iTunes.

4.2.5 Security

PodBase uses authenticated and secure channels for all communication among devices within a pool. When a device is introduced to a PodBase pool, it receives appropriate key material to enable it to participate. Users have to present a password when they wish to interact with PodBase. Metadata and replicas are stored in cryptographically sealed form when stored on devices, in order to minimize the risk of exposing confidential data when a device is stolen. Moreover, PodBase respects the file access permissions of user files – encrypted replicas can be externalized only by a user with the appropriate permissions on the file.

The strength of PodBase’s access control within a household is designed to be at least as strong as the access control between different users on the same computer. If stronger security isolation is required between devices or users, then they should not join the same pool. For instance, if a user’s office computer contains confidential material that must not leave company premises, then it must not join the user’s home PodBase pool.

4.3 Replication

Whenever two devices are connected, PodBase needs to decide which files, if any, should be deleted, copied, or moved between devices. The decision is based on the latest reconciled metadata available to the PodBase agent.

4.3.1 Greedy replication

The first implementation of PodBase used a greedy replication algorithm. This algorithm considers the current replication state and the available space on the connected devices, and it only copies (i.e., never moves or deletes) replicas of files. A device obtains a new copy of a file whenever the device finds that file on a connected device, the file is under-replicated according to the current local view of the system state, and the device has enough space to store a copy of the file. Files that should be replicated for both durability and availability are replicated first, for durability alone second, and for availability alone last.

This simple algorithm can be trivially shown to monotonically increase the number of replicas and converge to a stable state. During a preliminary user study, the algorithm performed well in most households, but tended to get stuck in sub-optimal replication states in certain cases, for example:

- Once the algorithm has placed replicas for durability, it is unable to move the replicas if the placement turns out to be suboptimal for availability.
- When users carry a storage device (e.g. a notebook) between office and home, the algorithm is unable to use the device in the most effective way, namely to temporarily store replicas for transport between the two locations.
- When the space occupied by user files increases or a device fails, the algorithm cannot re-distribute replicas. In general, a re-distribution may be necessary to

ensure an even replication level, and to prioritize durability over availability.

- Two devices that initially interact replicate each other's files. If a third device is later connected, no room may be left for its files on the original devices. Although there may be plenty of space on the third device, the algorithm cannot replicate the device's files.

Since generalizing the greedy algorithm to cover these and other cases proved difficult, we chose to redesign the system to use techniques from operations research. We pose replication as an optimization problem and, using a linear programming solver, compute a replication plan to achieve the optimal state.

This approach minimizes the assumptions we make about the environment (i.e., the range of configurations and usage patterns), and defers most decisions to runtime, when the actual environment can be observed. The following subsections describe the approach in more detail. However, we first examine a more detailed concrete example of the shortcomings of the greedy algorithm.

Detailed Example

In order to further illustrate the shortcomings of the greedy algorithm, we give two more detailed examples where the algorithm does not perform well. The first scenario, is when two devices, A and B , are connected, B may be filled with availability replicas (least priority) from device A . When A is later connected to another device C that has unreplicated files (high priority), it cannot replicate these files. It has greedily

replicated low-priority files and cannot undo that decision.



Figure 4.2 : A small device that periodically connects to different devices has the potential to act as “communications device”, transporting replicas between devices that cannot directly connect or are connected via a slow link.

Second, as another example, if A and C have much larger capacity than B but A and C are never directly connected, as in Figure 4.2, then only a subset of files with a combined size up to the capacity of B can ever be replicated between A , B and C , no matter how often B is connected to A and C . A smarter replication algorithm could move replicas from B to either A or C , allowing B to be used as a “communications device” that carries replicas between A and C .

These examples are not exhaustive, and are meant to illustrate some of the cases where the current replication algorithm breaks down, and to motivate the need for more intelligence in the replication algorithm.

4.3.2 Adaptive Replication

In order to solve the problems of the greedy replication algorithm, we introduce a new adaptive replication algorithm. It is not an algorithm in the traditional sense, it specifies the current system state, and the goals, and then formulates a planning problem. This planning problem is then fed to a linear solver, and the resulting solution is used by the system.

In this section we describe the components of this approach by first describing the planning process, and then how the plan is used by the system to replicate. In the following section, we will explain the actual LP formulation of the problem in more detail.

Planning Process

We model the system state, as well as the effects of the actions, as a set of linear arithmetic constraints. To reduce the complexity and the size of the formulation, we group files into categories. All files that appear in the same set of devices (as user files or as replicas) are in the same category.

The system state is then encoded (the formulation is discussed in more detail in the following section) by specifying, for each category, the total amount of space occupied by all files in that category. This significantly reduces the number of variables in the problem formulation, which is no longer dependent on the number of files but only on the number of devices in the system.

To model the connectivity among devices, a graph is constructed with a link between each pair of devices that can potentially be connected. The link weight specifies the estimated *cost* of data transfer among the devices. This cost is calculated based on the maximum connection speed and the probability that the devices will be connected on a given day, based on the history of past connections. In this calculation, more recent connections are weighted more heavily.

Finally, we model the actions (copy, move, delete) PodBase can perform, and their effects on the system state. Because in general, devices only have pairwise and intermittent connectivity, a sequence of connections may be required in order to affect a certain state change (e.g., copy some files from A to B, and then from B to C). The formulation then encodes how the possible sequences of actions modify the number of bytes occupied by files in each category.

Encoding the problem this way enables us to symbolically describe all the possible plans that PodBase could execute in order to manipulate the distribution of files. Given this formalization, the goal is to find a plan that optimizes the desired goals.

Planning

The optimization involves multiple stages, narrowing the set of candidate replication plans in each. First, the maximal replication factor k is computed based on the available space in the system. Then, we optimize for durability by computing replication plans that can achieve a k -replication for all files. Next, we optimize for cost by narrowing the set of plans to those that minimize the sum of the link weights. In the next stage, we select among the remaining plans those that maximize availability. In the final stage, we select a plan that minimizes the number of necessary replication steps. PodBase then executes the first step of the resulting replication plan, by copying, moving or deleting replicas on the currently connected devices.

This description is slightly simplified. In practice, we do not consider plans with

more than three replication steps for efficiency (few interesting plans with more steps occur in practice).

The optimization favors cost over availability, because high cost plans are highly undesirable: they may rely on links with low bandwidth or rare connectivity. Notably, this choice still permits good availability, because the cost optimization generally leaves many candidate plans from which the availability optimization can select. The reason is that all plans involving the same set of connections have the same cost, and there is a combinatorially large number of such plans, corresponding to the different placements of replicas that can occur as a result of these connections.

The cost optimization does, however, eliminate plans that create more than k replicas, even if availability calls for more. To enable additional replication for availability to occur, PodBase changes the order of optimizations once the durability goal has been reached. In such situations, availability is optimized before cost.

Plan Execution

The output of the planner is a series of steps the system should take to reach the durability and availability goals. The system executes only the first step in this plan, afterwards it recomputes based on the changes that occurred in the system state. It also recomputes the plan periodically when conditions change. We next show how the entire process works, by stepping through an illustrative example.

4.3.3 Adaptive Replication Formulation

In this section we discuss the details of the adaptive replication formulation. We first introduce a set of concepts, and then provide more details of the linear programming (LP) formulation. Finally, we discuss how some implementation details impact the formulation, and how costs and weights are derived for the running system.

In developing this formulation, we developed several variations which proved not to meet our needs in terms of scaling to the problems sizes required by our user study. In this section we present only the final formulation used in PodBase.

Preliminary definitions

We will use terminology to describe each component of the linear programming formulation. For the sake of explaining the formulation as concisely as possible, these may differ slightly from the definitions used in the rest of the thesis. The first is a *device*, a device is simply a container for data. A device has a finite capacity which can be consumed by storing files or replicas. A device can have *connections* to other devices, these connections have a speed associated with them. A *configuration* is the current set of devices and connections.

The next concept is a *state*. A state is the current configuration of data stored on devices. A *step* describes the evolution of a state after a set of *actions*. An action transforms the current state into a new state. The set of possible actions is restricted by *constraints* on valid states. A *plan* is a set of actions to be taken over a series of

steps.

Constraints

A constraint describes the set of valid states that can exist. An example of a constraint is that no device can store more data than it has capacity for. Through the use of constraints the state space of possible plans is limited to those that are actually feasible. In this section we briefly describe that constraints that are used in the PodBase LP formulation.

- The amount stored on device can not exceed its capacity.
- A non-replica file can not be deleted or moved.
- A copy or a move action can only occur between connected devices.
- The amount of data copied or moved, must not exceed the original amount of data.

This simple set of constraints is enough to limit the actions of the system to those that are feasible in practice.

Actions

While it would be possible to encode complex actions in an LP, we choose to use only two actions (which can be combined to create a 3rd logical action). These actions are simply to either make a copy of a piece of data to another device, or to remove a copy

of a currently stored piece of data. The results of the action is reflected in the current state of the device after it has happened (when the plan is recalculated). From these two basic actions, moving a piece of data between two devices can be done by first copying to the destination and then removing it from the source.

More precisely, copy takes in a state, and then adds a device to the set of devices that is storing the data. Similarly, remove takes in the state and removes a single device from the list of devices storing the file. Associated with each action is a cost. This cost describes roughly the amount of time it would take to execute this action. The cost of a remove is free, as it can happen more or less instantaneously. The cost of copying between two devices is the amount of time that it would take to move the data around. This cost is based on a model, which will be described later in this subsection.

Goals

Up to this point we have constructed the building blocks of the LP. We have defined the valid system states, and actions which allow system states to change over time. Now we must provide a goal which describes the eventual system state to work towards. In developing PodBase we have developed several different goals, which can be combined to compute replication plans that have a certain set of properties. These goals are the following:

- Maximize durability: maximize the number of copies of all data (see Sec-

tion 4.1.3)

- Maximize availability: maximize the number of files which meet the availability goals (see Section 4.1.3)
- Minimize cost (time): minimize the cost of the sum of all actions in the plan
- Simplicity (minimize the number of steps): maximize the number of actions which occur in the first n steps of the plan. This is usually a re-optimization once other goals are optimized for, it is used to ensure the actions in the plan occur as early as possible. This is done interactively to find the minimum n for which the solution is feasible.

The goals can be combined using a multi-step optimization procedure. We will discuss how these are combined in Section 4.3.3.

Storage sets

In theory it would be possible to use the LP formulation with all of the above concepts on a per-file level, with one variable for each file. However, given the number of files in a PodBase deployment, this is not feasible given the current state of linear optimizers. Given this, we kept track of classes of files, rather than individual files. A class of files is the set of all files stored on the same set of devices in the same form. An example is all files stored as user files on A and as replicas on B. This set is associated with a number, which is the number of bytes in that set. An action can move data between

these classes by copying or deleting. This optimization limits the number of variables to be included in the LP to the number of classes, which is much smaller than the number of files in a PodBase deployment.

Deriving action costs

One key component of the above formulation is the cost of an action, particularly a copy action. We assume then in all but the current step, a connection will exist between all devices in the system, with a given cost. Assigning a cost of an action is based on the previous throughput seen between two devices. It must also take into account the the probability two devices will come in contact with each other. We use the previous connection pattern between two devices to estimate the likelihood that two device will connect in the future. We combine these two factors, and apply a discount factor to make older samples worth less. From these factors we assign a weight to a link.

The weight of a link in the formulation is derived in the following way. For each source, destination IP pair, we first define a value $throughput_{peak}$ which is the maximum observed throughput between that pair. Then for each day in which connections were observed a daily connectivity is calculated in the following way:

$$time_{connected}/time_{total} \tag{4.1}$$

This value is the percentage of the possible time these two devices were connected.

This percentage is then multiplied by $throughput_{peak}$ to measure the possible throughput between those two devices. Now a discount function $w(throughput_{potential})$ is applied which values samples based on how long ago they occurred in the past. For a sample of age x (in days) the discount function $w(x)$ is defined in the following way.

$$w(x) = 1/e^{-\log(0.1)/30*age} \quad (4.2)$$

The function exponentially weights the more recent samples higher than older samples. If the sample is older than 30 days then it has a value of zero.

The final expected throughput for a pair of devices is the sum of all weights divided by the maximum age. In the end the weights calculated result in the graph connecting all nodes in the system. In this graph links with higher weight are cheaper to use in that data can flow along the link in less time, and those with lower weight cost are more expensive. From this weight, the cost can be defined as the amount of time it would take a quantity of data to flow across a link. For example, a link with a weight of 100KB/s, carrying 1MB, would have a cost of 10s. Through the use of the costs, a total cost for a set of actions can be computed.

Optimization procedure

An individual LP formulation is a set of actions, constraints, and a fixed goal. However, in PodBase we often want to optimize multiple goals. To do this we iteratively solve multiple LP problems, and fix some components of the next problem

with values derived from the previous problem. While these building blocks can be arranged in any order, in our actual PodBase deployment we fixed them as described in the previous section. 4.3.2. We now look at an example that shows the whole process through a concrete example.

As an example, we illustrate how the planner finds a series of steps that reach the goal state. In this scenario, data must be transported by intermediate devices, and the system must correctly prioritize durability replicas. In this case we have 3 devices, A, B and C. The initial distribution of files is the following: A has 100 units of data, 50 of which is of a type that should be made available, B has 0 units of data, and C has 5 units of data. A and C are capable of storing 500 units of data, and B is capable of storing 50 units of data. A and C never directly connect to each other, but B is connected to both A and C. At the beginning, no data is replicated on any of the devices.

Formulating the problem

In this scenario, the goal state should be that C is storing a replica of all of A's data and vice-versa. B should be storing the 50 units of data for availability. The greedy algorithm will never arrive at this state. B would take 50 units of data from either A or C (whichever it was attached to first), and then the system would not be able to make any further progress.

The input to the planner is a description of the scenario, and a number of steps

Initial File Distribution

A=50,50

B=0

C=50

Device Capacities

A = 500

B = 50

C = 500

Connectivity

A - B = 1

B - A = 1

B - C = 1

C - B = 1

Maximum Steps

steps: 10

Figure 4.3 : Summary of system state and environment, which is used to generate a linear programming problem. This is the input that is needed in order to generate the initial planning step, which then generates a set of linear programming problems.

that are allowed in the final plan. An example the format used to describe the scenario is shown below in Figure 4.3. The goal state is based on the definitions in 4.1, and is to have all data maximally durable, the data that is useful for availability present on all devices, and with minimum cost.

From this specification, a linear programming problem is generated and passed to a solver. The result is used to determine the actions at each step, and must be repeated once for each step.

Generated Plan

The result of the above described process is used to generate a series of steps that PodBase must take in order to reach the goal state. We describe below the steps returned by the solver for the example scenario. The actual solution is a the value of certain variables in the solution of a linear program. We translate these results to English for the sake of clarity.

- Step 1: Copy 50 units of data that are not needed for availability from A to B
- Step 2: Copy 50 units of data that are not needed for availability from B to C
- Step 3: Delete all data from B; Copy 50 units of data from C to B
- Step 4: Copy 50 Units from B to A
- Step 5: Delete 50 all data from B; Copy 50 units of data needed for availability from A to B


```

Minimize
totalcost
Subject to
durable1 = 150.000000
available4 = 50.000000
durable3 = 100.000000
durable4 = 50.000000
durable5 = 50.000000
available1 = 50.000000
available3 = 50.000000
available2 = 50.000000
durable2 = 100.000000
.....
fa01 + fa11 + ..... + fabc11 - capacitya = 0 .....
cpya0b1 + 1 cpya1b1 + 1 cpyab0c1 + 1 cpyab1c1 + 1 cpyac0b1 + 1
cpyac1b1 + 1 cpyb0a1 + 1 cpyb0c1 + 2 cpyb0ac1 + 1 cpyb1a1 + 1 cpyb1c1
+ 2 cpyb1ac1 + 1 cpyba0c1 + 1
cpyba1c1 + 1 cpybc0a1 + 1 cpybc1a1 + 1 cpyab0c1 + 1 cpyab1c1 + 1
cpyc0b1 + 1 cpyc1b1 + 1 cpyca0b1 + 1 cpyca1b1 + 1 cpycb0a1 + 1
cpycb1a1 + 1 cpyac0b1 + 1 cpyac1b1 + 1
cpybc0a1 + 1 cpybc1a1 - cost1 = 0
.....
cost1 + cost2 + cost3 + cost4 + cost5 + cost6 + cost7 + cost8 + cost9
+ cost10 - totalcost = 0

```

Figure 4.4 : Simplified snippet of LP program generated by example problem. The objective function in this stage of the formulation is to minimize the total cost, given that the durability and availability goals are satisfied (the lines directly following Subject to). The next constraint enforces that the data the files stored on a device in a step never exceed the capacity of that device (the data being stored in each storage set is capture by the variables starting with f. Each letter following f is a device which the data is present on, and the following number indicate what step, and what type (0 = durability, 1 = availability)). Next, the variables which encode actions in the system are shown, and they are related to the cost. Finally, the costs are aggregated into the objective function. A variable above is always indexed by first the devices that it is / could be stored on, then the type of file (availability=1, durability = 0), and finally the step in the output plan. Action variables (above copy) also include a target device, in the case of the copy variables the step is omitted to shorten the variable names.

