Device-Independent On Demand Synchronization in the Unico File System

by

Jonatan Schroeder

B.Sc. Computer Science, Universidade Federal do Paraná, 2002

M.Sc. Computer Science, Universidade Federal do Paraná, 2006

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August 2016

Abstract

As people increasingly use a wide variety of devices throughout their day, it becomes important to give users consistent and efficient access to their data in all their devices, while at the same time take into consideration the resource limitations in each device. This dissertation presents a storage platform that presents users with a single view of all their data, independently of the device they are using. Each of the user's devices has full and up-to-date information about the entire data structure and metadata, and is able to retrieve any file transparently as needed. File content, including history, is efficiently distributed among the user's own devices according to where it is used, with no need for centralized storage and no data stored in devices that are not owned by the user. Peers communicate with each other directly to maintain a consistent state. Users also have no need to know where the data is stored, or to remember in which device the latest update to a file has been performed. The user is able to adapt the platform to devices with different storage and network characteristics. While the latency to access content is affected, and some data may not be available at all times if some devices are disconnected, applications have access to tools that allow them to adapt to data availability.

Preface

A summary of some of the contributions in this dissertation, as well as early versions of the experimental results, were presented in *Device-Independent On Demand Synchronization in the Unico File System*, published in the 6^{th} International Conference and Workshop on Computing and Communication (2015) by Jonatan Schroeder and Michael J. Feeley. My contributions in the paper include the identification of the research problems, the design and implementation of the proposed system, the gathering and analysis or the experimental results and the preparation of the manuscript. Dr. Feeley has played a supervisory role throughout the process.

An early version of Unico, at the time named GitOD, was presented in *GitOD: An on demand distributed file system approach to Version Control*, published in the Proceedings of the 2012 International Conference on Collaboration Technologies and Systems (CTS) by Jonatan Schroeder. This paper described some of the same research problems as those used in this dissertation, but applied specifically to version control systems.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

AFS Andrew File System (see Section 2.1)

CFS Cedar File System (see Section 2.1)

CCN Content-Centric Network (overlay network that addresses and routes nodes based on content identification)

DHT Distributed Hash Table (key-value data structure that uses a P2P network to distribute the storage)

FUSE Filesystem in Userspace (architecture that allows the implementation of virtual file systems in user-level domain)

GRAND Git Revisions As Named Data (see Section 2.3)

HDFS Hadoop Distributed File System (see Section 2.1)

LRU Least Recently Used (data structure that keeps track of used objects, sorting them by the last time they were used)

NAT Network Address Translation (methodology used to translate TCP addresses, mainly when the number of available IP addresses is limited)

NFS Network File System (see Section 2.1)

NTFS New Technology File System (proprietary file system developed by Microsoft)

OLPC One Laptop Per Child

P2P peer-to-peer (network paradigm where nodes connect directly to each other instead of to a central server)

POSIX Portable Operating System Interface (family of standards used in systems derived from Unix and its variants)

RTT Round Trip Time (amount of time between when a network message is sent from a host and its response is received)

SHA-1 Secure Hash Algorithm 1 (cryptographic hash function)

SMB Server Message Block (network file system developed by Microsoft)

SSL Secure Socket Layer (network protocol that adds an encryption and authentication layer to a stream of data traffic)

SSH Secure Shell (network protocol that allows secure remote access to a device user terminal and files)

SSHFS Secure Shell File System (network file system that mounts a file system using the SSH protocol)

TCP Transport Control Protocol (transport-layer network protocol that implements connection-oriented reliable streaming of data)

VCS Version Control System (platform used to manage synchronization and history of collaborative repositories, see Section 2.3)

Acknowledgements

This dissertation is dedicated to God. Without His guidance and inspiration this dissertation would never have been completed.

This work has been completed thanks to the love and patience of my family. My wife, Bianca Neufeld Schroeder, has been my inspiration and support for so many years. My daughters, Melissa Grace Schroeder and Megan Elizabeth Schroeder, have given me the motivation to continue. My parents, Siegfried and Mirian Gortz Schroeder, have been a source of encouragement and support from the beginning.

I would like to thank my supervisors, Dr. Son Vuong and Dr. Michael Feeley, for their valuable input and guidance. Their orientation and patience, along with the input from Dr. Alan Wagner, allowed me to focus my research in a successful direction.

I am thankful for the input from colleagues in my research groups, NSS and NIC, for their suggestions. In particular I would like to thank Mohammed Alam for his friendship and for always being ready to discuss ideas and critique manuscripts. Additionally I appreciate the input from other colleagues like Lynsey Haynes, Jianing Yu, Nowrin Anwar, Jason Chou, Stanley Chiu, Wei Li, Anthony Yu, Albert Chen, Ricky Chen, Andrew Tjia, Carmen Liang, Nguyet Nguyen and others.

I thank the people in the Department of Computer Science for providing the structure that allowed me to complete this work. In particular I am obliged to mention the help and support received from Chen Greif, Donald Acton, Steven Wolfman, William Aiello, Raymond Ng, Hermie Lam and Joyce Poon.

Chapter 1

Introduction

1.1 Problem Description

The proliferation of new mobile computing platforms and associated device specialization has opened a new era in which people regularly use multiple devices throughout their day, depending on where they are and what they are doing. At the same time, as their lives become more digital, these users are placing increasing value on an expanding volume of data that they manipulate with these devices. Each device is a portal to this data; each with different capabilities adapted to particular working environments or styles. But, as tasks increasingly span multiple devices, the notion that each device has a separate data storage is rapidly outliving its usefulness.

Typical hard disk drives today reach the order of terabytes, even for personal computers. An average home network connection, however, provides under ten megabits per second connectivity for download, and even less for upload [4, 39]. This connectivity would require months of continuous transmission to transfer all the contents of a full hard drive. Users must instead manually choose a subset of their data that is synchronized to all of their devices, leaving the vast majority of their data stuck in the past, attached to the device that stores it, accessible only explicitly through that device.

Furthermore, full replication is not well suited to the device heterogeneity common in modern personal computing. Some devices produce data, while other de-

vices consume it. Some applications and file formats are device dependent: some devices will access the raw original file, other devices will only read the generated final result. Some devices have abundant storage, while other devices can only store a small subset of the user's data.

This dissertation addresses the problem of providing users access to all of their data using any of their devices, while keeping some objectives:

- giving the user access to the latest version of any file in all of their devices;
- efficiently using the user's own available resources, particularly storage and network;
- allowing a user to have control of their own data and the policies that dictate where it is stored, while providing a reasonable default policy for users that do not require such customization;
- limiting the conflict rate and access latency.

1.2 Existing Approaches

Local file systems such as NTFS or ext4 on these devices stop at the device's edge [12, 40]. They organize local storage, allowing it to be searched and providing local access to data. But they provide no support for finding data on remote nodes, moving data among nodes or managing the consistency of replication data. These tasks, which become increasingly important as device types proliferate, are left to application-level solutions or are performed manually by users.

Cloud storage systems such as Dropbox, Google Drive and Apple iCloud have stepped into this breach to provide device-independent storage that is replicated to "the cloud" and synchronized across a user's devices [16]. But the effectiveness of this approach is limited by the cost of cloud storage. In order to support data synchronization using this model, all the user's files need to be stored in a superfluous server structure, as well as in all the devices where these files are used. Not only does this imply inefficient use of the user's own available storage resources, it also means a potentially wasteful use of bandwidth to keep the data up-to-date. As a result, it is likely to become increasingly expensive for such a platform to contain a

user's entire data corpus: their family photos, home videos, important documents, music, personal communication, etc.

Another concern raised about commercial cloud systems is that the user has no control over where the data is stored. Since these platforms may distribute the data across servers owned by different companies, or even in different countries, the privacy of a user's data is at the mercy of policies and laws beyond the user's control [48].

An alternative approach followed by systems such as OriFS and BitTorrent Sync uses peer-to-peer synchronization, rather than the cloud, to replicate file data on multiple devices [5, 19, 39]. Here again, however, if one wants every device to *see* a single, unified file system, the file system must be fully replicated on those devices and every file change must be propagated, eventually to all of the devices. And so, for both cloud- and peer-to-peer systems, the overheads are typically just too high to justify moving a user's entire data corpus into the shared sphere they provide.

1.3 Contributions

This dissertation presents a new strategy for data synchronization across a user's devices by separating a device's *view* of the file system from the reality of where files are stored. Using this strategy, a person uses a single file system that appears exactly the same on every device. Data consistency is handled transparently, while the latest version of a file is retrieved from a storing device when needed. Each device can see every file regardless of where it is stored, and applications can access files transparently without change.

This dissertation makes the following statement:

> Transparent, on demand partial replication of a user's data corpus across their devices, allied to flexible pre-fetching and synchronization policies, provides an efficient use of storage and network resources with modest trade-off of access latency and availability.

This dissertation also presents a prototype implementation of this strategy, called Unico. This system is implemented as a user-level file system, allowing a user to have access to their files in a transparent interface.

The thesis statement above brings forth four main objectives:

Efficient use of storage resources. Platforms like Dropbox and OriFS, by default, maintain a local copy of all the user's data in all the user's devices. In the approach proposed in this dissertation, data is transferred to each device on demand, or pre-fetched based on configurable policies and usage analysis. This strategy leads to reduced storage requirements, since data that is not used in a particular device is never transferred to it.