- Step 6: Copy 50 units of data needed for availability from B to C.

The final result of following this plan is the optimal goal state. All data is durable, and the data should be available is present on all devices. At the intermediate steps, no replicas are made for availability before the durability goal has been reached. Note, that while this example is simple, and the adaptive algorithm will work in much more complex scenarios. The units of data and edge weights are set to simple values for pedagogical reasons, and do not represent the values that one would expect while running PodBase.

Chapter 5

Experimental evaluation

In this section, we present experimental results obtained with a prototype implementation of PodBase. In a first set of experiments, we verify that the system behaves as expected. Then we present measured results from a deployment of PodBase in a small user community. This data allows us to assess how PodBase performs in practice, and provides experience with the types of devices users have, how often and in what manner devices are connected, and how user data is created and accessed.

Given these results, we will show that PodBase is practical, useful, and lessens the burden of storage management for ordinary users.

5.1 Implementation

PodBase is implemented as a user-level program written in Java. Most of the code (48,512 lines) is platform-independent, with the exception of a small amount (about 1000 lines) of custom code for each supported platform (currently Windows 2000 and higher, Mac OS X). This platform-specific code deals with the way different operating systems mount disks and name files. The current implementation requires that storage devices export a file system interface, and that all computing devices are able to run Java 1.5 bytecode.

Running PodBase on specialized platforms like cell phones or game consoles is feasible, but requires additional engineering effort. Since we were able to cover the majority of computing and storage devices owned by the users in the study, we felt that limiting ourselves to these two platforms and to storage devices that exported a file system interface was a reasonable trade off between engineering effort and our research goals.

In our deployment, computing devices contact a server that calculates replication plans. The server was provided to simplify the installation of PodBase and is not fundamental to the system. With an additional step, PodBase can be configured with a local solver.

5.2 Controlled experiments

5.2.1 Computation and storage overhead

PodBase crawls the file system of storage devices to determine which files exist and change over time. Each time a new file is discovered or an existing file is modified, the file must be hashed and added to the pool’s metadata.

Reading from disk and hashing a large amount of data (e.g., when a device is first added to PodBase), can be time consuming. We measured the amount of time the first crawl took when a new drive was added to the system. The measurements were taken on a 2.4 GHz Apple MacBook Pro, running OS X, one author’s primary computing device. The internal notebook disk contained 165,105 files with a total

size of 87.4GB. The initial crawl took approximately 5 hours to complete. Subsequent crawls, which only re-compute hashes for files whose time stamps and/or sizes have changed, took on the order of 10 minutes. The exact time required depends on the amount of new and modified data found on a device.

The size of the system’s metadata grows proportionally with the number of files and replicas managed by a PodBase pool. For the PodBase pools in our user study, the uncompressed metadata size ranged from 270MB to 2.5GB. This amounts to only a small fraction of the capacity of most modern storage devices. For the devices in our user study, storing the full metadata was possible in all cases. Smaller storage devices (e.g. older USB sticks or cameras) are supported in our design via the partial metadata mechanism.

Computing a replication plan requires invoking a linear programming solver. We currently use the CPLEX solver [23], a commercial solver, but CPLEX could easily be replaced by a non-commercial solver (e.g. clp [20] or glpk [39]). To verify this we ran a simple experiment and found that, for a typical household with 5 storage devices, using the free LP solver package clp [20] on a 2.4 GHz Apple MacBook Pro, it took between one and thirty seconds to calculate a plan, and the solver required 15MB of memory. These results show that it is feasible to locally compute a plan every time a device is connected, using non-commercial tools.

5.2.2 Pairwise transfer throughput

Next, we measure the performance of PodBase when two devices are connected over a variety of media. The goal of this evaluation is to understand how quickly data can be replicated over several common types of connections. Table 5.2.2 shows the speed at which PodBase is able to create new replicas for 87.4GB of files, when running on a 2.4 GHz Apple MacBook Pro. The experiment for DSL¹ was stopped early (after 5 days), and thus the transfer time is an estimate based on the measured throughput.

Connection	Transfer Time (h)	Speed (MB/s)
USB	9.50	2.00
Firewire	12.0	1.58
802.11g	23.5	0.81
Ethernet	10.5	1.95
DSL (up)	475 (est.)	0.049

Table 5.1 : Transfer throughput for different connection types

Most measured throughputs are lower than the nominal data transfer rates of the connections. PodBase deliberately limits the transfer rate to avoid interfering with other activities and to maintain responsiveness. Also, the PodBase agent performs various computations, such as hashing files to verify integrity and encrypting replica contents, which reduce the throughput.

Note that transferring data, as well as crawling disks and computing replication plans are background activities, which were designed to interfere with the responsiveness of the system as little as possible.

¹Uplink for this connection is 1Mb/s

5.2.3 Data restoration

Next, we test PodBase’s ability to successfully restore the contents of a lost device. We simulated the loss of a notebook after the replication phase in the previous experiment was completed. PodBase successfully restored to a USB hard drive all 211206 files (75GB) that were present at the time of the last crawl of the “lost” notebook. The restoration took 5 hours 27 minutes to complete, which includes decrypting all of the necessary replica contents.

5.2.4 Replication

Next we verify that, given sufficient connectivity and storage, PodBase replicates all files as expected. In the experiment, a single user with four devices (a desktop, laptop, iPod, and external hard drive) runs PodBase in the background while going about his normal daily business.

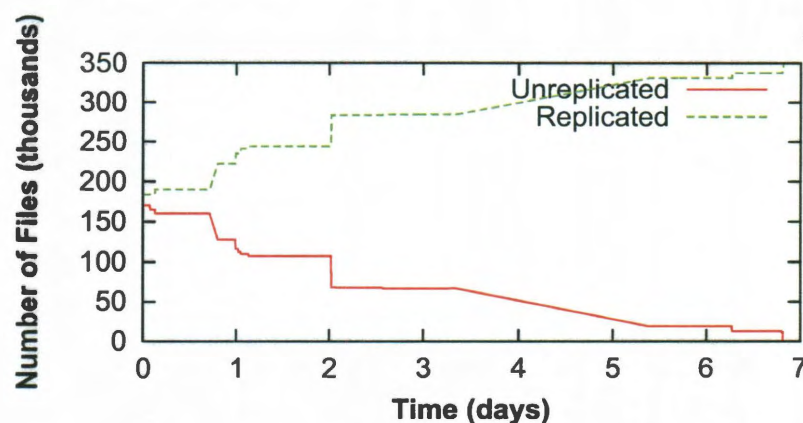


Figure 5.1 : Replication over time for an example user

Figure 5.1 shows the number of files that are unreplicated versus the number of

files that are replicated at least once, over the course of five days. At the beginning of the experiment (before PodBase was started), slightly more than half of the files were already replicated at least once on the devices. The reason is that two of the devices were computers with the same operating system installed. As time goes by, PodBase creates more replicas, and most files become replicated. The plateaus in the graphs correspond to times when no connectivity existed among any of the devices.

5.2.5 Adaptivity

In section 4.3.1 we described scenarios where the greedy replication algorithm performed poorly. We tested the new replication algorithm and verified that it was able to handle these cases gracefully. This result confirms that the new algorithm is able to adapt and perform well in tricky situations.

5.2.6 Partial metadata reconciliation

Next, we experiment with small devices that carry partial metadata. In our example, there are three devices: two full metadata devices, which never directly connect to each other; and a small device, which is connected to each of the other devices once per day. The small device is able to carry 100MB of metadata about other devices, and the total metadata size is 2GB (1GB per device).

Initially, the two large devices are completely unaware of each other; no new data is added after the experiment begins. It took ten days or 21 connections for the metadata on the two large devices to converge, which is expected based on the

relative size of the metadata and the small device.

This example show that metadata converges even in extreme scenarios. In our experience, small devices are rare and connectivity tends to be much richer in practice, leading to much faster convergence.

5.3 User study methodology

PodBase’s operation in a practical deployment is strongly dependent on the number and usage of devices, as well as the type, amount and usage of data within a household. To study how Podbase performs in actual usage, we conducted two user studies. The first was with the greedy replication algorithm, the second used the improved adaptive algorithm. In each, we asked ten members of our institute to deploy the system in their households and collected trace data over a period of approximately three months. We asked the users to, as much as possible, ignore the presence of PodBase and use their devices the way they would normally use them. If necessary, several users were given additional storage, these additional storage devices are indicated in each user study.

For practical reasons, the number of households and users in our study is small and covers a relatively short period of time. Moreover, at least one member of each household was a computer science researcher. Therefore, there is a likely bias towards users who have an interest in technology, like to surround themselves with electronic devices, and tend to produce and consume large amounts of information. As a result,

we cannot claim that our results are representative of a larger and more diverse user community, or a long-term deployment. However, we believe that our data gives a valuable glimpse at PodBase’s likely performance in practice.

In each user study, we collected anonymized data about file creation, modification and deletion on each device, when and where replicas were created, and which devices were connected at what times. We use these logs to generate the graphs used in the rest of this section.

5.4 User study 1

In the first user study, we evaluate PodBase using the simple greedy algorithm.

To start, we provide a brief overview of the households used in our deployment and the characteristics of the devices used in each.

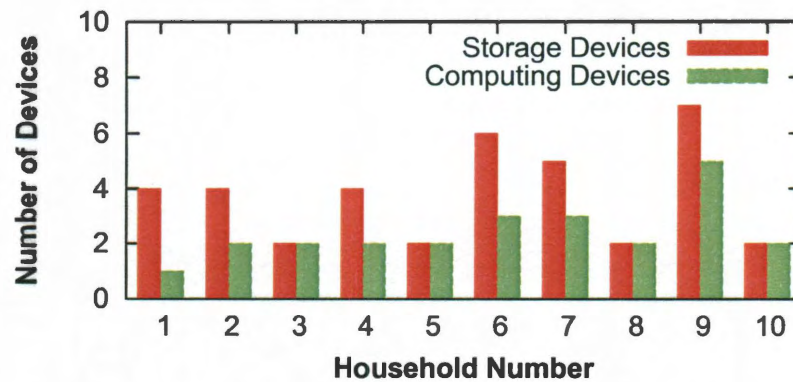


Figure 5.2 : The number and type of devices present in the deployment, by household.

Figure 5.2 shows the number of storage and computing devices in the PodBase pool of each household. The number of computing devices used in the households

ranged from one to five. Some users had no additional storage devices, while others had up to three. Households 1 and 4 received an additional one terabyte USB disk, which is reflected in the data.

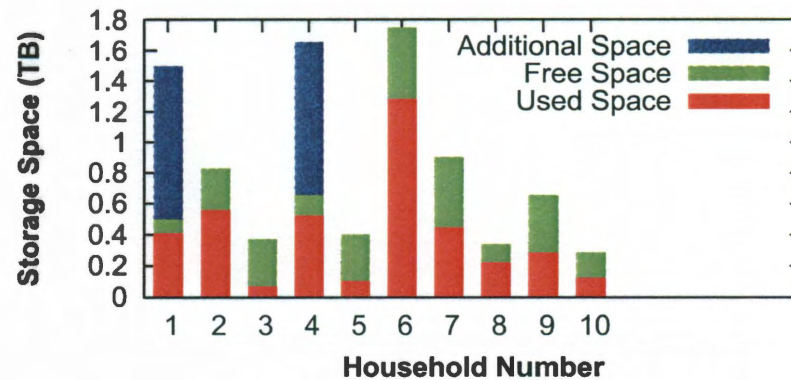


Figure 5.3 : The amount of storage capacity and free space present on devices before PodBase begins replication. Additional space corresponds to the USB disks households 1 and 4 were given.

Figure 5.3 depicts, for each household, the total size of the household's storage pool, divided into used storage and available storage at the beginning of the deployment and before PodBase was activated. The additional storage given to households 1 and 4 is shown as "additional space". After this addition, seven of the households had at least half of their total storage capacity available. This does not imply that the remaining three households cannot replicate their data; whether they can depends on how much duplication there is among their existing user files.

5.4.1 Replication results

In this section, we evaluate the performance of PodBase by looking at the state of the deployment at the beginning and the end of the trace collection. During the user study, the replication value of $k = 2$ was used for all users, and mp3 files were preferentially replicated for availability.

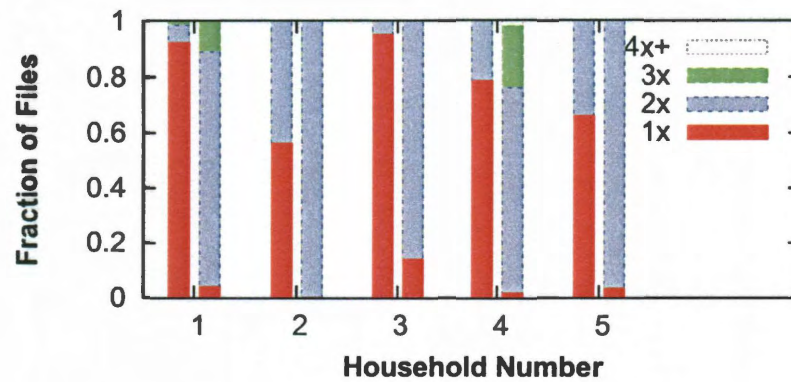


Figure 5.4 : The initial (left bar) and final (right bar) replication status of households 1,2,3,4,5

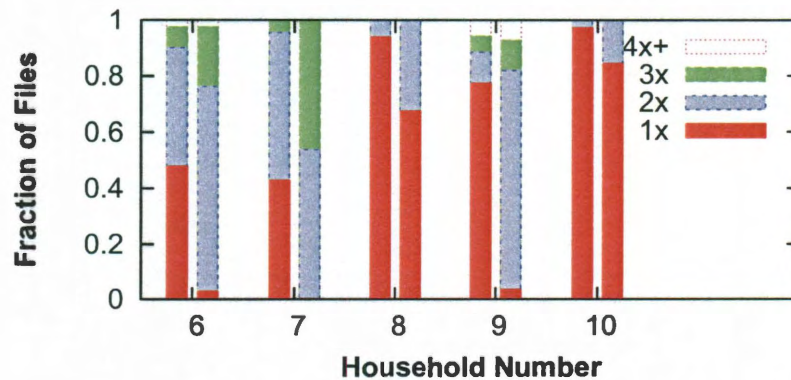


Figure 5.5 : The initial (left bar) and final (right bar) replication status of households 6,7,8,9,10

Let us look at the replication state of the system before the households ran Pod-Base. As shown in Figure 5.4 and Figure 5.5, many households had files that existed on only one device, leaving these files vulnerable to data loss if the device were to fail. Also, many households had a significant number of files already replicated, either as copies of the same file or different files with identical content.

At the end of the trace, seven households (1, 2, 4, 5, 6, 7, 9) had most (more than 95.8%) of their files replicated. The remaining, unreplicated files were recently created or modified and had not yet been replicated at the end of the trace. Households 3, 8 and 10, on the other hand, still had a significant number of their files unreplicated, namely 14.2%, 67.6% and 84.5%, respectively.

These three households each had only two devices: a notebook that was used at home intermittently and a desktop workstation in the office. The device pairs were connected over a DSL connection, but the notebooks were not active sufficiently long to allow full replication over the relatively slow broadband connection. Had these users left their notebooks turned on when not in use, or had they brought their notebooks to the office, their files would have been replicated. As a sanity check, we provided the two least replicated users with an external USB hard drive and asked them to connect it once to each of their machines for 24 hours (after the trace collection ended). By doing this, each was able to replicate 95% of their files.

5.4.2 Availability

A secondary goal of PodBase’s replication is to place replicas of files on devices where they are likely to be useful. In our deployment, we specified that mp3 files should be preferentially copied to devices that are capable of playing music.

In analyzing the trace, we found that relatively few mp3 files had been replicated. This is surprising, because several households had extensive mp3 music collections. Investigating further, we found that most of the households had already manually replicated all of their music files on the relevant devices. Thus, PodBase did not have the opportunity to improve availability.

Two users, however, originally had a significant number of mp3 files that were on their laptops but not on their desktops. PodBase replicated these files onto the desktops, and the mp3 plug-in described in section 4.2.4 had externalized the music files. This happened during an earlier run of PodBase (i.e., before our trace collection started), therefore it did not show up in our trace. The users had gained access to 426 songs (2.82GB) and 2611 songs (17.2GB), respectively, on their desktop computers that were previously stored only on their notebooks.

5.4.3 Replication latency and throughput

We next look at the maximal replication throughput in each of the households. Since all households had many files to replicate at the beginning of the trace collection, the rate at which data was replicated early in the trace is a lower bound for the total

replication throughput of a pool. This value provides a lower bound for the rate of new or modified data that a household could generate, such that PodBase would still be able keep up replicating the data.

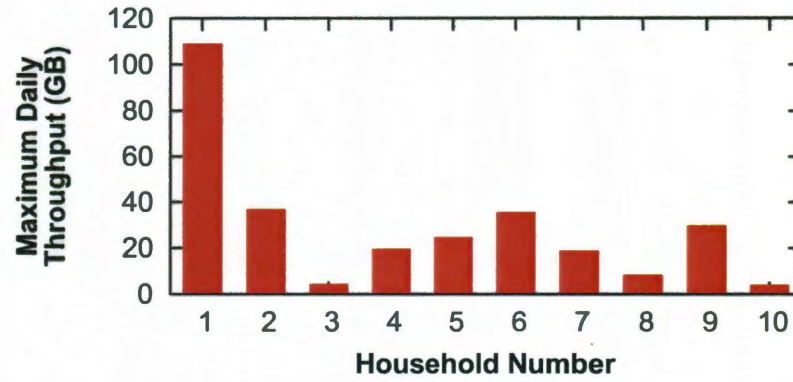


Figure 5.6 : Peak daily throughput for each household

Figure 5.6 shows that the peak throughput varies among the households in our deployment. The values range from 4 to 110 GB per day. This result shows that PodBase can keep up with a high rate of data generation.

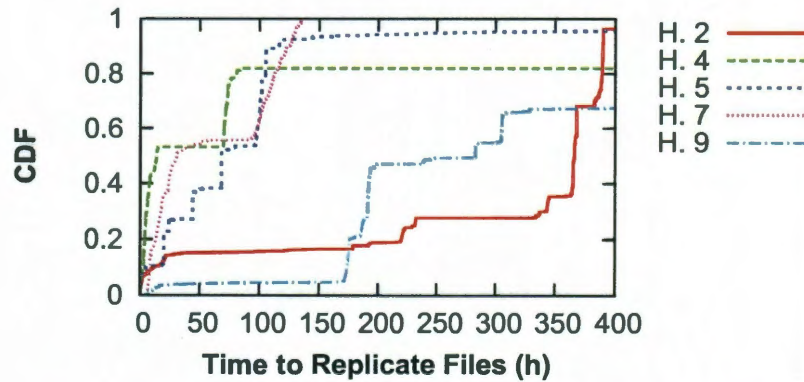


Figure 5.7 : Replication latency for households 2,4,5,7,9

We now examine the replication latency, the elapsed time until a new or modified file becomes replicated. We first examine those households with relatively short latencies. Figure 5.7 shows a CDF of how long it took to replicate a file. For households 2, 4 and 5, over 50% of files were replicated within approximately four days. Households 7 and 9 took longer because there were extended periods with no connectivity.

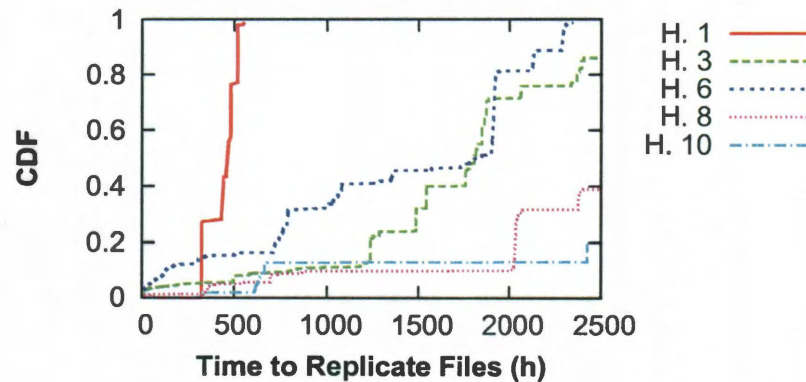


Figure 5.8 : Replication latency for households 1,3,6,8,10

Second, we examine the latency of the households who took significantly longer to replicate their files in Figure 5.8. Household 1 replicated very quickly once connectivity was established between a pair of devices. Households 3 and 6 replicated more slowly but consistently, which is due to the fact that much of the replication occurred over a DSL connection. As mentioned earlier, household 8 and 10 did not complete replication because they had only slow and intermittent connectivity.

We note that our measured replication latencies are conservative, because in most households, PodBase was busy replicating the user files found initially on the various devices during a large part of the trace collection. In steady state, PodBase would

have to replicate only newly created or modified files, which would reduce the latencies considerably. Nevertheless, PodBase was able to replicate data in a timely fashion in those households that had sufficient storage and where users cared to connect devices sufficiently often.

5.4.4 File conflicts

We previously argued that reconciling divergent replicas is an orthogonal problem to the one PodBase is trying to solve, and that we expect divergent edits to be rare in PodBase’s target environment. For these reasons, PodBase does not provide automatic means for reconciling conflicting copies of mutable files. However, plug-ins can provide this capability for specific file types. In the following experiment, we quantify how often file conflicts occurred in our deployment.

Nine of the ten households from our user study allowed us to log non-anonymized file modifications for one month. Among these households, each had on average of 633135 files. On average, 15184, (2.3%) of files were modified over the course of the trace. We saw a total of 449800 modifications across all of the nine households. We analyzed these modifications and found 62 (.001% of all modifications) files (by content) that had divergent concurrent updates.

Most of these concurrently modified files were text documents that were maintained by a revision control system, and users had checked them out on their notebooks or home computers. In this case, the revision control system provides a mech-

anism for reconciling the files. In the other cases, the files were email messages whose status (e.g. read, answered, forwarded) had changed on different devices. The status of these files is eventually synchronized through the mail server. Thus, a separate mechanism in PodBase is not necessary to handle these files.

5.5 User Study 2

In the second user study, we deployed PodBase using the adaptive replication algorithm. We present many of the same results as in the previous user study, as well as a comparison of the adaptive algorithm with the greedy algorithm used in the previous study.

First, we provide a brief overview of the households² used in our deployment and the characteristics of the devices used in each.

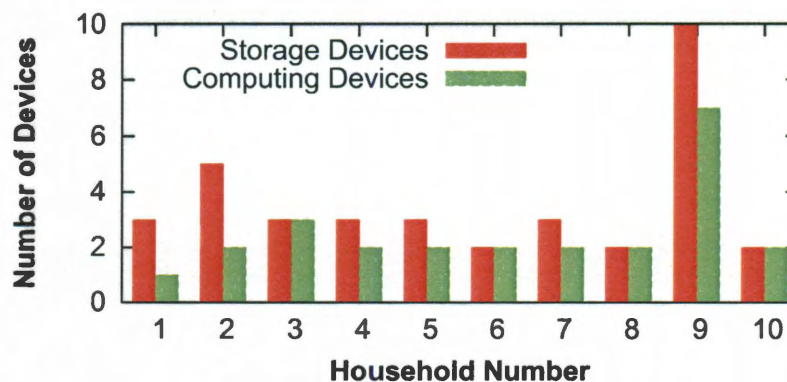


Figure 5.9 : The number and type of devices present in the deployment, by household.

²Note that the set of users in the first and second user study do not completely overlap, there is no direct correspondence between the household numbers in this and in the previous user study.

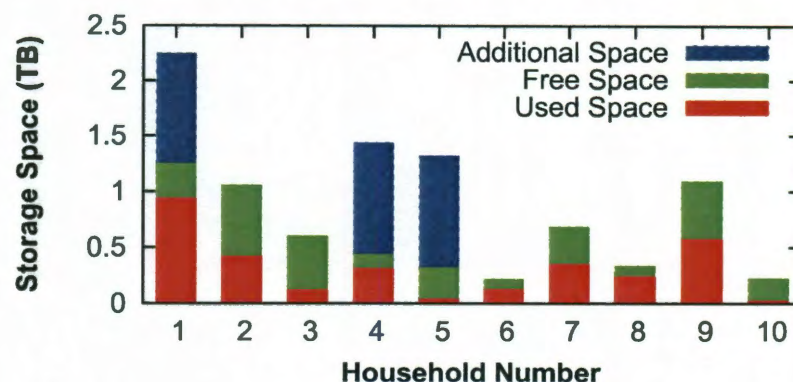


Figure 5.10 : The amount of storage capacity and free space present on devices before PodBase begins replication. Additional space corresponds to the USB disks households 1, 4, and 5 were given.

Figure 5.9 shows the number of storage and computing devices in the PodBase pool of each household. The number of computing devices used in the households ranged from one to seven. Some users had no additional storage devices, while others had up to three. Households 1, 4 and 5 received an additional one terabyte USB disk, which is reflected in the data. Household 4 has a virtual device that is backed by a cloud storage service to which that household subscribed. PodBase uses this device like any other, taking into account its capacity and connection bandwidth.

Figure 5.10 depicts, for each household, the total size of the household's storage pool, divided into used storage and available storage at the beginning of the deployment and before PodBase was activated. The additional storage given to households 1, 4 and 5 is shown as "additional space". After this addition, seven of the households had at least half of their total storage capacity available. This does not imply that the remaining three households cannot replicate their data; whether they can depends on

how much duplication there is among their existing user files.

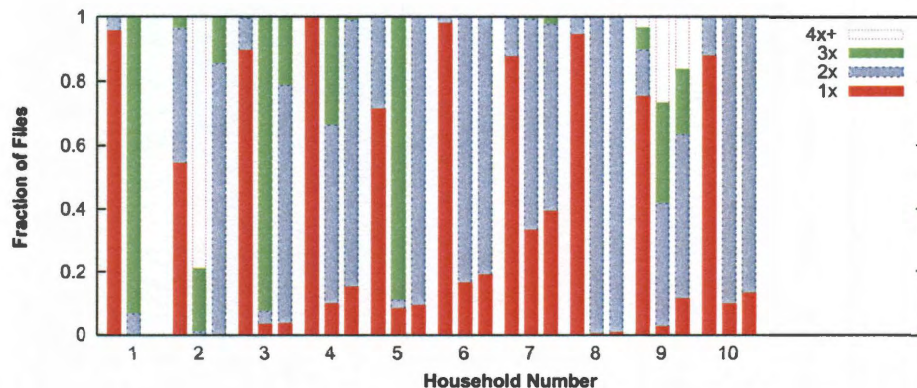


Figure 5.11 : The initial (left bar) and final (center bar) replication status of each household. Final results for the greedy algorithm (right bar) are shown for comparison.

5.5.1 Replication results

In this section, we evaluate the performance of PodBase by looking at the replication state at the beginning and the end of the trace collection. During the user study, the replication factor was dynamically chosen by the replication algorithm. An installed plug-in caused mp3 files to be replicated for availability on devices that can play mp3 files.

Let us look at the replication state of the system before the households ran PodBase. As shown in Figure 5.11 (left bars), many households had files that existed on only one device, leaving these files vulnerable to data loss if the device were to fail. Also, many households had a significant number of files already replicated, either as copies of the same file or different files with identical content.

The middle bar in Figure 5.11 shows the replication state at the end of the trace collection³. Five households (1–3, 8–9) had most (more than 97%) of their files replicated. With the exception of household 9, which had not quite finished replicating its original files, the remaining household’s unreplicated files were recently created or modified and had not yet been replicated at the end of the trace. Households 4, 5, and 7 were not able to replicate as much, as these households had only intermittent connectivity between a pair of their devices. These households had two devices that were well connected to each other, and one device that was either mostly offline or connected via a slow DSL connection. In these cases, all of the data was replicated between the well connected devices, but the data on the poorly connected device was not replicated fully.

Households 6 and 10 replicated as much data as possible but did not have enough space on one of their devices to fully replicate the remaining 19% and 10% of their files, respectively. In order to improve upon these results, the users would have had to add additional storage to the system. This could be accomplished easily by purchasing inexpensive additional storage.

As a sanity check we had users from households 4 and 10 bring in their notebooks in order to verify the limitations we describe above. Simply having household 4 bring its notebook into the office, where there was good connectivity between devices, allowed his data to be fully replicated. For household 10, we attached a one terabyte external

³The result for household 7 was obtained by re-playing the trace, because a bug was discovered during the user study that had influenced the final state of this household

drive to their computing device that had data to be replicated. After doing this, less than 0.5% percent of files remained to be replicated.

Several households (1–5, 7 and 9) were able to achieve a replication factor greater than two for some of their files. This is due to the replication algorithm realizing that the storage available was enough to increase the replication factor. In Household 2, 80% of the users files were replicated 4 times or more.

For comparison, the right bar in Figure 5.11 shows the replication state at the end of the trace when the greedy replication algorithm is being used, with a replication factor of two. (To make the results comparable we replayed the trace from the later deployment against an implementation of PodBase with the greedy algorithms.) Household 1 chose not to provide their trace for privacy reasons, and thus we can not include the results.

In households 6, 8 and 10, which have only two devices, the behavior of the two algorithms is almost identical. In households 2–5 and 9, the adaptive replication algorithm performed better, showing the advantages of dynamically calculating the replication factor in the adaptive algorithms. Additionally, household 9 benefited from a multi-step replication plan used by the adaptive algorithm, which was able to make effective use of a device to transport data.

5.5.2 Availability results

A secondary goal of PodBase is to place replicas of files on devices where they are likely to be useful. In our deployment, a plug-in specified that mp3 files should be preferentially copied to devices that are capable of playing music.

In analyzing the trace, we found that three of the households had already manually replicated all of their music files on the relevant devices. Thus, PodBase did not have much opportunity to improve availability in these households. However, it did provide a significant gain in availability for several of our households. Household 3 had its entire music library of 415 music files made available on all three of its computing devices. Household 7 had 851 music made available by PodBase and Household 8 had all 1318 music files made available. Household 9 had 1500 music files from a music library that was otherwise loosely synchronized between their devices, made available on two additional devices. An additional two households originally had a significant number of mp3 files on their laptops but not on their desktops. PodBase replicated these files onto the desktops, and the mp3 plug-in described in section 4.2.4 had externalized the music files. This happened during Used study 1 (i.e., before our trace collection started), therefore it did not show up in our trace. The users gained access to 426 songs and 2611 songs, respectively, on their desktop computers that were previously stored only on their notebooks. There was no increase in availability for one household that had no music files.

As described in Section 4.3.2, the replication algorithm first optimizes for dura-

bility, then cost (time to complete), and finally availability. A concern might be that by doing this, we are limiting the amount of availability the system can provide. We looked at the impact of this optimization process on the one household (9) for which the final replication plan did not result in full replication of files for availability. In this household the final replication plan results in 95% of the optimal availability. The remaining 5% were not achieved because the replication had not yet finished at the end of the trace collection, and not because of a limitation in the algorithm.