Efficient use of network resources. The limited amount of data stored in each device also implies that less bandwidth is needed to transfer files between devices. In addition to that, a process of adaptive synchronization frequency reduces these requirements even further by allowing an idle node to stop receiving constant updates for a particular set of files.

Transparent replication of data. One of the goals of this dissertation is to allow users to access their files with no required knowledge of where the data is stored. This dissertation proposes a system where the latest version of a file is automatically retrieved as soon as its data is needed, and a user does not need to change their regular workflow.

Modest trade-off of access latency and availability. It is clear that, by not storing data locally, a user may be required to wait for the data to be retrieved before having access to it. In addition to that, if files are stored only in unavailable devices, the content may not be retrieved in some instances. In order to allow a user to adjust this trade-off to a desired degree, this dissertation includes configurable policies that adjust the behaviour of pre-fetching and invalidation mechanisms.

In summary, these are this dissertation's main contributions:

1.3.1 Transparent On Demand Synchronization

Possibly the core contribution of this dissertation is a transparent on demand platform for single-user, multi-device data synchronization. This platform transfers data on demand between devices without explicit user intervention and without requiring a user to have previous knowledge of the location of the latest version for the required content.

User's data is distributed across available devices, adapting the storage of each file's data to each device's capabilities and interests. Applications have the ability to explicitly identify where a file is stored by accessing extended attributes in file's metadata, but are not required to do so. The system also implicitly and transparently transfers files on demand as devices request them.

1.3.2 Metadata-Only Transfer

Certain operations such as searching and organizing can often be completed locally without requiring file content. To facilitate local search, the synchronization platform provides an enriched set of extended attributes for each file that includes type-specific information such as photo EXIF tags. Applications can also use extended attributes to determine where a file is stored, and to receive dynamic, access-latency estimates for non-local files.

1.3.3 Object Location Algorithm

In order to allow data to be transferred efficiently between peers, an object location algorithm provides a heuristic to identify and rank potential peers that may store specific content. This heuristic is based mainly on local knowledge of content availability in nearby nodes, and expected latency to retrieve such content.

1.3.4 Adaptive Synchronization

Unico also provides an adaptive synchronization process that allows access to the latest version of a specific file even with reduced synchronization frequency in place. This process is started when one or more devices are actively modifying a set of files, while other devices are idle. The latter nodes will mark these files as invalid. Once this happens, active nodes will drastically reduce the frequency at

which updates for those files are sent, allowing for further reduction of bandwidth requirements. Nodes with invalid paths will automatically revalidate paths once files are visited, so the reduced synchronization frequency is not expected to affect conflict rates.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows.

Chapter 2 describes the background of the dissertation and the work related to it. It describes existing research in areas related to this dissertation, particularly related to centralized and distributed file systems, peer-to-peer (P2P) synchronization platforms and version control systems.

Chapter 3 provides a brief description of the main objectives proposed by this dissertation, with particular consideration about how they contrast with existing approaches to data synchronization, and how they are achieved in the implementation of this dissertation.

Chapters 4 to 6 describe the implementation of the Unico prototype. Chapter 4 describes background concepts and platforms used in the implementation of the prototype. Chapter 5 overviews the model and specification of the key components in the implementation. Chapter 6 details implementation decisions and how particular issues encountered during the research are carried out.

Chapter 7 presents an evaluation of the working prototype. It examines the use of network and storage resources in a set of scenarios, as well as the impact on access latency, conflict rate and availability. Some detailed tables of result data are presented in Appendix A. Finally, Chapter 8 concludes the dissertation and presents directions for future work in this topic.

Chapter 2

Research Background

Distributed file systems are modules that provide a user with access to data using a regular system file interface, but distributes the actual storage of the data across other hosts in a network. Several strategies have been used to store these files, focusing on aspects such as security, fault tolerance, dependability and performance. Files may be stored in a single server or multiple servers, or even in peer clients [57].

This chapter presents a survey of existing research publications related to distributed file systems, and how they compare to Unico. It also describes research in related areas, such as distributed Version Control System platforms.

2.1 Centralized Distributed File Systems

Traditional distributed file systems are based on a client-server architecture. In these systems, a centralized server coordinates both the storage and versioning of all data in the file system. Often clients store little to no data locally, usually limiting local storage to caching. Some systems, however, store local copies of the files, potentially modified, but usually rely on the server for synchronization and conflict resolution.

Network file systems such as Network File System (NFS), Cedar File System (CFS) and Samba allow transparent access to data stored in a centralized server [6, 22, 55]. These file systems, however, have limited local caching or ac-

cess to previous versions of the data. They also rely on a single centralized server, and do not perform well over wide-area networks.

The Andrew File System (AFS) and Coda are examples of distributed file systems that store data locally and support disconnected operations [7, 31, 53]. These systems, however, still rely on a centralized server for conflict resolution and data synchronization, with no direct communication between peers.

Moose File System and Google File System are distributed file systems that provide redundancy and reliability by distributing data across a network of chunk servers, while showing data to the user as a single resource [20, 21]. DMooseFS is an extension of MooseFS that provides a cluster of servers for management and search, instead of a centralized server [67].

Lustre is a parallel distributed file system popular in large-scale cluster computing that also distributes data in a network of metadata servers and object storage servers [58].

Some distributed file systems, such as Elephant, Ceph and Hadoop Distributed File System (HDFS), have implemented mechanisms that allow a user to retrieve previous versions of a file or directory [52, 56, 64, 65]. These systems, however, rely on a central server or cluster to handle storage and consistency.

2.2 Cloud Storage Systems

Distributed file systems based on cloud storage, such as Dropbox, Google Drive and iCloud, provide synchronization mechanisms based on centralized storage [2, 16, 24, 48]. By default, these systems store a copy of the entire data corpus in all peers, and transfer files back to the cloud in the background when modified. Conflict is solved in the cloud servers. This strategy allows for disconnected operations and faster writes without network transfer, but there are a few problems. First, depending on the amount of data that is being synchronized, the cost of storing data in the cloud is significantly higher than local storage. Although these systems provides free, limited versions, these often lead to users being required to choose which files should be synchronized between devices, limiting the usability of this option. In addition to the storage cost per se, transferring data to the cloud

may be too slow, and may incur additional network costs associated to this data transfer.

Upthere is a commercial cloud-based platform that uses the "cloud" as the primary storage location. Any data stored in local devices is considered a cache, allowing it to easily be freed if space is needed [62]. Changes to files are stored directly in the cloud, instead of being synchronized once the file is saved. Although this approach promises to reduce conflicts resulting from synchronization, it still suffers from some of the drawbacks of systems like Dropbox—namely the cost of keeping and transferring data in the cloud—and centralized systems like Coda—such as reliance on a central authority for communication and conflict resolution.

2.3 Version Control Systems

A Version Control System (VCS) platform is a system used to manage changes in documents, programs and other kinds of information stored as computer files. A typical VCS allows multiple contributors to work in the same project simultaneously, while maintaining the history of all involved files. Contributors also have the ability to synchronize their local copy of the project with other contributors, based on changes performed asynchronously by each one of them. As Unico is built on top of a VCS (Git), it shares some characteristics with them, and some research objectives of VCS platforms are associated to the motivation behind Unico.

Among other tasks, a typical VCS is usually known for keeping a list of changes and/or snapshots of the project and its related files, allowing contributors to analyze the history of the entire project or of individual files, as well as restore previous versions of these files. Most available VCS platforms will also be able to synchronize changes performed by different contributors, merging these changes to create a single snapshot of the project [44].

Available VCS platforms may be classified by several criteria. In particular, these platforms are commonly distinguished between centralized and distributed systems. In a centralized (or client-server) VCS approach, a central repository (server) is used to maintain the revision history, and all contributors (clients) connect to this repository to perform revision control tasks such as synchronization and history examination. Clients usually maintain only the latest version of the

repository locally. By contrast, in a distributed (or decentralized) VCS approach, every contributor keeps a local copy of the entire repository, and synchronization is performed between peer repositories. In this approach, common operations such as creating a project snapshot (commonly known as a commit), viewing history, and reverting changes are performed locally. No central repository is necessary, although one may be used to simplify access control or to improve reliability [44, 66].

There are some efforts towards building a file system that provides access to a version control repository through virtual access to the repository files. For example, the Cascade File System is a proprietary system that allows a user to access the files in a Subversion or Preforce repository without the need to checkout files [11]. Similarly, cvsFS maps a CVS repository to a file system [13]. These systems, however, implement this access to centralized VCS platforms, which present a different set of challenges and aspects than distributed version control.

The Pastwatch system was developed as a distributed VCS focused on removing the need for a centralized server [9]. This system stores a shared branch tree using a Distributed Hash Table (DHT), while periodically accessing this table to update or add new versions to the branch tree. This system, however, in addition to failing to provide a space efficient environment, has long been discontinued, and more recent distributed VCS platforms such as Git include features that surpass many of the original motivations of this system.

Kapitza *et al.* presents a replicated VCS based on Git [34]. This system, called DiGit (Distributed Git), uses fault-tolerant object replication for increasing the availability of a central repository. This approach, though, is focused on the server dependability, and does not address local use of bandwidth and storage resources.