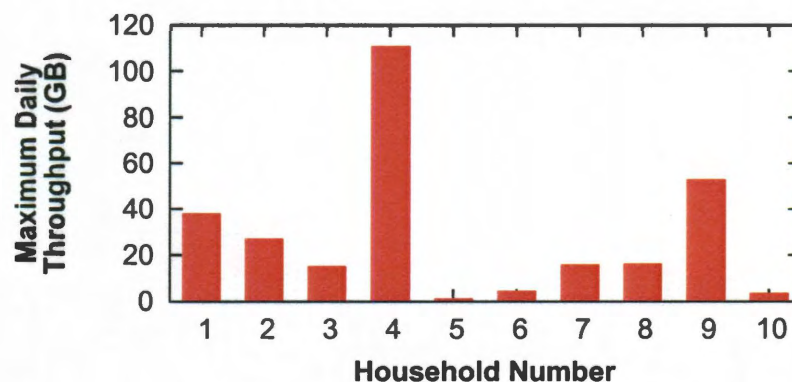


Figure 5.12 : Peak daily throughput for each household

5.5.3 Replication latency and throughput

We next look at the maximal replication throughput in each of the households. Since all households had many files to replicate at the beginning of the trace collection, the rate at which data was replicated early in the trace is a lower bound for the total replication throughput of a pool. This value in turn provides a lower bound for the rate of new or modified data that a household could generate, such that PodBase

would still be able to keep up with replicating.

Figure 5.12 shows that the peak throughput varies among the households in our deployment. The values range from 1.4 to 110 GB per day. This result shows that PodBase can keep up with a high to very high rate of data generation, using only the incidental connectivity that exists in our households.

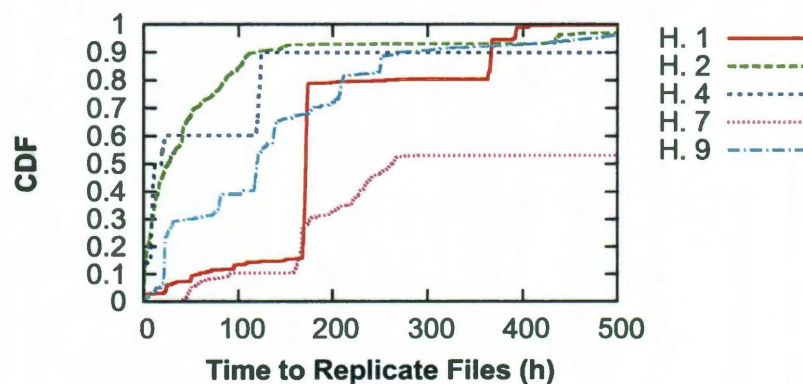


Figure 5.13 : Replication latency for households 1,2,4,7,9

We now examine the replication latency, the elapsed time until a new or modified file becomes replicated. If a file is not yet replicated, we include it in the CDF as having an infinite latency. We first examine those households with relatively short latencies. Figure 5.13 shows a CDF of how long it took to replicate a file. For households 2 and 4, over 50% of files were replicated within approximately one day. Households 1 and 7 took longer because there were extended periods with no connectivity. Household 9 replicated gradually over the course of the trace, as connectivity allowed.

Second, we show the latency of the households that took significantly longer to

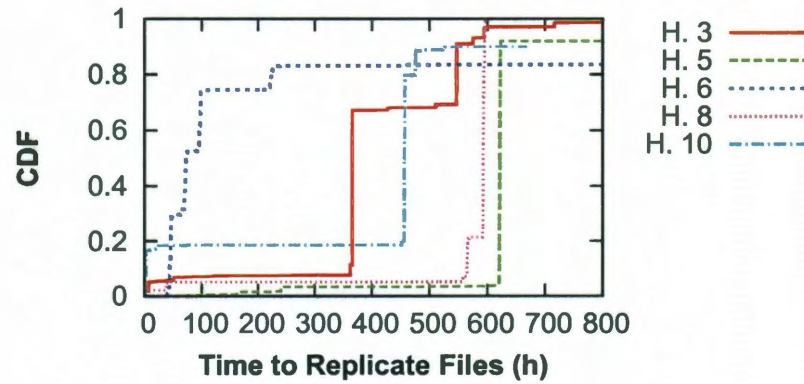


Figure 5.14 : Replication latency for households 3,5,6,8,10

replicate their files in Figure 5.14. In these households, device connectivity is the dominant factor in the replication latency. When there is connectivity, there are sharp jumps as files get replicated, followed by periods of disconnection, where no replication happens.

As with the previous latency results, these are conservative, because in most households, PodBase was busy replicating the user files found initially on the various devices during a large part of the trace collection. In steady state, PodBase would have to replicate only newly created or modified files, which would reduce the latencies considerably. Nevertheless, PodBase was able to replicate data in a timely fashion, subject to available storage and device connectivity.

5.5.4 File Workload

In this section we make more general observations about the workload in the user study.

Each of the households in the user study had on average 528,187 files taking up 332GB. After the initial crawl, on average of 21GB per day was generated by the addition of new and modifications of existing files. These numbers are skewed by one particular household that stored the disk image of an active virtual machine in the file system; without this household, the value was 381MB per day.

Our normal households generate new or modified data at a minimal/average/maximal rate of 4.5/36.1/316 Kb/s, while the “heavy” household generates 2.3 Mb/s. At an assumed upload bandwidth of 1 Mb/s, transferring the initial data while keeping up with updates would require between 3.7 and 121.6 (median 31.82) days for the normal households. (It would be infeasible for the heavy household, because the rate of new data exceeds the network bandwidth).

These results show that for timely replication of data, PodBase’s use of high speed connections and storage devices within a household is important. For the normal households, a broadband connection would suffice to replicate new data, but the heavy household would require a faster Internet connection. Relying solely on a broadband connection (e.g. to a cloud provider) for replication would require a long period of full network utilization, and increase the latency for files to be replicated.

As in the previous study, we also looked at file modifications. (This data is based on nine of our household, which allowed us to inspect their un-anonymized traces.) In this case, on average, 14,947 ($< 1\%$) of files were modified over the course of the trace and there were 104,635 file modifications across all of the nine households.

We analyzed these modifications and found only 64 (0.0006%) of the modifications concurrently updated the same file. Interestingly, as with the previous study, write conflicts rarely occur among personal devices in our deployment.

5.6 Summary

In this evaluation, we have evaluated PodBase both through controlled experimentation and through two user studies. We have shown that in the home environment PodBase works as designed, and provides automatic availability and durability to home users.

Part II

SLStore

In this part, we describe and evaluate SLStore. We first present the design of the entire system, including the storage ease abstraction, an adaptive planning component, and the sub-system that performs backup. We then present an evaluation, which measures the performance, overhead, and cost of deploying SLStore.

Chapter 6

System Description

In this chapter we describe SLStore. We begin by giving a high level view of the system components, and then describe each in detail.

6.1 System Overview

In this section, we give an overview of the major components of SLStore, their function, and how they interact with each other at a high level. A SLStore deployment is composed of a network of nodes, each of which hosts a subset of the components below. Figure 6.1 shows a sample deployment scenario.

Storage Leases: The low-level building block of SLStore is a novel storage abstraction called a *Storage Lease (SL)*. Data stored under a storage lease cannot be deleted or modified for a pre-determined period of time. The role of a storage lease in SLStore is similar to that of an off-line storage medium (e.g., tape storage) in enterprise storage solutions. Data stored on an off-line medium cannot be deleted or modified if the medium is not physically mounted. Because such media are only mounted infrequently (e.g. as part of a tape rotation), an error, fault, or compromise would have to go undetected for a long time in order for it to be able to corrupt all copies of important data. Storage leases have the same property, but without requiring me-

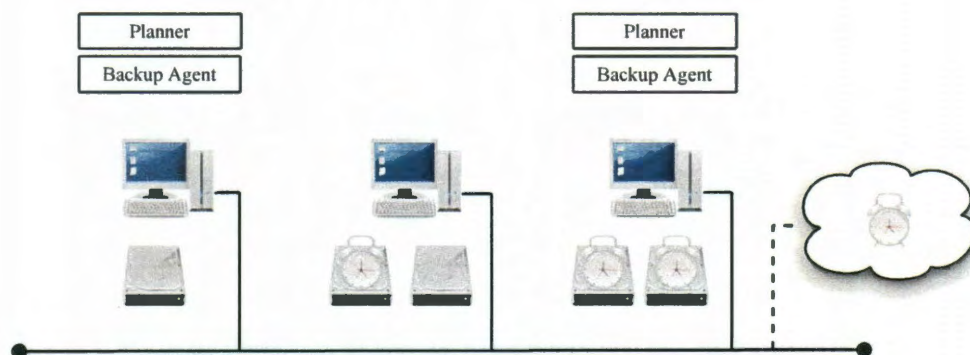


Figure 6.1 : A diagram showing 3 machines, each running one or more system component, connected via a network. A disk with a clock indicates a storage lease device, a disk without a clock indicates a conventional disk. Also shown is a connection to a remote cloud storage provider that supports storage leases.

chanical action to mount or unmount the storage medium. In Section 6.2 we describe the storage lease abstraction in detail, and show how it can be implemented.

Planner: The planner runs on each participating node. It takes as input the system-wide history of data addition and deletion, as well as the information on the available storage on the various storage devices (and possibly information about other services such as cloud storage). The planner obtains this information from the storage allocators. Based on this information, it solves a linear optimization problem to compute a backup plan that optimizes for maximal protection of data, given the available resources. The planner is described in more detail in Section 6.4.

Backup agent The backup agent is the system component that actually performs data backup. The backup agent takes two inputs. First, it takes information from the planner about the backup plan currently in effect. The backup plans specifies how often files should be backed up (incrementally or as part of a snapshot), on which

devices copies should be stored, and the period of the associated leases. Second, the backup planner determines the list of files that should be backed up, for instance, by scanning the filesystem for modified files. Based on this information, the backup agent stores data (and any associated metadata) on the appropriate devices. A backup agent runs on all participating nodes. The backup agent is described in more detail in Section 6.5.

6.2 Storage leases

In this section we present a novel storage abstraction, called a Storage Lease (SL). Storage leases can be used as a building block to protect data from operator error, security breaches and most software errors. We define the abstraction, then explain its requirements and properties. Finally, we show how to implement storage leases, either within the firmware of a storage device or as part of a higher-level storage service, such as cloud storage.

6.2.1 Motivation

In this section we provide background on current data protection techniques in both enterprises and in the home and small office environment. We motivate the introduction of storage leases by examining problems that are not currently addressed in the small business and home environment.

Data protection techniques

Creating redundant copies of data is the key to making it durable, but different levels of guarantees can be achieved depending on the way these copies are created and maintained. We break these down into the following four levels. First, *replicated file and storage systems* such as RAID [75] protect against storage device failure. Second, *on-line snapshots* protect against user error and application software faults. Third, creating *off-line or write-once copies* of data protects against storage system software and hardware errors, as well as security compromises that can affect on-line data. Lastly, creating additional *off-site copies* of the data protects against catastrophic failures that wipe out an entire site. We analyze how these different measures are usually applied in different environments.

Enterprise Environment

In an enterprise, data protection is a key IT responsibility. To minimize the chances of data loss and to implement legally mandated data retention policies, a combination of techniques are employed. Often enterprises use a tiered storage model where data is stored on backup servers, and then eventually moved to offline tape after a certain period of time. Servers often use some form of RAID configuration [75], and on-line snapshotting (e.g., NetApp WAFL [44], Sun's ZFS [125], or Windows VSS [115]). Off-line copies of the snapshot data are then created by copying the data onto tapes, and archived manually or using a tape robot. Lastly, tapes or disks are transported

periodically to a safe location to create off-site copies.

Even though storage leases are not designed primarily for enterprise environments, they may help improve enterprise backup practices by (i) write-protecting data until it has moved to offline storage, and (ii) lowering the cost of the solutions that are in place today by replacing expensive equipment like tape robots with the use of storage lease-enabled disks.

Home and Small Business Environment

Homes and small businesses have many of the same problems as enterprise users in that they have important financial or personal data that they wish to be retained. However, they can not afford the same professionally managed infrastructure as in the enterprise. Thus they are more vulnerable to viruses, security attacks, or even user error which could result in data loss.

Currently, diligent users install backup software (e.g., Apple's TimeMachine [110] or Microsoft's Backup and Restore [119]) but the fact that these backups are not offline implies that data is vulnerable to security compromises, like a virus or worm infection that destroys on-line data, or even to software bugs that silently erase backup data [36].

Some users may create off-line copies by periodically copying snapshots to write-once storage (e.g., DVDs). But, because this process cannot be automated, it is burdensome and often not performed on a regular basis (unless professional IT staff

is managing it, which is not practical).

Finally some users may use cloud backup as their means of data safety. However, cloud providers give no guarantees about the persistence of data, and the users have no knowledge about how the data is stored by the provider. If the provider make an error, the data stored could be lost [55]. Furthermore, data stored on the cloud can be overwritten, either by a malicious software or accidentally by the user. Also a limited upload bandwidth to the cloud can lead to a high data transfer time and therefore increase the window of vulnerability during which data is not backed up.

6.2.2 The storage lease abstraction

In this section, we present the Storage Lease (SL) abstraction. Storage leases can be used as a building block to protect data from operator error, security breaches and most software errors. We begin this section by defining the abstraction and its properties.

The main goal of storage leases is to protect stored data, for a pre-determined period of time, from accidental or malicious modification. Once written under a storage lease, a unit of data cannot be modified or deleted until the lease expires. During the lifetime of a SL, data stored under the SL is protected from modification or deletion due to human error, worms, viruses or other security breaches, and software bugs above the storage layer [36].

Storage leases provide the following guarantee. When a unit of data b is written

to an SL-enabled device D with an expiration time of t , the device ensures that:

At any time smaller than t , the device rejects write requests for data block b .

Furthermore, once a storage lease is written, the device can issue a certificate c that attests to the storage commitment.

This guarantee holds despite arbitrary faults outside of the system component that implements the storage lease. For instance, one of the solutions we will propose is implementing storage leases in the firmware of a disk drive. In this case, this guarantee holds despite any faults outside the device hardware or firmware (e.g., even if subsequently the host to which the device is attached becomes infected by malware).

Note that writing a block with a storage lease of infinity has the same effect as writing a block to write-once storage. Thus, write-once storage can be viewed as a special case of a storage lease.

Benefits of a write protection spectrum

Storage committed under a storage lease cannot be reclaimed during the lease period, but can be reclaimed anytime after the lease has expired. Thus, SL-enabled storage covers a spectrum from conventional read-write storage (which can be reclaimed at any time but does not protect data integrity) to write-once storage (which protects data but can never be reclaimed).

This additional degree of freedom enables a continuous trade-off between data

protection and storage commitment, which is important in environments that lack professional system administration: on the one hand, we need write protection, because taking storage off-line by manual or mechanical means is not practical. On the other hand, we cannot backup all data in write-once storage, because many systems produce large amounts of scratch data that would be expensive and unnecessary to store permanently. Storage leases provide a degree of freedom that can be used to resolve this tension.

Benefits of generating certificates

The certificate that is generated when writing a storage lease provides unforgeable confirmation that the data was written with a certain lease period to a given SL-enabled device.

Certificates are useful when SL-enabled devices are attached to machines that may be compromised. For example, consider the following scenario. A user has two machines in a home or corporate network, m_1 and m_2 , where m_2 is infected by a virus but m_1 is not. When m_1 tries to use m_2 's SL-enabled device to back up data, the malicious software running on the machine might act as if it had stored the data but in practice it discards it, leading m_1 to wrongfully believe the data had been backed up. To prevent this attack, the backup agent can request a certificate from the SL-enabled device. This certificate provides a signed confirmation of the write operation.

Furthermore, as we will exemplify in Section 6.3.5, certificates can also enable other uses of storage leases, namely to provide proof of storage commitment in environments where multiple parties from mutually distrusting administrative domains need to interact, such as peer-to-peer storage systems or cloud storage.

Managing storage space-time

Storage leases prevent the corruption of stored data, even when a computer's operating system is compromised by an attacker. However, leases also introduce a new attack vector: an adversary (e.g., a virus that has infected a machine) can commit the resources of an attached SL-enabled device by storing large quantities of useless data with long-term leases. The effect of such an attack is less serious than data corruption, as it is limited to preventing future use of the storage (denial-of-service). Nevertheless, it is a threat that must be addressed.

To understand how we can mitigate this attack, we must precisely characterize the resource that is being managed by an SL-enabled device. The resource commitment associated with a storage lease is proportional to both the amount of data and the period of the lease. This commitment is captured by the *storage space-time* $s_i = n_i * t_i$, where n_i is the amount of data covered and t_i is the period of lease i . The total space-time S committed in a device is $S = \sum_{i \in L} s_i$, where L is the set of existing leases (whether expired or not) in the device.

To mitigate *space-time filling attacks*, an SL-enabled device limits the rate at which

space-time can be consumed by new leases, i.e., $dS/dt \leq R$. The device enforces that R can only be set once by the administrator installing the device; it should be chosen to limit the space-time an attacker can consume within a given period. To illustrate the usefulness of this rate limiting, assume that a virus infection or a security breach is detected after no more than T_{detect} and that a device with capacity C has an expected lifetime of L , i.e., after L units of time the device will be filled with storage leases when used at its maximum rate. Then, we can ensure that an attacker can at most consume 1% of the device's total space-time by setting $R = 0.01 \frac{C \cdot L}{T_{detect}}$. Note that the attacker can choose to consume more data for a shorter time, or vice-versa, but the damage is limited either way.

Choosing R involves a trade-off between limiting the damage an attacker can cause and admitting legitimate leases at a sufficient rate to cover peak demand. In practice, however, satisfying the latter concern is not as critical as it may seem. First, there is often high demand for space-time during the installation of a new storage device, e.g., when an operating system or a database is installed. We can make sure this initial demand is not subject to the rate limit by setting R *after* the initialization. Second, legitimate users of leases (e.g., file systems or backup applications) can work around the rate limit should they exceed it during peak demand; they can simply request a lease with a shorter period than desired, and extend the lease prior to its expiration. An attacker, on the other hand, only has limited time until detection, and can at most commit $R \cdot T_{detect}$ of space-time.

Nevertheless, if an attack is not detected for a long time, a large amount of space-time may be wasted. To enable recovery from such an event, SL-enabled devices may provide a hardware switch (DIP switch or jumper) on the circuit board of the device, which can be used to reset all leases and the rate R . The switch can be operated by an expert as part of the system recovery. Note that the guarantees of storage leases are not affected by the presence of such a switch, because it cannot be activated by any software action.

Storage lease interface

The storage lease interface is similar to that of a block device. Defining the storage lease interface at the block level allows an implementation of storage leases in the firmware of a storage device (e.g., a disk drive). Such an implementation depends only on the integrity of the disk firmware and hardware, and is robust to bugs and compromises of all higher-level software.

The interface, as shown in Table 6.1, contains three key elements that depart from a traditional block device interface.

Lease period: The block write operation takes the expiration time t for the lease of the newly written block as an argument. A write fails if the target block is currently protected by a lease, or granting the lease would exceed the rate limit R . There is also an update operation used to extend the lease of an existing block, and an attest operation used to obtain the lease period of an existing block.

Certificates: The storage lease interface allows clients to obtain a signed lease certificate after data has been written with a given retention period. Note that the certificate must cover the expiration time (and not the lease duration) to prevent replay attacks.

Batching: To amortize the overhead of signature generation for certificates, a sequence of SL operations can be batched together and a single certificate issued for the batch. Reducing the number of signatures to one per batch instead of one per operation greatly improves the performance of, for instance, large file writes consisting of many block writes. Also, we must give upper layers control over how operations are delimited within a batch, so that the certificate can identify objects such as files or directories rather than blocks. Therefore, at any time within a batch, it is possible to obtain a hash of a series of data blocks operated on as part of the batch. These hash values are what is attested to when a certificate is subsequently generated.

Therefore, certificates have the following format:

$[Hash(o_1 + o_2 + \dots + o_n), D]_{\sigma_D}$ where o_1, \dots, o_n is a sequence of operations in a batch, as delimited by the application, $+$ indicates concatenation, and D the SL device. Moreover, o_i is the concatenation of the operation identifier, arguments and results of the i th operation in the batch.

6.3 Implementing storage leases

Next we focus our attention on how to implement the storage lease abstraction. We discuss three possible implementations: in the firmware of a disk device, in the OS driver of a storage device, and as part of a cloud storage service.

6.3.1 Firmware implementation

One implementation of storage leases adds the required functionality to the firmware of a disk storage device. Implementing storage leases at this level has a significant advantage: the trusted computing base required to enforce the properties of storage leases is limited to the firmware and the device hardware. Software faults and compromises affecting applications, file systems and operating systems do not affect the storage lease guarantees.

Implementing storage leases requires four tasks:

- Maintaining lease information for each data block and enforcing write protection.
- Keeping track of real time to decide when a lease has expired.
- Controlling the rate of space-time committed to leases.
- Generating signed lease certificates on demand.

We consider each of these tasks in turn.

Keeping lease information and enforcing protection

The firmware sets aside a set of data blocks on the storage device, which contain the lease expiration times for each of the primary data blocks (we call these lease blocks). Given that a 16-bit timestamp is sufficient for a lease granularity of one day, the overhead for storing an array of timestamps, one for each 4KB data block, is less than 0.05%.

Before writing to any block on the device, the respective lease block must be checked, and the write fails if there is an unexpired lease protecting that block. To speed up this step, a cache of lease blocks is maintained in the on-disk DRAM cache. Modified lease blocks are flushed to disk before the certificate for the respective batch of writes is returned (only then can the application be sure that the storage lease was successfully created).

The lease blocks are grouped on the physical disk into extents of at least the same size as the cache line of the on-disk cache. Therefore, once the disk head is positioned for a desired lease block, an entire extent can be read or written efficiently to/from the cache, taking advantage of likely spatial locality. The lease block extents are placed equidistantly on the disk, such that an extent contains leases for the data blocks closest to the extent. This choice minimizes, subject to the choice of lease block extent size, the seek distance between a data block and the lease block containing the associated lease. For instance, if the block size is 4KB (containing 2048 lease values) and a cache line has 64 blocks, then a lease block extent of 64 blocks follows every

extent of $64 * 2048 = 128K$ data blocks.

When lease blocks become corrupt, the disk can no longer validate whether a write to the corresponding data block is allowed. This case can be handled in different way, namely: (i) replicating lease blocks, which would double the overhead, (ii) optimistically allow the write, risking losing data, or (iii) upon detection, set all corresponding leases to the lifetime of the disk. Because a data block corruption is a relatively rare event, we decided for option (iii) because it is always safe.

Keeping real time

Keeping track of real time to determine when leases expire presents a design challenge. We cannot depend on the real-time clock of the attached computer, because that clock could be compromised by an attacker. Alternatively, adding a hardware real-time clock with the required battery to the circuit board of a storage device would add significant component and maintenance cost.

Fortunately, two observations come to the rescue. First, it is sufficient for the device to maintain a time that is earlier or at most equal to the actual real time. Having an earlier time means that leases are enforced longer than needed, which is conservative. Second, the passage of time can be tracked at a coarse granularity, because leases are for long periods, e.g. months. Thus, a coarse precision on the order of one day is acceptable.

Given these observations, it is sufficient to have the operating system periodically

provide a signed time certificate generated by a trusted time server to the device. A fault or compromise affecting the OS can at worst delay the flow of valid, up-to-date time certificates, which would delay the expiration of leases. As mentioned, delaying the expiration is conservative and causes no harm to data (though it delays the reclamation of storage).

The requirements for the trusted time server are straightforward in terms of precision and query load. As mentioned, a precision as low as one day is sufficient. Serving a cached certificate even to hundreds of millions of devices once every hour is straightforward with current server technology. However, a successful attack to significantly advance the clock on the server could place vast amounts of data at risk. Therefore, the signing keys must be kept off-line and certificates issued based on a manually verified clock. Once a day, a new certificate can be generated at an off-line computer and manually transferred to the on-line time server, using a portable storage device like a USB stick. Such time servers could be operated, for instance, by the vendors of SL-enabled storage devices.

We can further strengthen this scheme by requiring that k certificates are presented by independent trusted sources.

Rate-controlling space-time

As described in Section 6.2.2, the rate at which space-time is allocated to new leases is limited to prevent a compromised system from filling the disk with long-term lease

commitments. The parameter R can be set once on a new disk (or after a manual reset) using the SetRate operation.

Whenever a new lease is requested or an existing lease extended, the firmware checks the sum of space-time commitments due to new or extended leases during the past 24 hours, and rejects the request if the rate exceeds R . Of course, the system can retry a rejected request with a shorter lease period. Note that a request may be denied because the firmware's notion of real-time is behind. In this case, providing the disk firmware with a recent time certificate resolves the problem.

Generating lease certificates

The device firmware must generate signatures for certificates, and verify the signatures of time certificates.

The crypto primitives must be strong enough to prevent attacks on the signing keys. Because the keys need only be secure for the lifetime of a storage device (i.e., on the order of 5 years), we do not foresee the need for updates of the crypto libraries or keys on a device. The necessary keys can be assigned, and their validity verified once the devices are deployed, by the device manufacturer; crypto algorithms can be upgraded with each new device generation.

Another possible concern is the performance penalty due to signing. However, batching reduces the frequency of signature generation to a point where it is not of concern.

Finally, the state associated with a batch does not require more than a few hundred bytes, and can be stored in the device controller's existing RAM.

Firmware updates

To prevent an attacker from bypassing the lease protection, firmware updates must be protected. Therefore, the disk firmware must accept only updates that are signed by the device vendor or another trusted authority. The crypto infrastructure already in place for verifying time certificates can be reused for this purpose.

6.3.2 Driver implementation

An implementation of storage leases in the firmware of a storage device requires support from the device vendor. We believe that such support is feasible, because the cost for implementing leases is moderate and vendors have an incentive to include value-added features. For instance, disk vendors have recently added support for data encryption to disk drives [1].

If SL-enabled storage drives are not available, an alternative implementation is to provide the functionality as part of the storage device driver. Such an implementation is straightforward; however, a driver implementation increases the computing base trusted to enforce storage leases to include the OS kernel. Thus, the leases would be vulnerable to bugs and compromises of the OS kernel. This problem can be mitigated by reducing the size of the TCB, either through the use of a micro-kernel or a virtual machine monitor.

6.3.3 Storage enclosure implementation

Storage leases can also be implemented as part of a storage enclosure, for instance, a RAID box. Low-end enclosures of this type are now inexpensive enough even for home environments. Again, the implementation is straightforward. The storage enclosure firmware becomes part of the computing base trusted to enforce storage leases. However, unlike an operating system, this firmware is embedded, does not run applications and has a narrow interface. Therefore, it is likely to be far less vulnerable to compromise than an OS.

6.3.4 Cloud storage implementation

Storage leases can be implemented as part of a cloud storage service as well. In this case, the cloud service implements and ensures the guarantees associated with storage leases. Here, the trusted computing base for storage leases comprises the cloud service implementation. While this computing base is substantial, it is professionally managed and can be designed to have a narrow interface that minimizes the attack surface. Moreover, the faults it may suffer are unlikely to be correlated with a fault or compromise in the customer's computing base. Thus, a combination of client-side SL-enabled storage devices and an SL-enabled cloud service provides strong protection from operator error, software faults, security compromises, and even catastrophic site failures.

6.3.5 Applications of storage leases

In this section, we sketch applications of storage leases beyond SLStore. A storage lease enabled device appears to the operating system as a normal block device with an augmented interface. As such, the device is fully backward compatible and an unmodified operating system can use it as if it was a normal device. To make use of storage leases, however, an operating system, file system or application has to be modified to use the extended interface, and it has to be prepared to deal with the semantics of blocks protected by a lease.

In the following, we sketch four applications. The first a lease-aware snapshotting file system, is representative of integration with the existing operating system and file system. The other two, peer-to-peer storage and cloud storage, highlight interesting uses of lease certificates.

Lease-aware file system

A tight integration of lease-based storage requires a lease-aware file system. For instance, a lease-aware snapshotting file system could use leases to protect the integrity of in-place, copy-on-write snapshots, obviating the need to copy data to a dedicated device as part of a backup.

Starting from a snapshotting file system like WAFL or ZFS, the required changes are relatively modest. First, the file system must be extended to request leases for all data and metadata blocks associated with a given snapshot. Second, the file system

allocator must be made aware that leased blocks cannot be reclaimed until the lease expires. Lastly, metadata stored under a lease must be append-only. The changes are very similar to those required for SSD-aware file systems, where data blocks cannot be modified and reclaimed freely and individually. The design of a lease-aware file system is beyond the scope of this paper.

Peer-to-peer storage

Lease certificates can be used as proof of resource usage in volunteer-based systems where users might not have the right incentives to contribute to the system. For instance, in a cooperative peer-to-peer backup system [21], peers might pretend to contribute their fair share of resources, while in reality they are using the system without storing data on behalf of others. With storage leases, peers can demand lease certificates to other peers that are supposed to store their data, and exclude those that refuse to present lease certificates.

Cloud storage

Lease certificates can also be used by cloud providers to prove to their customers that their data is being stored for a certain period, and also that certain replication levels are being met. This would add value to a cloud storage service, since it would increase transparency, and overcome one of the main concerns that deter potential customers of cloud services, namely that a lack of proper data management from the cloud provider might lead to data loss, as has happened in the past [55].

6.4 Planner

In this section we discuss the planner, which generates a backup plan which conforms to a pre-defined backup policy. We begin with a description of backup policies.

6.4.1 Backup Policy

The backup policy defines how often a snapshot of the system's data should be created, how long these snapshots should be retained, and how many copies of the snapshot data should be stored on and off site. An example backup policy may specify that daily snapshots are stored on site for one week, that weekly snapshots are stored with one copy on site and another copy off site for one month, and that yearly snapshots are stored indefinitely off site. Here, it is assumed that on site copies are stored on SL-enabled devices and that off site copies are stored on an SL-enabled cloud storage provider, each with a lease period equivalent to the retention period of a snapshot.

SafeStore defines a set of default policies, from which the planner automatically selects the best feasible policy, given the available resources and the prevailing workload. If the resources are not sufficient to sustain the weakest default policy, or if the current policy will lead to an exhaustion of storage within the foreseeable future, the users are prompted to add more storage to the system. Sophisticated users can also define their own policies, but given that SafeStore is designed for lay users, it is discouraged.

The set of default backup policies are shown in Tables 6.2 and 6.3. The de-

faults are chosen to provide reasonable policies that maximize data protection with a given amount of available resources. They are influenced by common best practices in enterprise data management, and in commercial backup programs like Apple's TimeMachine [110].

6.4.2 Planning Process

The planner is responsible for (i) choosing the best backup policy from the set of default policies, and (ii) computing a backup plan that satisfies this policy. We discuss each of these task in turn. The planner formulates each task as a linear optimization problem and uses a state-of-the-art solver to compute a solution.

Choosing a policy

To choose a policy, the planner considers the currently available storage and space-time resources on SL-enabled devices in the system, as well as the expected rate at which data is likely to be generated or modified in the future. To estimate the latter, the planner looks at the rate at which data has been backed up in the recent history.

Based on this information, the planner determines the best backup policy that is either sustainable indefinitely, or leads to an expected storage exhaustion sufficiently distant in the future. By default, we consider a predicted exhaustion of storage within 30 days as acceptable, because it leaves users sufficient time to purchase and install additional storage, or to manually choose a weaker backup policy. (Because SafeStore is designed for lay users, the latter is strongly discouraged.)

The planner repeats this optimization daily, in order to be able to adjust to changes in the workload and storage availability. If the available resources drop below a level where the minimal backup policy can be implemented, users receive a strong warning that data can no longer be adequately protected.

Computing a backup plan

Given a policy choice, the planner next computes an assignment of snapshots to SL-enabled devices. This assignment is guided by three concerns:

- A snapshot should reside in its entirety on one device whenever possible. This policy maximizes the chance that a snapshot is available, considering that devices may fail and partial snapshots are of limited use.
- Related snapshots of the same data should be placed on the same device to maximize the benefits of de-duplication, while maintaining the same high availability of snapshots in the presence of device failures.
- The available storage space and storage space-time of all devices should be utilized.

The resulting assignment is then communicated to the backup agents.

6.5 Backup Agent

Given a backup policy and backup plan provided by the planner, it is the task of the backup agent to actually back up user's data. In this section, we describe the operation of the backup agent. Note that a backup agent may run on any participating node, including nodes that do not contain an SL-enabled storage device.

We describe the steps involved in backing up and restoring data in the following subsections.

6.5.1 Snapshots

A snapshot is a collection of files that represent the state of a (part of a) file system at a certain point in time. Because a snapshot is normally created concurrently with other activity on a given machine, the snapshot may not be consistent with any instant in the evolving state of the file system. This is a property of many backup applications and not unique to our solution. The alternative is to take the system offline while a checkpoint occurs (not practical) or to use a file system that explicitly support copy-on-write and checkpoints (e.g., ZFS).