Git Revisions As Named Data (GRAND) is an extension to Git that enables the synchronization of Git repositories over a Content-Centric Network (CCN) [33]. This system focuses on enabling the acquisition of data such as Git objects by name, instead of based on their physical address. This work focuses on the communication operations in Git, such as cloning and synchronization between peers, and does not modify the local object database.

A fully decentralized P2P-based VCS platform, called PlatinVC, is presented in [42]. This platform uses P2P concepts to create a virtual global repository, re-

moving the need for a centralized server, as well as some of the shortcomings of centralization, such as a single point of failure, ownership, and lack of scalability. This system, however, similar to regular distributed VCS platforms, keeps the entire repository locally, in addition to data used for P2P synchronization of the virtual server.

Git repositories have the option to share the object database with other local repositories. Through this feature, a repository will maintain a list of alternate object databases. When an object is available in an alternate database, the object will not be copied to the repository's database, saving storage space. This feature, however, is only available if alternate repositories are available locally in the same host, or through a separate virtual file system.

Git Annex implements an interface for users to store large files in external storage (external drives and media, cloud services, etc.), allowing users to easily identify the media containing some data and to share some of this data [28]. This platform, however, relies on the external media to be available when needed, or the additional cost of cloud services.

2.4 P2P Distributed File Systems

Most application-level services on the Internet are based on a client-server architecture, in which the service is provided by a single host or cluster of hosts, called servers, to other computers in the network, called clients. In this approach, clients usually do not share any of their resources with other clients. In contrast, a peer-to-peer (P2P) architecture is one in which all the participants in the network, individually called peers, assume a similar role in the network, both requesting service and providing resources to other peers [54].

One of the main advantages of P2P networks is the scalability of the network. In traditional client-server architectures, a higher number of clients means a higher demand for service, while the resources available to provide that service at the server, such as bandwidth, storage space and processing time and power, are unchanged. In contrast, the addition of a peer in a P2P network means that the demand for service is higher, but the resources available to provide this service are increased in a similar fashion [38].

Among the challenges of file systems that rely on P2P for communication is reliably identifying the latest version of the file system data. Peers in a distributed file systems based on P2P communicate directly with each other to transmit file data and coordinate conflict resolution. Since disconnections and network partitions may impede communication between all nodes at some times, a consensus may not be reached until all nodes are connected at the same time.

OriFS, similar to Unico, uses a Git-based repository as backing store for a FUSE file system [39]. It assumes abundant storage in all devices, and aims to include all the files and their history on every synchronized device through an aggressive pre-fetching approach. A limited subset of the nodes can be set up to fetch objects on demand, but these nodes assume that most other nodes have all the content; copies of files not stored locally are located using a discovery protocol on a per-file basis, which is not optimal if a large number of nodes stores limited data. Our approach uses a more appropriate object location algorithm based on object metadata, and uses this metadata for locally-based, network-wide search and to identify file storing nodes.

Cimbiosys is a replication platform that allows partial replication of data among nodes, based on predefined filtering rules [49]. Data is synchronized among nodes, and each node only stores content that matches a specific set of rules, while storing the metadata of remaining files for synchronization purposes. Cimbiosys shares with this paper the idea that nodes can't store everything, and uses peer-to-peer communication to achieve eventual filter consistency and eventual knowledge singularity. Cimbiosys differs from Unico in that object transfer is automatic based on predetermined rules and filters, and the transfer of files that don't match these rules has to be manually done by the user (usually through changes in a device's filter list), contrary to the transparency goal of Unico.

Ficus and Rumor are distributed file systems that allow updates during network partitions [25, 26]. These systems rely on what is described as an optimistic eventual consistency approach with reduced update frequency to decrease bandwidth and connectivity requirements, and automatic per-type conflict resolution scripts. Users of these systems may not be able to see the latest version of a file when switching devices due to the delayed update policies. Unico is based on a more

recent assumption that network disconnections are not as frequent, and that bandwidth can be saved by transferring data on demand [47].

BitTorrent Sync uses an approach similar to that of Unico or OriFS, but the synchronization is performed using the BitTorrent peer-to-peer algorithms [5]. This approach is also based on a similar networking model as the one used in this paper, where nodes are assumed to be connected to each other most of the time. Similarly to Dropbox and OriFS, however, all devices will store all the contents of all files, including those that are not actually used in all devices.

OwnCloud is a storage and synchronization platform that allows a user to host its data in their own devices [29, 45]. Although it gives a user access and control to its own data, the platform is based on a centralized server (or cluster), and does not take full advantage of the capabilities of a user's devices.

Chapter 3

Rationale

In this chapter I present a typical scenario that describes the main problems this dissertation aims to address. I also describe an overview of the main objectives that are confronted in this work, as well as the obstacles and issues that need to be overcome.

3.1 Sample Scenario

Consider a user, Bob, who uses the following devices each day: a home computer, a work computer, a laptop, a smart phone, a tablet, and a TV set-top box. Bob maintains all his personal and work data in Unico. This data includes a photo library stored partly on his phone, tablet, laptop and home computer and perhaps other devices. Bob uses a variety of applications to access his photos — a camera application to take photos, a slide-show application to display them, and an editing application to organize, archive, tag and enhance them. Each of these applications uses a local file system that lists every photo along with its extended attributes.

When the camera application takes a photo, for example, it might instruct the system to transfer it to his laptop or home computer. Or, instead, the slide-show or editing software could fetch the file on demand from the phone when needed or could set the file's metadata to direct that the file be moved or copied from the phone.

Similarly, Bob's TV set-top box might be set to display a slide-show of recent, highly rated photos organized by day and location. Since meta information such as photo date, ranking, and location is replicated in Unico on the set-top box, the slide-show application can easily and quickly determine the photo files to display and the order in which to display them and can then fetch them on demand during the slide-show. Since the device is only interested in a subset of the files, based on metadata (such as the ranking and date), it can avoid transferring unnecessary files by checking extended attributes of those files before retrieving their content.

Although Bob usually views some of his files on only a subset of his devices, he would like to have access to them on any device. For example, Bob does not usually handle personal photos at work, but he would like to show one of his co-workers the latest pictures of his vacation trip. Bob also does not keep all his work documents on his tablet or phone, since it has limited space, but during lunch he decides to practice his presentation, and does not want to bring his laptop to the restaurant.

While at work, Bob is working on an important large file (like a video), and making constant changes to it. Although Bob will later continue working on the file at home or on his laptop, and would like to have access to the latest modifications when there, this video file is very large, and he would prefer not to send these changes constantly, since they would use too much bandwidth at home. Bob would also like to see the video from his phone, and sending the data constantly to the phone would drain all its battery. Additionally, if the file is generated from a collection of source files and pieces by a specific application, the phone can limit its retrieval only to the final compiled file.

Other scenarios with the same characteristics will play out with Bob's other files. Bob can manipulate arbitrary files at work and then see these updates on any other device when needed. He can search all of his files on any device and he can organize where his files are stored, by making local changes on any device.

As illustrated by the example above, the goal of this dissertation is to provide a single view of all the user's data, consistent among all the devices, but making smart decisions on what data to transfer and when. Devices do not necessarily keep all the data locally, but have access to it by maintaining connectivity with other nodes and locating the data as necessary. As data is modified in one node, the

modification is made known to all other nodes and data is synchronized as needed, while avoiding data transfer if the new data is not expected to be immediately used in the other device.

Several distributed file systems are based on a network model that assumes frequent disconnections and limited connectivity between nodes. This work assumes that this model is no longer current. Most devices have some degree of connectivity to the Internet at all times, and although network fragmentation might happen, it is rare enough that it should be handled as an exception [47].

3.2 Objectives and Challenges

The main objective of this dissertation is to present the user with a consistent view of all the data, providing access to all files on demand. To achieve this goal I confront the following challenges.

3.2.1 Heterogeneity

The first challenge is to manage storage heterogeneity efficiently. Some devices have vast storage capabilities and others do not. Devices with limited storage should not be required to store content that is not needed locally.

Unico allows file data to be stored on arbitrary nodes and gives applications and users direct control over file location. Users may manually specify which files to store in each device, or allow Unico to make the decision based on usage analysis or preset policies for each device. When a non-resident file is accessed, the system fetches the file's content from a remote node.

3.2.2 Data Access Latency

The second challenge is to keep access latency reasonably low, given that many nodes store only a subset of files. One defining feature of Unico is that it allows different caching strategies for directory listings, file metadata and file content. While file content is usually cached in an as-needed basis, file metadata and directory listings can be cached much more aggressively. Most nodes are expected to replicate most directory listings and a substantial subset of the metadata for the entire global file system. Our goal is to support a rich set of file search opera-

tions locally by restricting these searches to metadata (and possibly locally-cached content). The effectiveness of this approach will be limited by the richness of the metadata. Unico thus augments the standard file metadata to automatically include type-specific content-based information such as EXIF photo tags as well as system-level information such as file location and availability.

3.2.3 Consistency

The third challenge is maintaining file-system consistency. Unico does this by relying on techniques borrowed from version control systems. Files are stored as histories of immutable file versions (called objects), each named by a collision-resistant hash of its content. Directories, including extended metadata, are versioned in a similar way; directories map file names to their content hash value.