A snapshot is made in two steps. The first step is to push all new data to the SL-enabled devices and gather lease certificates for the data. The second step is to migrate pointers to any unchanged data to the new snapshot and to update the lease period for that data. The decision about how to store new data is determined by the planner, as discussed in the previous section.

In order to be able to recover from catastrophic failures, a root for the last known-good snapshot must be locatable on the SL-enabled device. An initial block is pre-allocated to serve as the root of a chain of snapshot pointers. When a new snapshot is added, a pointer to that snapshot on disk is written into this block, and then a new block for the next pointer is pre-allocated. This process repeats to form a chain. Each block in this chain is written to the SL device with an infinite lease period. When the last known-good snapshot must be found, this chain is followed until the point in time desired or the chain ends. If while the system was compromised, this data structure was written to, the chain is still valid up until the point of compromise, and the last known-good snapshot is the end of the chain. If an attacker tries to compromise this chain, it will be detected the next time a good node attempts to make a snapshot.

6.5.2 Determining which files have changed

It is assumed that between successive snapshots, the backup agent is notified of all files that are being created or modified. This list is used to identify files that should be included in the next checkpoint. The method by which it is determined which files are changed is not important. Many operating systems provide an API by which an application can be notified of files that have been changed in a particular tree of the filesystem. When possible, the backup agent will leverage such API's. Otherwise, a separate process can crawl the file system and look for changes since the last snapshot.

6.5.3 De-duplication

The backup agent performs de-duplication, so that files stored on the same machine with identical content are backed up only once. For this purpose, the backup agents cache the lease certificate of each file they have backed up. During a snapshot creation, a backup agent broadcasts a query to determine if the file has already been backed up by another agent. Other backup agents respond with the associated lease certificate of each requested file they have backed up. In this case, the querying agent can store the existing certificate in its checkpoint, instead of actually writing the file.

Queries and responses are batched (1000 files per batch) for efficiency. Combined with the limited number of nodes in home office and small business environments, the simple broadcast approach has acceptable performance and overhead. Note that due to the secure lease certificates, faulty or compromised backup agents can at worst cause redundant data to be backed up.

6.5.4 Restoring a snapshot

When a storage device fails or a user wishes to roll back the system state, then a restore from a checkpoint occurs. The first step is to recover the checkpoint data structure. Then we gather all blocks in the checkpoint, and write them to the destination files that are recorded in the checkpoint data structure. Once the blocks are recovered, the data contained can be verified by hashing the contents and matching them to the hash and the signature of the hash stored in the receipts. Once all blocks

have been written and verified, the restore is considered successful.

6.5.5 Snapshot verification and repair

Periodically all currently live checkpoints are verified to ensure that the data stored matches the recorded hashes. This process is done in the background. If an error is discovered then the system attempts to repair the problem by making additional copies of the data in question.

Function	Description and Exceptions
<code>BId startBatch()</code>	starts a new batch and returns its id; fails if too many open batches
<code>void write(BId id, Bnum b, Expiration t,void *buf)</code>	writes data block #b from buffer buf with lease expiration time t; fails if block #b is protected by a lease, or space-time rate limit exceeded
<code>void read(BId id, Bnum b, void *buf)</code>	reads block #b and stores it in buffer buf
<code>void update(BId id, Bnum b, Expiration t)</code>	extends the lease of block #b to expiration time t; fails if t is earlier than the block's current lease expiration, or space-time rate limit exceeded
<code>Expiration attest(BId id, Bnum b)</code>	returns the current lease expiration time for block #b, or zero if block is not covered by an active lease
<code>Hash hash(BId id)</code>	returns the accumulated data hash in the batch, then reinitializes the hash accumulator; read, write and attest operations update the hash accumulator with the hash of the block they act on
<code>Certificate getCertificate(BId id)</code>	returns a certificate confirming the set of operations in the batch; the certificate is a signed hash of the set of successful operations in the batch, including their arguments and results
<code>void endBatch(BId id)</code>	terminates a batch; subsequent invocations of any operation with this batch id fail
<code>void setTime(TimeCertificate T)</code>	Advances the device's clock to the time provided in the certificate T; fails if T's signature can't be verified; ignored if T's time is less than or equal to the device's current time
<code>void setRate(long rate)</code>	Sets the rate at which the device admits new leases by specifying the maximal rate of storage space-time committed per day; only the first invocation succeeds

Table 6.1 : Storage lease device interface. Additionally, the device supports normal read and write operations. However, write operations to a block with an active lease fail.

Policy	Guarantees	Notes
1 copy of snapshot + 1 offsite	Data that was stored when a snapshot S_t was created is resilient to site, hardware and software failure for $R(S_t)$	Requires additional site or budget for cloud storage
2 copies of snapshot	Data that was stored when a snapshot S_t was created is resilient to hardware, software failure for $R(S_t)$	Tolerant of the failure of any SL device storing checkpoint data in a short period of time
1 copy of snapshot	Data that was stored when a snapshot S_t was created is to software failure for $R(S_t)$	Tolerant only to failure of node storing original data

Table 6.2 : Replication policies in order from strongest to weakest. The planner chooses the strongest feasible policy from this table, and then chooses the best feasible snapshot schedule from Table 6.3.

Snapshot Type (S_t)	Retention Time ($R(S_t)$)	Notes
Yearly	Indefinitely	Minimum snapshot schedule
Monthly	1 Year	Ability to do monthly snapshot implies yearly snapshots
Weekly	1 Month	Ability to do weekly snapshot implies monthly snapshots
Daily	1 Week	Ability to do daily snapshot implies weekly snapshots
Hourly	1 Day	Ability to do hourly snapshot implies daily snapshots

Table 6.3 : Snapshot schedules, ordered from weakest to strongest.

Chapter 7

Evaluation

In this chapter we evaluate the performance of SLStore. We first evaluate the performance of a storage lease device, which SLStore is built upon. We then describe the implementation of our prototype system. We then conduct a series of controlled experiments where we verify the system functions as designed. We follow this by running a set of trace-based experiments, whose goal is to measure the cost and overhead of running SafeStore in its target deployment scenario.

7.1 Storage lease evaluation

In this section, we evaluate the costs of storage leases. There are two main sources of overhead: the storage costs for the lease values, and the performance costs of checking leases, updating leases, and producing lease certificates.

As mentioned in Section 6.3, the storage overhead of a 16-bit lease value for each 4KB block is less than 0.05%¹. Lease values must be checked during each block write, leases values must be updated after a batch of block writes with storage leases, and a lease certificate may need to be generated when a batch terminates. The cost for

¹This value is conservative, because it may be possible to include a lease value within the existing low-level framing information stored with each block when a disk is formatted.

these operations depends on the placement of lease values on the disk, the policy for caching lease values in the disk controller’s DRAM cache, and in the case of lease update and certificate generation, the batch size. Our evaluation seeks to quantify these overheads, explore the tradeoffs among different lease value placements, caching policies and batch sizes, as well as the performance impact on real applications.

We use an implementation of storage lease in the DiskSim [14] disk simulator to explore different lease value placements and caching policies under different workloads.

DiskSim evaluation

In this section, we present results from an implementation of storage leases in DiskSim.

Experimental setup

To implement storage leases in DiskSim, we made modifications to support a subset of the storage lease API (i.e., `startBatch`, `write`, `read`, `endBatch`), reserve a region of the disk for lease blocks, support caching of lease blocks, and implement lease certificates as described in Section 6.3.

We use a validated disk model included in the DiskSim distribution. This disk is a 15K RPM Seagate 15K.5 drive, with a capacity of 146GB and an on-disk cache size of 16MB. We did not have access to the exact details of the internal cache organization, so we configured both disk models to use segmented LRU cache replacement and a write-through cache policy for writes. We used this particular disk model (A SCSI disk manufactured in 2007), as it was distributed and validated by the providers of

Workload Type	Description	Amount of I/O	Think Time	Identifier
Sequential	All data read/written to sequential blocks	400MB	0	SR/SW
Local	All data read/written to blocks with N blocks	400MB	0	LR/LW
Random	Random blocks read/written disk	400MB	0	RR/RW
Bittorrent	Download of movie file	3087MB	4231.91	BT
Game	Playing a strategy game	659MB	3517.8	G
Install	Downloading and installing Firefox	647MB	221.123	IN
Movie	Viewing a movie	1000MB	7391.63	M
Web	Websurfing and mail reading	591MB	3770.6	W
Bulk Sequential write	Bulk background write using storage leases	1000MB	0	B

Table 7.1 : Workloads used in evaluation

ID	Placement
B	The lease values are stored in a single extent of 4KB blocks at the beginning of the disk.
S x	The lease values are stored in equidistant extents of x 4KB blocks, followed by an extent of $x * 2048$ associated data blocks.
I	The lease value is stored together with the associated block.

Table 7.2 : Placements of lease values

ID	Lease cache configuration
U	Lease blocks are cached alongside normal data blocks in a unified cache.
C x	$x\%$ of the cache is dedicated to storing lease blocks.
I	The lease value is cached along with a cached data block.

Table 7.3 : Lease cache policies

DiskSim.

Table 7.1 list the workloads used in the evaluation. We use the DiskSim synthgen module to generate three different synthetic workloads. The *random* workload generates a sequence of single-block writes to random blocks on the entire disk. The *local* workload generates single-block writes to random block numbers within a range of 4000 consecutive blocks from the current position. The *sequential* workload generates single-block writes to consecutive block numbers. In each workload, a total of 400MB of data is being written.

In addition, we gathered several I/O traces from an author’s MacBook Pro notebook with 135GB of storage and the HFS+ file system. The traces were gathered using the iosnoop script, which is a wrapper around the DTrace [29] utility, and post-processed into a format usable by DiskSim. During each of the trace collections, the described activity was performed, while normal system and user background tasks

were running (e.g. updating the mail database, or reading configuration files for Skype). The I/O from these tasks were recorded in the trace as well. Unless otherwise stated, we ignored the think times between consecutive disk requests when re-playing the traces, which amplifies the load imposed by the trace on the storage lease device and yields conservative results.

Lease lookup overhead

Prior to a write operation, the associated lease value must be checked to decide if the write can proceed. The cost of this check depends on several factors: the placement of lease values relative to the associated blocks on disk; the policy and size of the cache for lease values; and the workload. We begin by measuring the impact of the lease lookup for different workloads, and for different lease value placements and caching policies.

We experimented with the lease value placements shown in Table 7.2 and the lease cache configurations shown in Table 7.3. The “I” cache configuration makes sense only with the “I” placement; in this configuration, we generously assume that the 16-bit lease values can be stored on disk and in the cache without reducing the effective size of disk or cache. Note that the line size of the cache is 16 blocks (4KB each). The lease blocks are aligned such that a cache line is used in its entirety either for lease blocks or for data blocks. For placements with less than 16 blocks per lease extent, we reduce the cache line size accordingly to avoid the obvious loss of cache

prefetching efficiency.

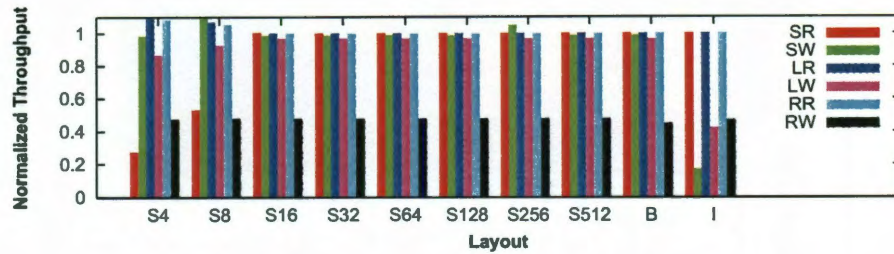


Figure 7.1 : Normalized I/O throughput (ops/sec) for synthetic workloads, with different lease placements shown on the x axis. The cache configuration is Unified (U), except for the inline (I) placement, where lease values are cached inline (I). The results are normalized to the throughput without leases.

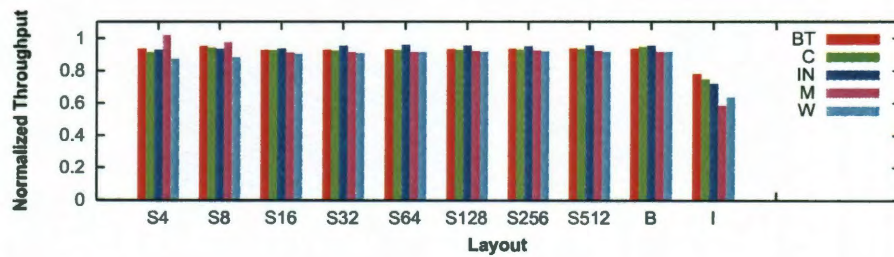


Figure 7.2 : Normalized I/O throughput (ops/sec) for trace workloads, with different lease placements shown on the x axis. The cache configuration is Unified (U), except for the inline (I) placement, where lease values are cached inline (I). The results are normalized to the throughput without leases.

Figures 7.1 and 7.2 show the normalized throughputs for the synthetic and trace-based workloads, respectively, with a unified cache (except as noted in the caption) and different lease value placements. We make the following observations:

- S16 appears to be the best overall placement, because it has good performance on the synthetic workloads (including the important sequential reads) and the trace-based workloads.

- Random writes (RW) are the worst-case workload for storage leases. The throughput of random writes is slightly less than half of that without leases, because caching is not effective in absorbing the disk access to read the lease value, which is required to validate the write. In practice, systems tend to avoid random writes to disks because of their inherently poor performance; therefore, we believe that the additional overhead should have limited impact on most systems.
- Ignoring the S4, S8 and I placements for the moment, the workloads are not strongly affected by the placement. In particular, the trace-based workloads all achieve between 8x and 100% of the normalized throughput. Note that the traces are replayed without think times; in a replay with think times, the increase in completion time resulting from storage leases would be negligible.
- The in-line placement (I) has very poor throughput for writes, because every block write requires an additional disk rotation between reading the lease value and writing the block. Caching is not effective, because the placement allows no pre-fetching of lease values. Inline placement also has the highest overhead on the trace-based workloads.
- The S4 and S8 placements achieves superior throughput for SW, LR and RR at the expense of a significant drop in sequential read (SR) performance. The reason is that the shorter cache line size benefits these workloads, while SR

benefits significantly from the increased pre-fetching that comes with a larger cache line size.

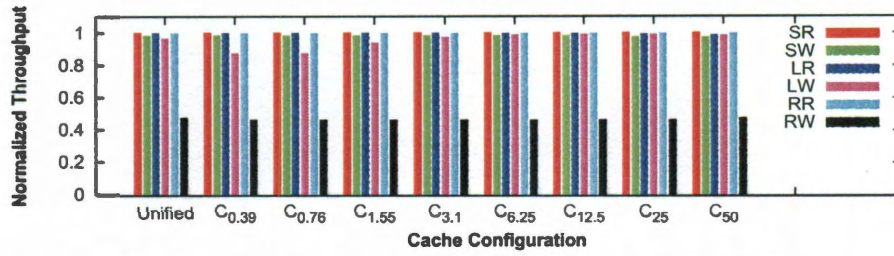


Figure 7.3 : Normalized I/O throughput (ops/sec) for synthetic workloads, with different cache configurations shown on the x axis. The total cache size is 16MB in all cases, and the lease value placement is S16. The results are normalized to the throughput without leases.

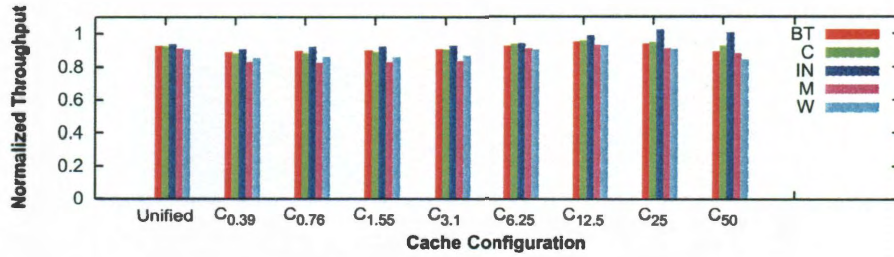


Figure 7.4 : Normalized I/O throughput (ops/sec) for trace workloads, with different cache configurations shown on the x axis. The total cache size is 16MB in all cases, and the lease value placement is S16. The results are normalized to the throughput without leases.