The global file system consists of a forest of commit points, each of which names a particular version of the entire file system by storing the hash of its root directory. The file system on each node consists of one of these commit points plus uncommitted local updates. Each node periodically produces a new commit locally, as changes are identified, and then propagates this commit to other nodes where it is merged with local information to produce a new commit on that node. Nodes continue this process in one-to-many fashion advancing each node's view of the current state of the system. The information nodes exchange to merge commit points is restricted to directory listings on paths to modified objects. File content is transmitted only if directed by prefetching rules or as needed to resolve update conflicts.

3.2.4 Managing Update Overhead

In the naive approach to synchronization, metadata updates made on one node are frequently flushed to every other node. Frequent updates may be needed to minimize the chance of conflict that arises when nodes access out-of-date information. A personal file system like Unico presents a simplified version of this problem, because the typical sharing pattern consists of a user sharing with themselves as they move from one device to another. The synchronization frequency needed for this type of sharing is much more course-grain than that needed for concurrent

sharing, and thus Unico has the opportunity to exploit this characteristic to further reduce synchronization overhead, which has been shown to be a problem for systems such as Dropbox that often handle concurrent sharing and thus perform frequent updates [37].

To realize this benefit, each Unico node adaptively invalidates the portions of its namespace that are not under active local use and are being actively modified elsewhere. This invalidation is triggered when a node receives a merge request from another node as a result of update activity on that node. The receiving node identifies the overlap between the namespace regions updated remotely and those that are inactive locally. It chooses region sizes to be initially very large (e.g., the entire file system or large sub-trees) and then adjusts sizes in de-escalating fashion as necessary. It then marks these regions as locally invalid and informs the updating node. The updating node responds by lowering its update frequency for the invalid regions when synchronizing with that node. In the common case, where a person uses only one device at a time, the entire namespace on every idle node is typically marked invalid and all metadata updates are propagated with low frequency. This degraded update frequency is selected to maximize write absorption at the updating node while it is active.

When a previously idle node accesses a file in an invalid path, that path is revalidated. This process forces a synchronization with remote nodes, allowing the idle node to obtain the latest version of the file. In addition to that, idle nodes revalidate invalid paths at regular intervals even with no file access. This mechanism allows nodes to keep a reasonably up-to-date version locally, while limiting the effects of disconnections while invalidation is in place. If, during this update, the conditions for invalidation are still present, the path is invalidated again.

3.2.5 Availability

A potential disadvantage of not replicating all file data everywhere as other systems do is that the data a user wants might not be available when they want it. There is, of course, an inherent trade-off between replication overhead and availability. The advantage of Unico is that it provides applications and users with the basic mechanisms, built into the file system, to examine and adjust file location and

replication, allowing different applications to choose different availability policies that trade the cost and benefits off against each other in different ways.

3.2.6 Transparency

Finally, a central goal of Unico is file system API transparency. Applications access the global file system through a virtual file system using a regular file system API. Any file accessed by a user that is not locally available in that device is transparently transferred on demand from another node before it is served to the user.

Extended metadata are presented to applications for reading and writing through file extended attributes. Applications can use these attributes in various ways. One important use of extended attributes is to allow applications to distinguish local and remote files and to handle each differently, if they choose. This distinction is important because remote files have different performance and failure semantics. Applications should be able to interact with the metadata of the entire, global file system, but this ability comes with the risk that an application might access a remote file without accounting for its higher latency or the possibility of temporary unavailability. Waldo *et al.* argue in [63] that distributed systems require that the programmer of an application must be aware of the latency and other issues inherent in accessing data that may be stored in a different address space. And so, for applications to make effective use of the system, it is likely useful for them to examine certain extended attributes such as file location or expected latency.

In addition to per-file location, the system also tabulates node connectivity information that allows it to present an access-latency estimate as dynamic metadata of each remote file. A multimedia application, for example, may change its buffering policies based on the local availability of a file's content or the expected latency to retrieve it.

Although Unico presents users with mechanisms for further tuning the availability and behaviour of the file system, it does not require most users to make use of these mechanisms. Unico is developed so that, by using its default policies, or even with basic configuration options, users can have a reasonable expectation of availability and transparency without further fine tuning.

Chapter 4

Background Concepts

This chapter describes some of the concepts related to the research presented by this dissertation. Section 4.1 defines file systems and describes its application in modern operating systems, while Section 4.2 describes VCS platforms and some of the strategies used to store and synchronize changes by multiple devices.

4.1 File Systems

The most popular operating systems in use today, such as Linux, Mac OS X (and other POSIX-based systems) and Microsoft Windows, present persistent data in a hierarchical structure of directories and files [12, 50, 51]. In this structure, regular files contain blobs of data in an application-specific format, while directories link names to files and other directories (called subdirectories). A file may be identified by a path, which corresponds to a sequence of names that link the root directory, through a sequence of subdirectories, to an entry that links to the file. In addition to regular files and directories, individual operating systems may define other types of files, such as symbolic links and named pipes.

Persistent data is normally stored in block devices like hard disks, solid-state drives and optical disks. These devices do not inherently use any path-related structure to store data, storing all data in blocks of contiguous bytes, and addressing data based on internal addresses. In order to organize data into a hierarchical structure of files and folders, and present it to users and applications, the operating system uses

modules called file systems. Applications access persistent data by calling particular operations in the operating systems, identifying the intended file by using a file path. The operating system then determines the specific file system associated to that path, and directs the request to that file system, which will translate the operation and path and perform the corresponding task. This provides a user transparent access to its data, and the user is not required to have knowledge of the data storage structure or direct access to the underlying storage platform [14].

In addition to directly providing a hierarchical view of data stored in local block storage devices, file systems may be used to provide a similar user interface to other types of storage media. Examples include [6, 30, 36, 41, 52, 55]:

- network file systems, that give users access to files stored in other devices across a network (e.g., NFS, SMB, SSHFS);
- virtual file systems that compute data on request (e.g., `procfs`, which provides data on the availability of system resources);
- virtual file systems that give a user direct access to read and update archived files like tar or zip;
- virtual file systems that encrypt or compress data before it is stored in persistent media;
- versioning file systems, which automatically store old versions of files and provide snapshot access to archived data.

Although the most common file systems are implemented as kernel components and run in kernel space, in some cases a file system running in user space is more suitable. In particular, such a file system can be mounted by a non-privileged user, can be more easily installed and updated without changing the kernel, and allows the system to be less susceptible to corruption in case of a programming error or misconfiguration, due to its less privileged running environment [10, 36].

One of the most common mechanisms to implement a user-space file system in POSIX-based systems is Filesystem in Userspace (FUSE) [36]. FUSE provides a library that includes a series of callback functions that can be associated to file

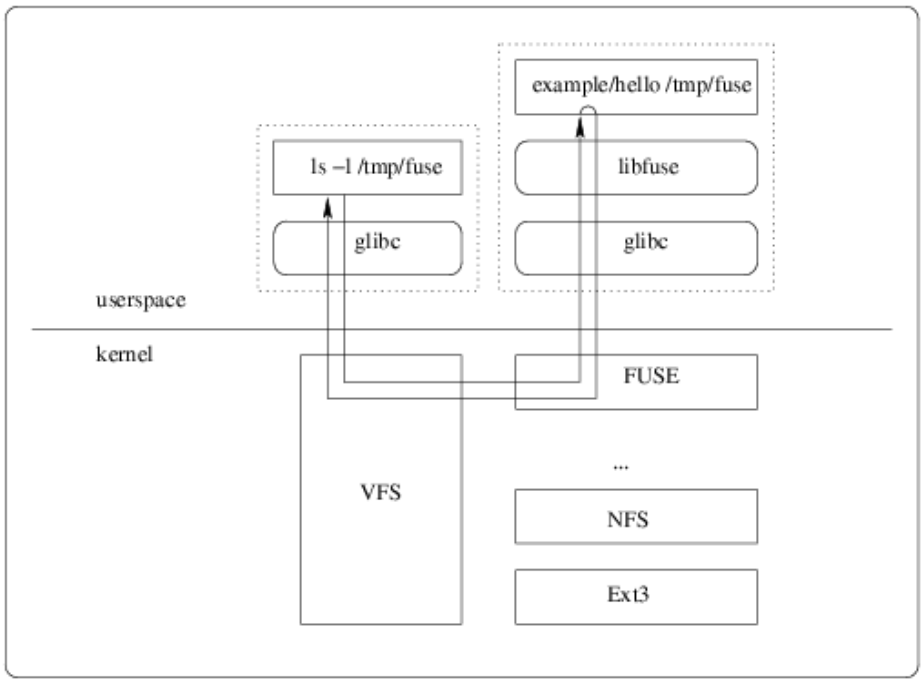


Figure 4.1: Control flow of a system call to a FUSE-mounted directory. In this example, the directory `/tmp/fuse` is mounted to a process called `example/hello`. The call to list the directory entries in the directory is forward to the virtual file system process in the kernel, which directs it through FUSE to the corresponding callback function in the file system process. The result of this call is processed and sent back to the user application [36].

operations, such as reading and writing in a file, renaming and deleting files, reading and manipulating directories, and so on. Once a program using FUSE is linked to a specific mount point (a file path prefix), a user that accesses files and directories within this mount point will be directed to the FUSE kernel module, which will direct the operation to the corresponding callback function in the file system program in user space. The returning value of this operation will then be returned to the user by FUSE. Figure 4.1 illustrates the control flow of a system call to a FUSE-mounted directory.

In this dissertation, the Unico prototype is implemented using FUSE. Details about the callback functions implemented by Unico are described in Section 6.1.

4.1.1 Privacy and Trust

A common concern related to data synchronization in file systems is data privacy. File systems should guarantee that users' files are only visible to the users themselves and those with whom they provide proper authorization.

A basic requirement of data privacy is authentication. When the file system in one device connects to another device, it must provide proper assurance that they are authorized by the user to access the user's repository. In a centralized system, including cloud-based systems like Dropbox, this authentication can be done using a central authentication protocol like OAuth [27].

In file systems based on direct peer-to-peer communication, device authentication must rely on a component of mutual trust. Both the node initiating a connection and the one being connected to must provide proof that they have been granted permission to access the data. A simple implementation of such a mechanism would be a predetermined list of authorized nodes, but addition and removal of nodes would be harder in such an implementation.

BitTorrent Sync uses an authentication mechanism based on capabilities and digital certificates [32]. Each node generates an X.509 self-signed digital certificate. Existing nodes maintain a list of authorized peers and their certificates, and validate these certificates when connection is established. To provide a new node access to a repository, an existing node generates a random hash, and provides a link including the address of the existing node and this hash. The new node will use this hash to establish a connection, and will provide the hash to the existing node, which will compare it against its own list of valid hashes. These hashes expire after a set amount of time, or once they are used by a new node.

Since privacy concerns are outside the scope of the research described in this dissertation, the Unico prototype described here does not fully support authentication. However, all communication is encrypted by transmitting all data using an SSL layer, as described in Section 6.3. An authentication process similar to the

one described in this section could be added in a future version of Unico without major impact on the overall functionality of Unico.

Some file systems provide the ability to specify which users or groups in the system are allowed (or denied) access to particular files or directories. Since this type of permission is beyond the scope of this project, and because Unico is based on Git, which does not provide it, Unico does not support this kind of permission specification. The only type of permission allowed in Unico is the ability to specify if a file is executable or not. One complication of implementing this type of feature is the fact that different devices have different users, and users in different systems cannot easily be mapped to each other.

Several distributed file systems, such as Dropbox and OriFS, provide the ability to share a specific directory with other users, without sharing the entire repository [23, 39]. If any user that shares this directory changes files in it, all users receive the related changes. This ability presents its own challenges in a peer-to-peer system, as not only do users need to make sure that devices are simultaneously available, but also they need to trust each other. Unico does not support this feature, but it could be implemented by replacing the tree being shared with a commit, and merging changes on that commit itself. Git repositories support this strategy through what is called a git submodule [8].

4.2 Version Control Systems

Computer systems can provide support to the cooperation between peers in a collaborative project or task. There are several tools and systems used for this collaboration. A Version Control System (VCS) is a system used to manage changes in documents, programs and other kinds of information stored as computer files. A typical VCS allows multiple contributors to work in the same project simultaneously, while maintaining the history of all involved files. Contributors also have the ability to synchronize their local copy of the project with other contributors, based on changes performed asynchronously by each one of them [3, 44].

Among other tasks, a typical VCS is usually known for keeping a list of changes and/or snapshots of the project and its related files, allowing contributors to analyze the history of the entire project or of individual files, as well as restore pre-

vious versions of these files. Users are also able to inspect details about specific changes, such as who, when and how it was made. Most available VCS platforms will also be able to synchronize changes performed by different contributors, merging these changes to create a single snapshot of the project. Additionally, a VCS will frequently give a contributor the ability to keep separate branches of project development. This feature may be used to maintain several unrelated threads of development simultaneously, as well as allowing changes to be done in previous versions of a project without affecting the main thread of development [44].

This section explores recent development in popular VCS platforms, with an emphasis on the distributed VCS strategy. The basic functionality of Git is explained, as well as internal concepts relevant to the Unico project implementation.

4.2.1 Distributed Version Control Systems

Centralized version control systems are tools used to manage changes and collaboration for a set of files and directories comprising a project, based on a centralized server. This server is responsible for creating, maintaining and reverting snapshots (also known as commits) of the project and keeping all project's metadata, providing access to the repository history, and merging changes between different collaborators [44, 66].

In a distributed version control system, a different approach is taken, in which each collaborator keeps a local working copy of the repository. In theory, no centralized version exists, and synchronization is done through communication between peer repositories, although a centralized peer can assume an authoritative role if deemed necessary. In effect, each collaborator keeps a forked copy of the project locally, and develops on top of this working copy. When it becomes necessary to release modifications done by several collaborators into an official version, these working copies are merged into a single copy.

Because collaborators keep a local copy of a VCS repository locally, common operations in a version control system, such as committing a snapshot of the project and comparing versions, are also done locally. This functionality allows collaborators to work individually in a more productive way, making use of VCS features without the involvement of a network for most common tasks. It also allows users

that do not have permission to submit an official version, or that intend to work on a draft of a modification that should not be immediately published, such as an experimental feature, to work on a local version of a project and still take benefit of the VCS platform features.

In some distributed VCS software platforms, users are able to maintain several local implementation branches simultaneously. This feature, among other purposes, allows a user to easily switch between developing new features in an unstable version and fixing a problem in a stable version.

Because there is no centralized server to maintain version numbering, and collaborators work on versions asynchronously, the globally acceptable identification of a commit cannot be done through sequential numbering, like in most centralized VCS platforms. A longer, globally unique identifier has to be generated for each revision. Common strategies for commit identification include content hashing or including the author/repository identification in the identification.

4.2.2 Git

One of the most popular distributed VCS platforms, and among the first free open source alternatives, is Git. This system was initially created to maintain the development of the Linux kernel, as a replacement to the proprietary BitKeeper system, and has since become available as a separate tool for version control [8, 60]. Git is widely used in code development projects, especially in the open source community, including projects like Android [1], Drupal [17], GNOME [59], One Laptop Per Child (OLPC) [43] and Perl [46].

One of the main characteristics of Git, particularly in contrast to some centralized VCS platforms, is that it keeps track of content, and not of individual files. In other words, Git handles each version of the system (which is internally called a commit) as the snapshot of all file contents in the repository at a specific point in time, and not as a difference with previous versions (although the difference between two objects may be used for compressing purposes). One consequence is that operations such as renaming, moving and copying files are handled algorithmically instead of being manually specified by a user [8, 60].

Git describes all the contents in a repository based on four types of objects: a *blob* represents the contents of a file; a *tree* contains the listing of a directory; a *commit* represents one version or snapshot of the entire repository; and a *tag* links a string representation to another object (most commonly a commit). These objects are further discussed in Section 4.2.3.

Every Git repository contains a Git folder (usually named `.git` in the root directory of the repository). This folder contains, among others, a directory with all the objects in the repository, called the object database. It also contains configuration settings and pointers to branches, logs and other information used by the repository.

Git implements a large set of operations that can be used to perform common tasks in a repository. These operations are usually divided in two main categories: porcelain and plumbing. Porcelain operations are high-level operations, and the most commonly performed by the repository user. This category includes operations like copying a repository into a new location (`clone`), creating a new commit, or snapshot (`commit`), analyzing and comparing previous versions (`log`, `diff`) and synchronizing the repository with other repositories (`fetch`, `pull`, `push`). Plumbing operations are low-level operations, usually called internally in the system, although they may be called by the repository user for advanced tasks. Operations in the plumbing category include creating and obtaining a specific Git object, generating a list of unused objects for garbage collection, calculating an object's SHA-1 hash and sending and receiving individual objects from peer repositories.

Based on the fact that Git is a distributed VCS platform, and that branches are created frequently, Git presents some efficient merging algorithms. These algorithms deal with projects that were changed in parallel by two or more collaborators, and synchronize changed files, including those that may have resulted in conflicts. The following merge algorithms are available:

- A *fast-forward* merge is used when one commit to be merged is a direct child of the other commit, i.e. only one of the commits actually moved forward, and the current commit is updated to the new commit.

- A *resolve* merge uses a 3-way merge strategy in which both commits and a common ancestor are used to identify changes done through each side and applying them both.
- A *recursive* merge is an extension to the resolve merge, and has some special treatment for cases in which there is more than one common ancestor. This is the default algorithm when only two commits are merged and a fast-forward merge is not possible.
- A *subtree* merge is an extension to the recursive merge used mostly in cases where one project is merged into another one.
- An *octopus* merge deals with merges comprising more than two commits. It is usually recommended that this strategy be used only for changes that don't result in major conflicts. This is the default algorithm when more than two commits are merged.
- An *ours* merge ignores any change done by the other repository, superseding the merged commit with the one found in the current repository and branch. The merged commit is kept in the history of the repository, but its contents are not used in the merge itself.

4.2.3 Git Object Model

In a Git repository, all objects are internally identified by their corresponding SHA-1 hash. This 20-byte identification (usually represented by its 40-byte hexadecimal representation, or a shorter prefix if it happens to be unique in the object database) is used to index blobs (file contents), trees (directory listings), commits and tags. A repository will keep a copy of all required objects indexed by this hash, allowing quick access to an object if its SHA-1 hash is known [8, 18].

Similar to Git, this work assumes, for all purposes, that the probability of conflict in SHA-1 is negligible. For reference, a system with, e.g., one exabyte of data (10^{18} bytes), consisting of objects averaging 10KB each (i.e., 10^{14} objects in total), will have a probability of collision lower than 10^{-20}. Since we assume that most users will have a significantly smaller number of objects in their repository, the possibility of collision is ignored both in Git and in this dissertation.