Next, we explore how different cache configurations affect the overhead of storage leases. Figures 7.3 and 7.4 show the normalized throughputs for the synthetic and trace-based workloads, respectively, with the S16 placement and different cache configurations.

The unified cache configuration achieves the best throughput across all synthetic workloads. While $C_{12.5}$, C_{25} and C_{50} achieve better throughput on some of the trace workloads (C and IN), they have slightly worse throughput on sequential writes, which is an important workload. We conclude the unified cache configuration is the best overall. This result is intuitive, as it provides the greatest flexibility in using the cache resources in the way most appropriate for a given workload.

We have performed additional experiments to explore many combinations of lease value placement and cache configuration. They confirm that S16 and a unified cache is a good overall configuration across the workloads we have studied.

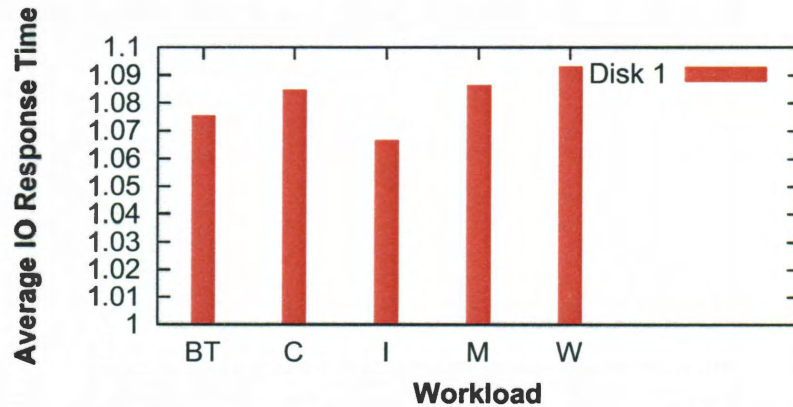


Figure 7.5 : Average per-operation I/O response time for the trace workloads. The traces were replayed with the recorded operation inter-arrival times (think times), and the per-operation response times measured. The placement is S16 and the cache configuration is unified. The results are normalized to the average per-operation response time without leases.

Figure 7.5 shows the average per-operation response time in our trace-based workload, with the S16 placement and a unified cache. Here, we played the operations in the traces while observing the recorded inter-arrival times, and measure the response

time for each operation. The increase in response time due to storage leases on the trace based workloads is between 3 and 23%.

Lease write overhead

Next we focus on workloads where data is written with a storage lease. In this case, there are additional overheads for writing lease values and for generating a lease certificate at the end of a batch of writes with storage leases.

In the experiment, we run the different trace workloads (replayed without think time) concurrently with a mock backup workload, which writes 1 GB of data sequentially in batches of increasing size. A lease certificate is requested for each batch. We assume that the disk controller has a crypto co-processor, which efficiently computes cryptographic data hashes and RSA signatures. Such crypto chips are inexpensive and widely available in devices like broadband wireless routers. In our simulation, we assume an inexpensive, low end coprocessor like the SafeNet SafeXcel-1741 [89] which can hash data at 325MB/s, and generate an RSA signature in 8.4ms. We added support to DiskSim to keep track of the crypto co-processor utilization and block if a new request is posted while it is still busy with a previous request.

In addition to the cryptographic overhead, the updated lease blocks must be written to disk when the end of a batch is requested. This requires flushing all modified data and lease blocks associated with that batch from the cache.

Figure 7.6 shows the throughput of the combined trace and mock backup workload

for different batch sizes, normalized to the throughput of the same workload on a disk without lease support. Decreasing the batch size increases the frequency at which the signing occurs and dirty lease blocks must be flushed to disk. The results show that decreasing the batch size below 10MB has a significant impact on performance. In order to get good performance for storage lease writes, the sizes of the batches should be maximized. In backup applications, this is easy to achieve. It should be noted that our results are conservative, because (i) we assume a low-end crypto chip (more powerful co-processors can generate an RSA signature in tens of microseconds), and (ii) we replayed the trace workloads without think times.

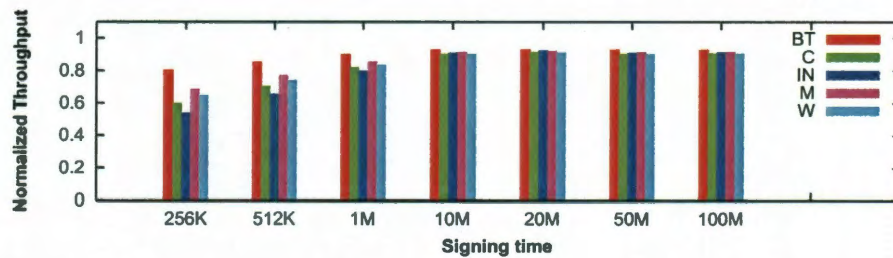


Figure 7.6 : Normalized throughput (I/O ops/sec) for combined trace and mock backup workload with lease, with different batch sizes shown on the x axis. Crypto processor with 325 MB/s hash throughput, RSA latency 8.4ms. Unified 16MB cache, placement S16.

7.1.1 SLStore evaluation

7.1.2 Implementation details

We have developed a prototype of SafeStore, which includes an implementation of a SL-enabled device. The storage lease device was implemented as a user level Java

wrapper for an ordinary disk device, because we did not have access to the firmware of a real disk. In the implementation, all blocks within a batch are written to a single file, using standard read and write system calls. For signing receipts we use RSA [86], and for computing hashes we use SHA1 [69]. All encryption and hashing implementations are from the BouncyCastle [12] Java library.

The rest of SafeStore is also implemented as a user level Java program. Our prototype implements all of the functionality described in the previous sections. SafeStore utilizes BerkeleyDB [6] as the storage back end for storing the checkpoint data structure, which includes a list of files, and the receipts that correspond to those files. In order to perform de-duplication, we query the other nodes in the system using application level multicast, which is implemented using the FreePastry library [74]. The SafeStore code base currently consists of 20,738 lines of code, not including libraries.

The planning module in SafeStore makes a series of calls to an outside linear programming solver when computing a backup strategy. In the current implementation we use the CPLEX solver, but this choice is not fundamental, and the solver could be replaced by an alternative solver.

7.1.3 Methodology and traces

In this section, we discuss the methodology and traces we use in this evaluation.

Methodology

In evaluating SafeStore, we have several goals, all of which require differing methodologies. First we wish to quantify the performance overhead of using a storage lease device. We do this in a controlled manner by using synthetically generated file workloads, and measure the time relative to simply copying the data. We next wish to verify the backup system functions correctly and measure its overhead with a realistic workload. In order to do this, we use our full evaluation to perform both backup and restore, on a running machine, and verify the correctness. In order to measure overhead in a wider variety of scenarios than just a single machine we use a trace driven evaluation. Finally, to get a longer term picture of the cost of running SafeStore, both in terms of storage and financial cost, we use a simulation and modeling based approach, based on the above mentioned traces. We next discuss the traces we will use in our evaluation.

Traces

As there are, to the best of our knowledge, no traces available for home office and small business storage workloads, we utilize two sets of traces that we gathered. The first was gathered from a previous project, and is from home users. This trace includes the storage workload from the households of 8 computer science researchers² over the course of approximately one month. It includes detailed information at

²In the final evaluation there will be one additional household included

file granularity, including name, size, modified time and content hash. The storage workload included in this trace includes both personal and work data. Given this mix, we feel that this trace is a reasonable first approximation of a home office environment. These traces cover approximately 30 days worth of use. The households in this trace had between 2 and 10 storage devices which contain data to be backed up.

The second trace is from the main backup up server at a computer science research institute. This trace includes more nodes (25 laptop nodes), but it includes much less detailed trace information. In this trace, it simply includes a backup id, size and amount of new files generated since the previous backup, as well as the size and amount of existing data stored. The data in this trace has does not provide individual file information so we can not test the effectiveness of de-duplication with this data. While this is just a single data point, we feel that this trace is somewhat representative of a small business setting. For each node, the trace includes 30 backups. For a user who is often connected this means it covers 30 days. For users who are intermittently connected, it may cover a much longer period of time.

In the rest of the evaluation we will refer to the first trace as the home office trace and the second as the small business trace.

7.1.4 Backup evaluation

In this section our goal is to show that the backup functionality in SafeStore works as it should. We first verify that it both successfully can backup and restore data, as

well as adapt to changing conditions. We then measure the overhead of the system by running the system for 30 days with both the small business and home office traces.

Verifying functionality

In order to verify the system functionality, we had the system make a backup of a users laptop. We then used the restore functionality to verify that both the backup functionality worked correctly and that we could successfully execute a restore.

Adaption to increased or decreased resources

We verified that the planner does indeed adjust its strategy as expected when resources are added or removed. It correctly adjusts as the rate at which files are added changes over time. If the rate increases past what the system can handle it will raise an alert.

Trace driven overhead evaluation

In order to show the SafeStore works in a more challenging environment than simple benchmarks, we drove the implementation using the previously described traces in order to measure the overhead, both storage and networking, of the system. Since our traces contain events for several devices, sometimes over different time periods we shift the first event for each device to be at the same time (and adjust all of the subsequent events for that device accordingly), and then replay the trace.

In order to simplify the experiment, we included only one resource node with

unlimited capacity (data is deleted from the disk if it is close to full). Since the traces last for 30 days, and we wish to test the system in a time efficient manner, we limited the system to take no hourly checkpoints, and we included a mechanism for fast forwarding idle periods when no trace events were occurring, and no backup was in progress. In all of the following graphs, results prefixed with H are the results from the home office trace, the results prefixed with S are from the small business trace.

Figures 7.7 and 7.8 shows the results for this experiment. In all cases the system ran successfully, and finished with all data backed up, and the correct number of snapshots. The figures show absolute and relative overhead respectively. The overhead is all data sent or stored that is not contained in the original files in the trace. Figure 7.7 shows that the network overhead varies from less than 10% to 22%. This overhead is predominately from sending checkpoint metadata to the backup nodes to be stored. The storage overhead is less, as it is required to only store non-expired checkpoint metadata. The overhead for the small business trace is the highest, as it contained the most files, and these were of small average size. Thus, the resulting checkpoint metadata is larger than for the home office cases. This overhead could be reduced by the use of incremental checkpoints, but at the cost of increased complexity.

In Figure 7.8, we examine the absolute overhead. The upper bar shows the amount of actual non-file data that was sent over the course of the experiment. The lower bar is all non-data that is stored on the storage lease device at the end of the trace. In the worst case household, the network overhead is 20GB during the month long

experiment period. During the same time the disk overhead is 8GB. For the small office trace, the network overhead for the month is slightly less than 100GB, and the storage overhead is approximately 40GB.

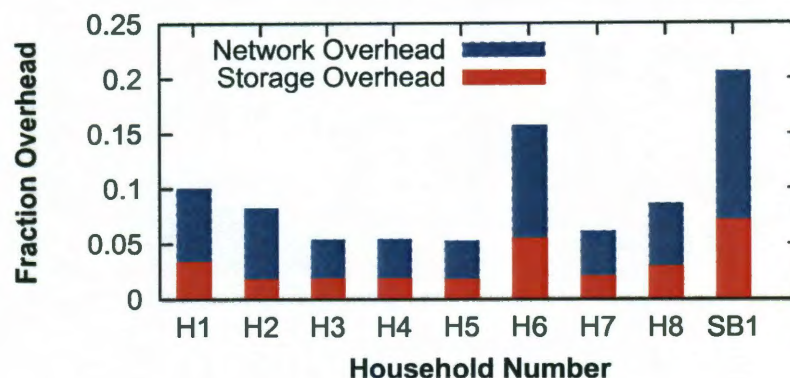


Figure 7.7 : Percentage overhead in a variety of home offices

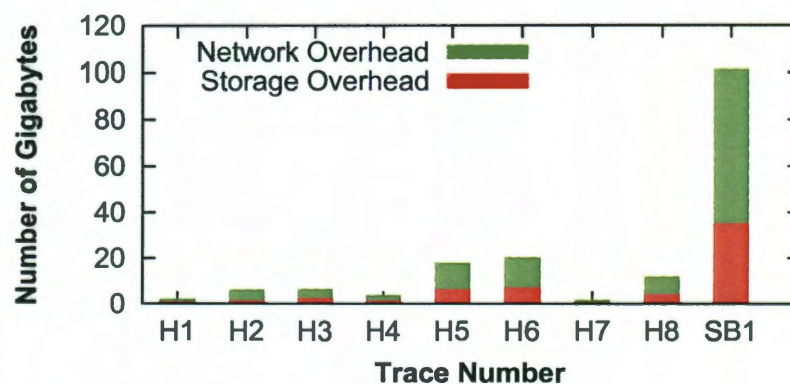


Figure 7.8 : Absolute Overhead in a variety of home offices.

One task the system performs is that of de-duplication. In order to quantify the effectiveness of the de-duplication in SafeStore we measured the amount of data transferred relative to the amount of data in the trace. In Figure 7.9, the result of

this experiment is shown. The different user groups reduced the amount of data sent by between 5% and almost 50%. Due to limitations of the small business trace, we were unable to quantify the effectiveness of de-duplication, and so just show the data sent. This result shows that de-duplication is effective. Note, that these households were relatively heterogeneous, incorporating a mixture of USB hard drives, media players, and computers with different operating systems. In a more homogeneous environment with only computing nodes, we would expect that the effectiveness of de-duplication will increase.

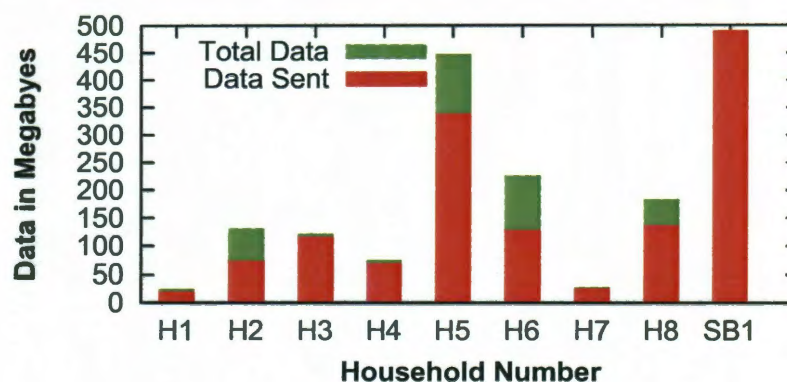


Figure 7.9 : Effectiveness of de-duplication

Another way to look at these results is to examine the amount of bandwidth required to support the entire backup application. For the most intense household, it would require 470GB of data (the amount of de-duplicated data plus overhead) to be transferred in one month. For the small business trace the total amount of data transferred is slightly less than 600GB of data. The bandwidth required to support this would be 0.19MB/s and 0.23MB/S respectively. This is, however, the

worst case scenario for the system, all machines are turned on simultaneously, and all begin backing up at the same time. If backup clients are added to the system more gradually, this bandwidth requirement would be spread over a longer period of time. All values above would double when backing up to 2 storage lease devices instead of one.

7.1.5 Simulated long-term results

In this section, we seek to look at the longer term costs of running SafeStore. Since none of our traces contain a full years worth of data, we take the traces we have and extrapolate them to a year by taking the average amount of data added, modified, and removed for the time period after the trace ends. We exclude the data that is already stored before the beginning of the trace. Instead of running full fidelity experiments for one year, which would have been prohibitively time consuming, we use the planning functionality built into SafeStore, to compute the amount of resource running the system would consume. Once we have computed the resource requirements, we can convert these into monetary terms by analyzing how much providing those resources would cost.

Please note that these results omit many details of the full simulation in order to compute costs. We currently do not take into account de-duplication, overhead, or the removal of data into account in these calculations. Refining the model is a topic of ongoing work.

Resource requirements.

In Figure 7.10 we show the resource requirements for each of the households we analyzed. The bar on the left is the amount of storage required to be free for the initial backup of all of the users data. The bar on the left, is the amount of new data that must be allocated by the storage lease device for each user. The initial requirements for all are on the order of a 100GB, and the rates required vary between 10MB and 100MB a day. For all of these users, their yearly storage requirements could be filled by a 1TB disk, which currently sells for around \$100. The policy followed in the experiment is to have daily, weekly, monthly and yearly checkpoints.