One of the purposes of using a SHA-1 hash as the object identification is to avoid saving multiple copies of the same content. Because two files with the same content will contain the same hash value, they can be saved only once and thus save system resources. This property also applies to the same file in two different commits in a project. If a file was not modified between one commit and the next one, it will still be indexed by the same hash value, and thus will be saved only once.

Among additional advantages of using the SHA-1 hash to identify an object is that contents received from a questionable source can be verified by calculating the SHA-1 hash of the received content. This feature also makes sure that all copies of the same object in any repository are exactly the same [61].

Blobs, corresponding to file contents, are identified by the SHA-1 hash of the content itself, with a prefix indicating the type of object. A blob object does not contain any reference to the file name or its mode (permissions) flag.

The representation of a tree, corresponding to the contents of a directory, contains a list of all object names contained in the tree, associating them to the SHA-1 hash of their content, the type of object (usually blob or tree), and additional flags such as the execution permission flag. This content is also indexed by the SHA-1 hash of this list.

For each commit, related information fields such as author, committer and a brief description are included in a text representation, in addition to the SHA-1 hash of the commit's root tree. This information is again saved and identified by the SHA-1 hash of the data. In all commits except for the first, the commit information will also contain a SHA-1 reference to a previous commit, allowing a user to obtain the history of a project by following the link to previous commits. In some cases a commit might have a reference to more than one previous commit, which identifies a merge between two (or more) lines of development.

Figure 4.2 shows an example of a Git repository structure. In this figure, the latest version of the repository is represented by the commit object `d291ac`,[1] shown

[1]To simplify the identification of Git objects in this example, only the first six characters of the 40-byte hexadecimal representation of the SHA-1 hash are shown for simplicity. This representation of objects through a prefix of their SHA-1 hash is supported by Git itself, as long as the prefix is unique among objects in the database.

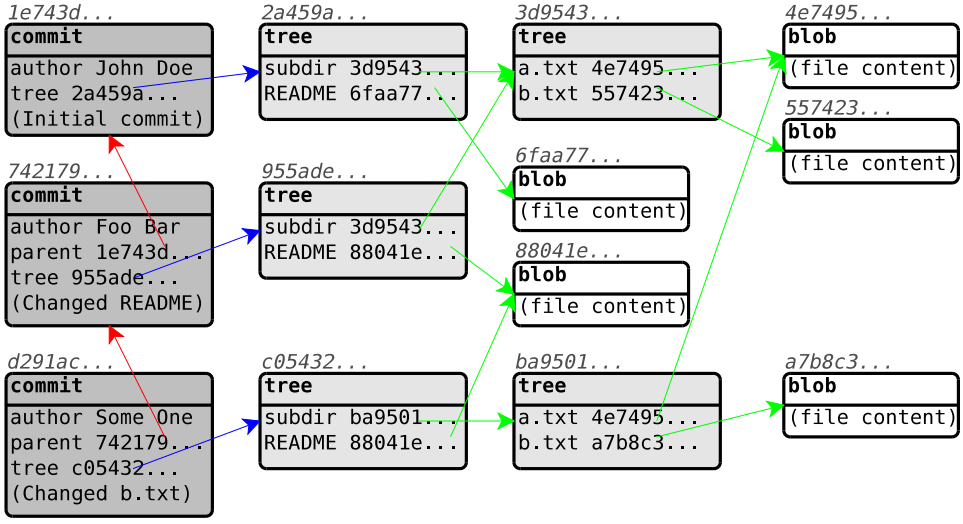


Figure 4.2: Example of a Git repository structure including blob, tree and commit objects.

in the bottom left corner of the figure. This commit has its content represented by the tree object *c05432*, corresponding to the root tree of the project. This tree contains a file named `README` (with contents on blob object *88041e*) and a subdirectory named `subdir` (tree *ba9501*), which contains two files, `a.txt` (blob *4e7495*) and `b.txt` (blob *a7b8c3*).

The latest commit shown in Figure 4.2 is based on a previous version represented by commit object *742179*. These two commits contain different root trees, but both contain the same reference to `README`, pointing to the same blob object, which means that `README` can be considered unchanged in this commit without even visiting the `README` object itself. The `subdir` tree, however, points to a different object. Upon further investigation, a change in `b.txt` can be detected. Through the same process, it is possible to identify that the change between commit *742179* and its predecessor *1e7439* affects `README` only. The simple fact that `subdir` points to the same SHA-1 in both commits allows an application to discover that nothing was changed in `subdir` or any of the files it contains, without the need to recursively visit that directory.

Git, in the same way as other VCS platforms, allows a user to register annotated tags to identify commits that for some reason need to be represented in a clearer way, such as milestones, major versions and releases. These tags may be represented using the same object structure already used for blobs, trees and commits. An annotated tag will contain a tag name and the SHA-1 hash of the tagged commit, as well as additional information such as a description and the tag author. A tag may also be signed by a reliable contributor to indicate that a commit is acknowledged and/or endorsed by that contributor. Although not common, tags can also be used to identify other types of objects, such as blobs or trees.

Chapter 5

Model Design

This chapter presents the design and specification of Unico, as an overview of its key components. A diagram of the main components of Unico and how they interact with each other is depicted in Figure 5.1. A more detailed description of how this model is implemented in Unico is presented in Chapter 6.

Throughout this chapter, as well as in following chapters, the term *node* is used to refer to an individual device, while the term *peer*, or *peer node*, is applied to a set of nodes connected to each other, or to refer to other nodes the node in context is connected to.

5.1 Object Database and API

Some of Unico's components are implemented on top of Git, as described in Section 4.2.2. Unico borrows from Git the design of the object database and the platform for merging changes performed in separate nodes.

A full Unico repository logically consists of an object database that contains, in its entirety, every revision of every file and directory in the entire repository history. Typically, however, each node stores a subset of the repository files and objects, and collectively the nodes contain the entire database.

Unico's object design is similar to that of Git. Every individual piece of information in a repository, such as the content of each version of every file and the listings of each version of every directory, is represented by an object. All objects

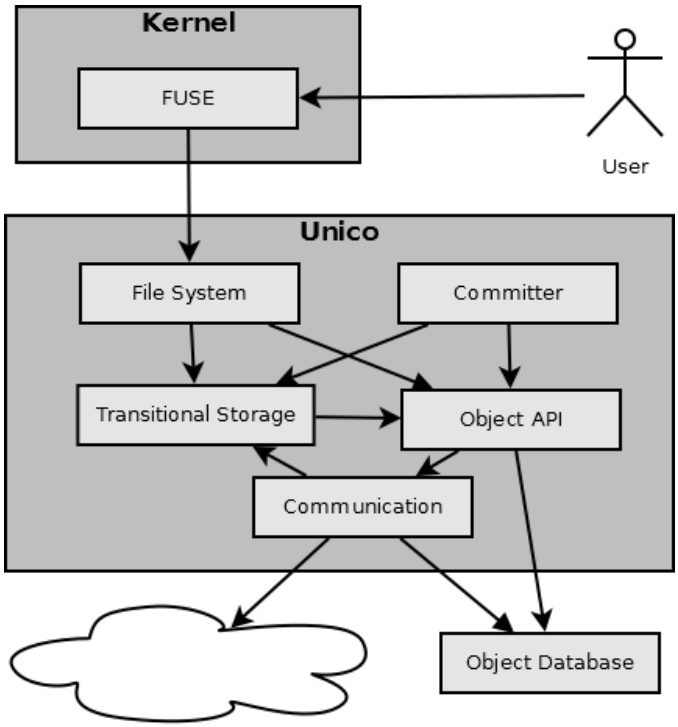


Figure 5.1: Diagram of Unico's components and their relationships.

are identified by the SHA-1 hash value of their content, and as such are immutable. Git implements four types of objects: blobs, trees, commits and tags. Unico does not directly use tag objects. The other types are described in the sequence.

The content of a regular file is stored in the object database as an object of type "blob". This type of object contains as its content an unstructured sequence of bytes. Since objects are identified by the SHA-1 hash value of their content, files with the same content will be represented by the same object. Any change in the file will generate a new object identified by the SHA-1 hash of the new content. Symbolic links are also represented as blobs containing the destination of the symbolic link, and are distinguished from regular files by the mode flags in the directory listings.

A directory is represented as an object of type "tree". This kind of object contains a sequence of names in lexicographical order, with each name linked to a mode (containing flags like the file type and permission) and the SHA-1 of the

object that contains the content of the file. The linked object may be a blob (for regular files and symbolic links) or another tree (for subdirectories). A change in any of the files in the directory will trigger a change in its SHA-1 hash, as explained above. This change will be reflected in the tree content linking the name to a new hash, which will in turn change the SHA-1 hash of the tree itself. The same behaviour will be observed for new files, removed files and renamed files. Changes in subdirectories will also incur in change in content for the parent directory. The subdirectory will point to a new object, which in turn gives the parent directory a new hash, continuing to the root directory.

Unico uses the versioning structure of Git to represent versions of the file system's content. Every version of the system is represented by a "commit" object, containing as its content, among other internal fields, the SHA-1 of the root directory for the file system, and the SHA-1 of the previous commit. This structure creates a walkable linked list of the entire history of the file system, starting from the current version (called the head commit). Although most commits have a single previous commit, a commit may have no previous commit, in case of the initial version of the repository; or it can have two or more previous commits, when it is the result of a merge of changes done in parallel, usually in different nodes.

5.2 File System

Unico's file system component, implemented as a user-mode file system using FUSE [36], acts as a proxy for the entire repository. Unmodified files are accessed directly from the repository, based on the object linked from the current commit object and the tree entries corresponding to the file path. If the object is not available locally, it is transparently fetched from a storing node into the local object repository. In addition, the root file system contains a "`.snapshot`" directory that lists entries for older versions of the file system, which are accessible to applications in read-only fashion.

As the files in the repository are modified, their content is temporarily stored in a transitional data structure called an *exception list*, until the data can be added to the local object repository in a background process. Any access to a file that has an entry in this list will have the corresponding operation directed to the tran-

sitional data. New files and directories are also stored in this list, while deleted files are marked as being non-existent. This list is stored in volatile memory and made persistent using an on-disk redo log, using a tree-like structure map linking names to objects or operations. File operations stored in this list are not visible to other nodes until a new commit is created, which is expected to happen every few seconds as needed.

The file system is designed to prevent access to object content that does not need to be retrieved. For example, moving a subdirectory from one directory to another will retrieve the source and destination parent directories, but not the content of subdirectory being moved or the files contained in it, since their content is not required for the operation. Also, a file being modified with an `O_TRUNC` flag (which truncates the file before it is open) does not require the previous content of the file to be retrieved, since it is being discarded by the truncate operation. These optimizations and others like them are intended to minimize bandwidth requirements and limit data transfer to content that is actually expected to be used.

5.3 Committer

The commit process runs in parallel with normal file access. It incrementally copies modified files and directories from the exception list into the repository. As it does, it computes a new SHA-1 content hash for each object, updates the parent directory listings and creates a new entry in the exception list to use these new listings. When a directory object is updated, the entries naming that directory's subdirectories and files in the exception list can be removed and replaced by a single entry that maps the directory name to the hash of the object created to apply these modifications. In this way, the commit process shrinks the exception list until it is represented once again by a single object referring to the file system's root directory. Once this happens, a new commit record can be created and propagated to the rest of the system.

Unico groups changes happening in fast sequence into a single commit, to avoid a deep commit history for related changes. By default, after a commit is created, Unico waits four seconds after the first change before creating a commit.

This time period has been proposed as an optimal time for grouping file operations in Dropbox [37], and is used for the same purpose in Unico.

5.4 Communication

The communication module is responsible for data transfer between nodes, including the synchronization of the latest version of the data, object transfers and state updates.

In the most common case, every Unico node knows the address of at least one other node in the pool. Once a connection is established, nodes share the address of their other known peers, allowing for connections between nodes that did not previously know each other. Further communication — including data transfer and synchronization — is performed among peers directly, without the need of a centralized node or the cloud. The connection is only interrupted in case of connectivity loss, detected using TCP socket tools like keepalive messages. Since Unico is designed for use by a single user, the total number of nodes is expected to be small, and each node will be connected to all or most of the other nodes. A cloud-assisted node discovery tool is also available as an alternative way to discover additional nodes.

Node synchronization is achieved by broadcasting a commit object to all reachable peers, who then merge it with their current version. As described above, this object is a concise description of local changes made since the last commit that consists of the SHA-1 hash of the file system's root directory and the name of the previous commit, thus creating a walkable history of the repository data.

When a peer receives a new commit object, it merges that revision with its own commit object, using a variation of Git's recursive merge algorithm. It does this by walking the commit history in both commits (the received commit and the local commit), until a common commit (called an ancestor) is found. In most cases, one of the commits is an ancestor of the other; in this trivial case (called a fast-forward merge), the most recent (descendant) commit is used as the result of the merge. If a fast-forward cannot be used, the merge is performed by executing a three-way comparison between both revisions and their common ancestor, examining only directory nodes and file hashes, and only for directories that had changes in one

of the merged commits. File contents are only examined if there is a file conflict. If there are uncommitted changes in the exception list of a local node when a new commit is received, the merge operation is delayed until these changes are applied into a local commit, and then the merge is performed as usual.

Up to this point the only information exchanged between the two nodes are the small commit objects and the remote node's directory objects for paths to modified objects. No file data has been transferred. For objects with conflicting updates, however, the content of the remote file is also required to attempt content-based conflict resolution. Type-dependent conflict resolution is possible in some cases, but if resolution is not possible, both versions are retained and a distinguishing suffix is added to the name of one of them. Due to the nature of commit history, all file versions are available on demand if required. Update/remove conflicts (where one device updates a file while another device removes it) are handled by relying on this history, so that no data or action is lost.

When an object is required locally in a node, it needs to be obtained from another node that has this content. In order to locate a possible source for it, an object location algorithm is used. This algorithm uses information previously obtained in the background to choose and rate connected nodes based on their probability of having the content and the expected latency to retrieve it. The same process is used to obtain structural or content metadata. The algorithm is described in Section 6.5.

5.5 Path Invalidation

In cases where Unico identifies that a single node (or a subset of them) is performing a significant sequence of changes in the same subset of files, while other nodes are idle in respect to those files, Unico triggers a path invalidation process based on configurable policies. When a path is invalidated by an idle node, active nodes will stop sending updates related to the invalidated path to this idle node for a configurable amount of time (e.g., thirty minutes), which allows Unico to save bandwidth and group changes to the same set of files in sequence.

A path invalidation is triggered in a node A when it receives a sequence of commits from node B that modify a set of files that have not been visited by node A in the recent past. Once a list of paths that are idle in the local node but be-

ing modified in other nodes is compiled, this list of invalid paths is broadcast to other nodes. Before these nodes send a new commit, they compare the changes performed in that commit with the list of invalid paths in the receiving nodes. If all the changes in that commit correspond to files in the invalid path list, the commit is not sent to that node. A more detailed description of this process, as well as related algorithms and examples, are presented in Section 6.7.

Paths are invalidated with the broadest prefix possible, considering both the files changed by peers and files visited locally. If, for example, a peer sends changes to files inside the directory `/a/b/c/d`, but the local node has not visited any file inside the directory `/a/b` recently (but has visited other files inside `/a`), the path `/a/b` is marked as invalid.

If a user visits a file in an invalidated path, the local node will revalidate the path and trigger a forced synchronization from other nodes. Nodes receiving the revalidation message will automatically send their latest commit, allowing the revalidating node to update its local version of the file before presenting the data to the user. A finer invalidation path may be triggered in the sequence if necessary. In the example above, if a file in `/a/b/x` is visited, the directory `/a/b` is revalidated, but the new commit triggered by this revalidation may invalidate the path `/a/b/c`.

The path invalidation mechanism allows a node to update files at any frequency without affecting idles nodes or wasting network resources with synchronization messages. At the same time, idle nodes are still able to obtain the latest version of the repository as soon as they become active, reducing the rate of conflicts.

5.6 Metadata

In addition to file contents and directory listings, Unico also synchronizes file metadata among nodes. Some of these metadata are made available to users through extended attributes in the file system. Unico distinguishes three types of metadata: metadata related to a file location (such as modification time), metadata computed from the file's content (such as MIME type), and volatile metadata that depends on the device (such as content location). Each of these types needs to be synchronized and stored differently, depending on its relation to the actual content or the file path

or name, as described below. Section 6.4 describes how these metadata are made available to the user, and provides some examples of each type of metadata.

Structural Metadata Information that is associated with a file location, and cannot be computed directly from the file content alone, is called *structural metadata*. Examples include the creation and modification times, and mode (permission) flags, as well as user-defined extended attributes. This information is stored in hidden objects that are synchronized with the directory that contains the files in question.

Content Metadata Information that summarizes file content in some way is referred to as *content metadata*. Examples include type-independent data, such as the size of the file and the MIME type, and type-specific data, such as EXIF tags, image thumbnails and video length. By definition, two files with the same content are expected to have the same content metadata. Only metadata considered significantly smaller than the content itself are computed and transferred. Unico stores and transfers content metadata in a separate data structure, linking objects to a set of properties. Storage of file metadata will depend on user policies.

System Metadata Information that is added by Unico and whose main purpose is to assist in synchronization is called *system metadata*. Examples include file location information and access-latency estimates for non-local files. This information is used by Unico to identify nodes for synchronization and data transfer, but can also be used by applications through access to extend attributes, in order to facilitate availability estimation.

5.7 Prefetching

Unico allows individual users to specify files that should always be transferred to a device, using prefetching rules similar to those found in Coda [7]. These rules consist on a per-device list of regular expressions that identify directories whose files should always be available in that device. For example, a rule describing the regular expression "`work/*`" in the work computer will automatically transfer any file

inside the `work` directory (or any of its subdirectories recursively) to that device. A rule describing "`photos/*.jpg`" will transfer all files with `jpg` extension in the `photos` folder. By default no rule is specified, in which case only the root tree itself is prefetched by default. A device with a prefetching rule "`*`" will automatically transfer all files every time they are changed. Section 6.6 describes in more detail how this process is implemented in Unico.