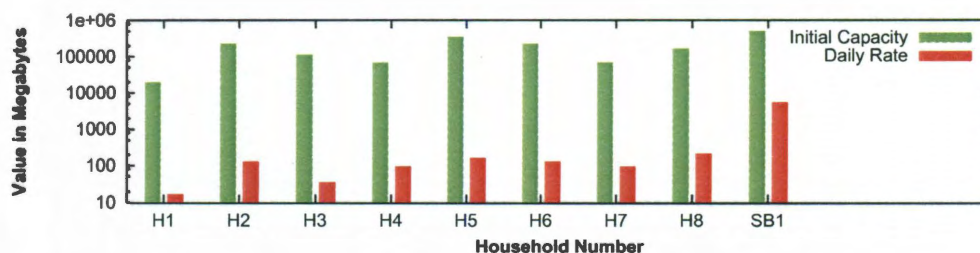


Figure 7.10 : Minimum settings required for initial capacity and rate, assuming no deduplication.

7.2 Summary

In this evaluation, we have used a variety of techniques to show that SafeStore works as described in the design. We have shown that it functions correctly, performs well, and is reasonably cost-effective.

Chapter 8

Conclusion

In this thesis, we have shown how to provide transparent storage management in environments where expert system administrators and dedicated hardware are not available. We described two new systems: The first, PodBase, transparently ensures availability and durability in the home, for mobile, personal devices that are mostly disconnected. The second, SLStore, provides enterprise-level data safety (e.g. protection from user error, software faults, or virus infection) while requiring minimal management.

We have built prototypes of both systems, and evaluated them. In the course of building and designing the system, we were required to develop several novel techniques, problem formulations, and abstractions. Experimental results show that both systems are feasible, perform well, require minimal user attention, and do not depend on expert administration during disaster-free operation.

Bibliography

- [1] Self-encrypting hard disk drives in the data center. Technical Report TP583, Seagate, Inc, November 2007.
- [2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Notices*, 33(11):81–91, 1998.
- [3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating System Review*, 36(SI):1–14, 2002.
- [4] BackupPC: Open Source Backup to disk. <http://backuppc.sourceforge.net/>.
- [5] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *Usenix Annual Technical Conference*, 2009.
- [6] Oracle Berkeley DB Java Edition. <http://www.oracle.com/database/berkeley-db/je/index.html>.

- [7] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *In Proceedings of NSDI'06*, 2006.
- [8] Marcos Bento and Nuno Preguia. Operational transformation based reconciliation in the FEW File System. In *Proceedings of the Eight International Workshop on Collaborative Editing Systems*, November 2006.
- [9] Kenneth P. Birman, Mark Hayden, Ozgur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.
- [10] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43, New York, NY, USA, 2000. ACM.
- [11] William J. Bolosky, John R. Douceur, and Jon Howell. The farsite project: a retrospective. *SIGOPS Operating System Review*, 41(2):17–26, 2007.
- [12] Bouncy Castle. <http://www.bouncycastle.org/>.
- [13] R. Bradshaw and C. Schroeder. Fifty years of ibm innovation with information storage on magnetic tape. *IBM Journal of Research and Development*, 47(4):373–383, 2003.

- [14] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and Contributors. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University Parallel Data Lab, May 2008.
- [15] M. Cao, T. Y. Ts'o, B. Pulvarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the 2005 Ottawa Linux Symposium*, 2005.
- [16] Carbonite. <http://www.carbonite.com/>.
- [17] J. D. Carothers, R. K. Brunner, J. L. Dawson, M. O. Halfhill, and R. E. Kubec. A new high density recording system: the IBM 1311 disk storage drive with interchangeable disk packs. In *AFIPS '63 (Fall): Proceedings of the November 12-14, 1963, fall joint computer conference*, pages 327–340, New York, NY, USA, 1963. ACM.
- [18] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.
- [19] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. *SIGARCH Computing. Architecture News*, 18(3a):322–331, 1990.
- [20] COIN-OR Linear Programming Solver. <http://projects.coin-or.org/Clp>.

- [21] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. *SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.
- [22] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 120–132, New York, NY, USA, 2003. ACM.
- [23] High-performance mathematical programming engine - IBM ILOG CPLEX - Software. <http://www-01.ibm.com/software/integration/optimization/cplex/>.
- [24] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87*, pages 1–12, New York, NY, USA, 1987. ACM.
- [25] M. Demmer, E. Brewer, K. Fall, S. Jain, M. Ho, and R. Patra. Implementing Delay Tolerant Networking. Technical Report IRB-TR-04-020, Intel Research, 2004.
- [26] John R. Douceur and Jon Howell. Distributed directory service in the Farsite file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 321–334, Berkeley, CA, USA, 2006. USENIX Association.

- [27] Drobo Product Description. <http://www.drobo.com/Products/drobo.html>.
- [28] Dropbox. <https://www.dropbox.com/>.
- [29] dtrace. <http://www.sun.com/bigadmin/content/dtrace/>.
- [30] Data Backup and Recovery Products. <http://www.emc.com/products/category/backup-recovery.htm>.
- [31] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proc. SIGCOMM '03*, Aug 2003.
- [32] Alan D. Fekete and Krithi Ramamritham. Replication. volume 5959 of *Lecture Notes in Computer Science*, chapter Consistency Models for Replicated Data, pages 1–17. Springer Berlin / Heidelberg, 2010.
- [33] Brian Ford. *UIA: A Global Connectivity Architecture for Mobile Personal Devices*. PhD thesis, Department of Electrical Engineering and Computer Science, Sep 2008.
- [34] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *Proc. OSDI '06*, Nov 2006.
- [35] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. User-relative names for globally connected personal de-

- vices. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, Feb 2006.
- [36] Ron Garret. A Time Machine time bomb. <http://rondam.blogspot.com/2009/09/time-machine-time-bomb.html>.
- [37] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.
- [38] Garth Alan Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, U. C. Berkeley, April 1991.
- [39] GNU Linear Programming Kit. <http://www.gnu.org/software/glpk/>.
- [40] Groove. <http://office.microsoft.com/groove>.
- [41] Olafur Guthmundsson, James Da Silva, James Da Silva, and Olafur Guomundsson. The amanda network backup manager. In *In Proceedings of USENIX Systems Administration (LISA VII) Conference*, pages 171–182, 1993.
- [42] Mark Hayakawa. WORM Storage on Magnetic Disks Using SnapLock Compliance and SnapLock Enterprise. Technical Report TR-3263, Network Appliance, 2007.
- [43] Technical Note TN1150: HFS Plus Volume Format. <http://developer.apple.com/mac/library/technotes/tn/tn1150.html>.

- [44] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.
- [45] John H. Howard. An Overview of the Andrew File System. Technical Report CMU-ITC-88-062, Information Technology Center, Carnegie Mellon University, 1988.
- [46] Data and Network Storage Products from HP StorageWorks. <http://h18006.www1.hp.com/storage/index.html>.
- [47] HP Upline. <http://www.upline.com>.
- [48] iDrive. <http://www.idrive.com>.
- [49] iTunes. <http://www.apple.com/itunes/>.
- [50] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. In *Proceedings of SIGCOMM '04*, pages 145–158, New York, NY, USA, 2004. ACM.
- [51] Alexandros Karypidis and Spyros Lalis. Omnistore: A system for ubiquitous personal storage management. In *Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, 2006.
- [52] Micheal Leon Kazar. Synchronization and Caching Issues in the Andrew File System. Technical Report CMU-ITC-88-063, Information Technology Center,

Carnegie Mellon University, 1988.

- [53] Kimberly Keeton, Terence Kelly, Arif Merchant, Cipriano Santos, Janet Wiener, Xiaoyun Zhu, and Dirk Beyer. Don't settle for less than the best: Use optimization to make decisions. In *Proc. HotOS '07*, May 2007.
- [54] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISks). *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 27(3):42–52, 1998.
- [55] Jason Kincaid. T-mobile sidekick disaster: Danger's servers crashed, and they don't have a backup. <http://techcrunch.com/2009/10/10/>.
- [56] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.
- [57] R. B. Lawrance, R. E. Wilkins, and R. A. Pendleton. Apparatus for magnetic storage on three-inch wide tapes. In *AIEE-IRE '56 (Eastern): Papers and discussions presented at the December 10-12, 1956, eastern joint computer conference: New developments in computers*, pages 84–90, New York, NY, USA, 1957. ACM.
- [58] Richard B. Lawrance. An advanced magnetic tape system for data processing. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 181–189, New York, NY, USA, 1959. ACM.

- [59] Live mesh. <http://www.mesh.com>.
- [60] Petros Maniatis, Mema Roussopoulos, T. J. Giuli, David S. H. Rosenthal, and Mary Baker. The lockss peer-to-peer digital preservation system. *ACM Transaction on Computer Systems*, 23(1):2–50, 2005.
- [61] Petros Maniatis, Mema Roussopoulos, TJ Giuli, David S. H. Rosenthal, Mary Baker, and Yanto Muliadi. Preserving Peer Replicas By Rate-Limited Sampled Voting. In *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.
- [62] Many PC Users don't backup valuable data. http://money.cnn.com/2006/06/07/technology/data_loss/index.htm.
- [63] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.
- [64] MobileMe. <http://www.apple.com/mobileme/>.
- [65] Mozy. <http://www.mozy.com/>.
- [66] Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 231–242, New York, NY, USA, 1994. ACM.

- [67] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.
- [68] NetApp Backup and Recovery Solutions. <http://www.netapp.com/us/solutions/infrastructure/backup-recovery/>.
- [69] NIST. Secure hash standard. May 1993.
- [70] Lev Novik, Irena Hudis, Douglas B. Terry, Sanjay Anand, Vivek Jhaveri, Ashish Shah, and Yunxin Wu. Peer-to-Peer Replication in WinFS. Technical Report MSR-TR-2006-78, Microsoft Research, 2006.
- [71] Overview of FAT, HPFS, and NTFS File Systems. <http://support.microsoft.com/kb/100108>.
- [72] Outlook. <http://www.microsoft.com/outlook/>.
- [73] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.
- [74] Pastry - A scalable, decentralized, self-organizing and fault tolerant substrate for peer-to-peer applications. <http://www.freepastry.org/>.

- [75] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM.
- [76] PC Pitstop Research. <http://pcpitstop.com/research/storagesurvey.asp>.
- [77] Daniel Peek and Jason Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proc. OSDI '06*, November 2006.
- [78] B. C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.
- [79] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus Distributed File Systems. In *Proc. WMRD*, pages 20–25, November 1990.
- [80] Nuno Preguia, Carlos Baquero, J. Legatheaux Martins, Marc Shapiro, Paulo, Srgio Almeida, Henrique Domingos, Victor Fonte, and Srgio Duarte. Few: File management for portable devices. In *Proceedings of The International Workshop on Software Support for Portable Storage*, March 2005.

- [81] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.
- [82] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.
- [83] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Conference Proceedings*, pages 183–195. USENIX, June 1994.
- [84] E. Riedel, C. Faloutsos, G.A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, jun. 2001.
- [85] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
- [86] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

- [87] Maya Rodrig and Anthony LaMarca. Oasis: an architecture for simplified data management and disconnected operation. *Personal and Ubiquitous Computing*, 9(2):108–121, 2005.
- [88] Simple Storage Service (S3). <http://aws.amazon.com/s3>.
- [89] SafeXcel-1741 Security Co-processor. <http://www.authentec.com/1741processor.cfm>.
- [90] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, March 2005.
- [91] Brandon Salmon, Frank Hady, and Jay Melican. Towards Efficient Semantic Object Storage for the Home. Technical Report CMU-PDL-07-107, Parallel Data Laboratory, Carnegie Mellon University, 2007.
- [92] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: Semantic Data Management for the Home. Technical Report CMU-PDL-08-105, Parallel Data Laboratory, Carnegie Mellon University, 2008.
- [93] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: semantic data management for the home. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 167–182, Berkeley, CA, USA, 2009. USENIX Association.

- [94] Brandon Salmon, Steven W. Schlosser, and Gregory R. Ganger. Towards Efficient Semantic Object Storage for the Home. Technical Report CMU-PDL-06-103, Parallel Data Laboratory, Carnegie Mellon University, 2006.
- [95] Brandon Salmon, Steven W. Schlosser, Lily B. Mummert, and Gregory R. Ganger. Putting home storage management into Perspective. Technical Report CMU-PDL-06-110, Parallel Data Laboratory, Carnegie Mellon University, 2006.
- [96] Russel Sandberg. The Sun Network File System: Design, Implementation and Experience. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.
- [97] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.
- [98] R. A. Skov. Pulse Time Displacement in High-Density Magnetic Tape. *IBM Journal*, April 1958.
- [99] Windows Live SkyDrive. <http://skydrive.live.com/>.
- [100] Sumeet Sobti, Nitin Garg, Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *FAST '02: Proceedings of the Conference on File*

and Storage Technologies, pages 159–174, Berkeley, CA, USA, 2002. USENIX Association.

- [101] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilei Shao, Chi Zhang, Elisha Ziskind, Arvind Krishnamurthy, and Randolph Y. Wang. Segank: a distributed mobile storage system. In *Proc. FAST '04*, March 2004.
- [102] SOS Online Backup. <http://www.sosonlinebackup.com>.
- [103] Jacob Strauss, Chris Lesniewski-Laas, Justin Mazzola Paluska, Bryan Ford, Robert Morris, and Frans Kaashoek. Device transparency: a new model for mobile storage. *SIGOPS Oper. Syst. Rev.*, 44(1):5–9, 2010.
- [104] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *OSDI*, pages 165–180, 2000.
- [105] SugarSync. <http://www.sugarsync.com/>.
- [106] StorageTek Enterprise Backup Software. <http://www.oracle.com/us/products/servers-storage/storage/storage-software/031597.htm>.
- [107] Edward Swierk, Emre Kcman, Nathan C. Williams, Takashi Fukushima, Hideki Yoshida, Vince Laviano, and Mary Baker. The Roma Personal Metadata Service. In *Proc. WMCSA 2000*.

- [108] Jon Tate, Fabiano Lucchese, and Richard Moore. *Introduction to Storage Area Networks*. IBM Redbooks, 2006.
- [109] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP'95*, December 1995.
- [110] Time Machine. <http://www.apple.com/macosx/features/timemachine.html>.
- [111] Dinh Nguyen Tran, Frank Chiang, and Jinyang Li. Friendstore: Cooperative Online Backup Using Trusted Nodes. In *SocialNet'08: First International Workshop on Social Network Systems*, Glasgow, Scotland, 2008.
- [112] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, 1996.
- [113] Unison File Synchronization. <http://www.cis.upenn.edu/~bcpierce/unison/>.
- [114] Michael Vrabie, Stefan Savage, and Geoffrey M. Voelker. Cumulus: filesystem backup to the cloud. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 225–238, Berkeley, CA, USA, 2009. USENIX Association.

- [115] What Is Volume Shadow Copy Service. [http://technet.microsoft.com/en-us/library/cc757854\(ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc757854(ws.10).aspx).
- [116] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, 1983.
- [117] Mike Wawrzoniak, Larry Peterson, and Timothy Roscoe. Sophia: an information plane for networked systems. In *Proc. HotNets-II*, 2003.
- [118] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Conductor: orchestrating the clouds. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pages 44–48, New York, NY, USA, 2010. ACM.
- [119] Windows Backup and Restore. <http://www.microsoft.com/windows/windows-7/features/backup-and-restore.aspx>.
- [120] Windows Home Server. <http://www.microsoft.com/windows/products/winfamily/windowshomeserver/default.mspx>.
- [121] Windows live sync. <http://sync.live.com/>.
- [122] WinFS Team Blog. <http://blogs.msdn.com/winfs/>.
- [123] Xdrive. <http://www.xdrive.com>.

- [124] Qin Yin, Justin Cappos, Andrew Baumann, and Timothy Roscoe. Dependable self-hosting distributed systems using constraints. In *Proc. HotDep '08*, Dec 2008.
- [125] Solaris zfs. <http://www.sun.com/software/solaris/zfs.jsp>.
- [126] Jinsuo Zhang, Abdelsalam (Sumi) Helal, and Joachim Hammer. UbiData: ubiquitous mobile file service. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, 2003.
- [127] Zmanda. <http://www.zmanda.com/>.
- [128] Zune Software. <http://www.zune.net/en-us/software/collection/default.htm>.