Although it would be possible to create prefetching rules based on other criteria in addition to path names, we opted for a simplified approach. While criteria based on extended attributes and metadata could be useful in a few situations, such as automatically prefetching image files with high rating, the additional cost of implementing this kind of prefetching would probably not account for a significant improvement in usability. In addition to that, since extended attributes are not always available in all nodes, this approach would require either the additional transfer of metadata for all files for comparison against prefetching rules, or the synchronization of the rules themselves, forcing other peers to keep track and send potentially redundant data and wasting bandwidth resources.

Chapter 6

Implementation Details

Unico is implemented as a fully functional prototype file system with transparent access to a repository. This prototype is implemented in Linux, using Git 2.0 as the source platform for object manipulation. All file operations are implemented using FUSE callback functions. This chapter describes the most important implementation strategies used in Unico.

6.1 FUSE Integration

When Unico is run in a directory, it starts a FUSE process that will be responsible for all file operations in the local directory. This process can run in the foreground or background, depending on command line arguments. When the process starts, callback functions are registered for every supported file operation. These functions are called by the FUSE driver when the operations are performed in the mounted virtual file system.

Files available in the mounted directory before Unico starts are also included in file operations. As an optimization, to allow a user to start using the file system while these files are processed, the mount point is opened as a directory before FUSE is started, allowing for operations to be performed in the underlying directory even as Unico is mounted. As the file system runs, these files are silently transferred to the object repository in the background, and eventually included in

the next commit. Once there are no more files in the directory, it is no longer used by Unico for further operations.

Unico implements the following FUSE callback functions:

- Initialization/clean up (*init, destroy*);
- File input/output (*create, open, read, write, release, flush, fsync, truncate, ftruncate, fallocate*);
- File metadata management (*access, getattr, fgetattr, chmod, utimens*);
- Directory listing (*readdir*);
- Symbolic link management (*readlink, symlink*);
- Path manipulation (*mkdir, unlink, rmdir, rename*);
- Extended attribute management (*getxattr, listxattr, setxattr, removexattr*).

Since Unico is not directly backed by a block device, *bmap* and *statfs* are not implemented. Unico has no support for file ownership (*chown*), hard links (*link*) and file locking (*lock*). File nodes (*mknod*) are implemented using a direct tunnel to the mount-point directory, and are not included in commits or synchronized across peers.

Each operation will have its own tasks and responsibilities, but in general it will be implemented as follows:

1. The path is added to a Least Recently Used (LRU) list (described in Section 6.7);
2. If the path is available in the mount-point directory, tunnel the operation to the corresponding file in that directory and finish the process;
3. The path is compared against the list of invalid paths, revalidating the corresponding path if necessary, as detailed in Section 6.7;
4. If the path corresponds to a previous version (the `.snapshot` directory described in Section 5.2), the commit corresponding to that version is found, otherwise the most specific exception list entry applicable to the path is used;

5. The object corresponding to the full path is retrieved by examining the object identified in the previous step and following the remaining path entries;

6. Any objects required by the operation that are not available locally, including trees in the path leading to the requested entry, are retrieved from remote peers as needed;

7. If the operation involves a modification (e.g., *write, mkdir, rename*, etc.), new entries are added to the exception list depending to the operation;

8. If needed, the extended metadata corresponding to the last modified date and time is changed to the current time;

9. The function returns and any further processing is performed in the background.

A background thread is responsible for processing the file changes added to the exception list to reduce this list to a single entry. This thread will initially create objects for files and directories in the mount-point directory and save them into the exception list, and delete the files. Once the mount-point directory has no outstanding file left, this process will look for leaves in the exception list and apply them into new tree objects containing them. Once these new objects are created, they are saved in the same exception list and the list is pruned. This process is repeated until the exception list is again represented by a single tree object for the root directory. File updates while this process takes place are allowed, and are included in the same commit if they are completed before the exception list is pruned to a single entry, or on the next commit otherwise. Concurrency mechanisms are used to ensure that data is not corrupted or lost, as described in Section 6.2.1.

6.2 Creating and Merging Commits

Once the exception list is represented by a single root tree object, a new commit is created pointing to this new tree. At this point any outstanding commits received from peer nodes are also merged with the newly created commit, and the resulting commit is then sent to all peers. A more detailed description of how this commit is stored and sent to other nodes is described in Algorithm 6.1.

```
if exception list has any element besides the root tree entry then return;
Previous ←SHA-1 of current commit;
if root tree entry has different SHA-1 than root tree in Previous then
    set current commit to new commit with exception list entry as root tree,
    current commit as parent and current node as author;
end
foreach pending commit PC received from a peer do
    remove PC from pending commit list;
    Merged ← current commit;
    if there is no current commit then Merged ←PC;
    else if PC =current commit then do nothing;
    else
        Common ← common ancestors between PC and Merged;
        if Common contains only PC then do nothing;
        else if Common contains only current commit then Merged ←PC;
        else if current node should run merge then
            Merged ←RecursiveMerge(PC,current commit);
        end
    end
    if Merged ≠ current commit then
        if exception list has not been modified since algorithm started then
            trigger invalidation check (Algorithm 6.5);
            set current commit to Merged;
            change exception list root entry to root tree in Merged;
        else
            add Merged to pending commits;
            break loop;
        end
    end
end
if current commit different than Previous then
    Store commit reference in persistent storage;
    Send new commit to each known peer;
    Trigger prefetch process;
end
```

Algorithm 6.1: Commit replacement algorithm, triggered every time the exception list is pruned or a new commit is received from a peer.

When the commits being merged are not a descendent of each other, a recursive merge is required, as described in Section 5.4. This process is usually required after two partitioned nodes are able to connect to each other after operating individually for some time, or if two nodes perform separate updates at the same time, or other similar scenarios. Unico uses Git's recursive merge algorithm with a simplified tree comparison and merge algorithm, presented in Algorithms 6.2 and 6.3.

Since two or more nodes may receive the same set of commits to merge at a specific point, particularly in case of connection reestablishment, this merge might be triggered in multiple nodes. In order to reduce the number of merge commits, as well as the processing time require to process them, a simple negotiation method is used where one of the nodes is selected to perform the merge. This process uses a simple comparison of SHA-1 hashes for both the local and remote commits, and does not involve message exchanges beyond regular synchronization messages. This implementation does not guarantee that the merge will be performed in a single node, but it guarantees that at least one of nodes will perform it.

A recursive merge may fail if an object needed to perform it cannot be retrieved from connected nodes. If this happens, the commit is saved, and the merge process repeated every time a new commit is received from a peer. A newly established connection to a peer that may contain the missing object will trigger a new synchronization message, allowing the previously saved commit to be considered in addition to the commit received from the new peer.

Two different implementations are available for triggering a new scan for changes in the background commit thread. By default, as soon as any file operation changing the file system (such as a content change, file rename/delete or change of permissions) is completed, the background commit thread is signalled. This thread will then wait for a fixed amount of time, four seconds by default. This delay allows the system to group related changes and reduce the overhead of constant synchronization, as described in Section 5.3. After this delay, the thread will proceed with the process described above.

An alternative implementation is also available. In this implementation, the background thread will not receive any signals from file system operations, and will instead evaluate changes automatically in fixed periods (e.g., every one minute). This implementation is based on OriFS, and was created to allow a comparison be-

```
TreeMerge(H,M,C)
    input  : local tree object (H), remote tree object (M),
             common ancestor object (C)
    output: resulting tree object
    if H = M then return H;
    if C = M then return H;
    if C = H then return M;
    NewTree ← empty tree;
    foreach entry E in H or M do
        (SH, SM, SC) ← SHA-1 for E in (H, M, C), respectively;
        if E is in H, but not in M or C then add (E, SH) to NewTree;
        else if E is in M, but not in H or C then add (E, SM) to NewTree;
        else if E is in both H and M then
            Conflict ←false;
            if SH = SM then add (E, SH) to NewTree;
            else if E is in C then
                if SH = SC then add (E, SM) to NewTree;
                else if SM = SC then add (E, SH) to NewTree;
                else Conflict ←true;
            else Conflict ←true;
            if Conflict then
                if SH and SM are both trees then
                    SR ← TreeMerge(SH,SM,SC);
                    add (E, SR) to NewTree;
                else
                    add (E, SH) to NewTree;
                    append unique suffix to E;
                    add (E, SM) to NewTree;
                end
            end
        else if E is in H and C then
            if SH ≠ SC then add (E, SH) to NewTree;
        else if E is in M and C then
            if SM ≠ SC then add (E, SM) to NewTree;
        end
    end
    return NewTree;
```

Algorithm 6.2: Tree merge algorithm

```
RecursiveMerge(H,M)
    input  : local commit object (H), remote commit object (M)
    output: resulting commit object
    CL ← common ancestors between H and M;
    if CL is empty then C ← virtual commit with empty root tree;
    else C ← pop(CL);
    while CL is not empty do
        C ← RecursiveMerge(C,pop(CL));
    end
    RT ← TreeMerge(H,M,C);
    return commit created from tree RT and parents H and M;
```

Algorithm 6.3: Recursive merge algorithm

tween this approach and the one used by default in Unico [39]. Chapter 7 describes the results of this comparison in several scenarios.

6.2.1 Concurrency

The main objective of Unico is to provide a platform for data synchronization among devices for a single user. As such, although there is some support for collaborative scenarios, such as multiple devices accessing the same file simultaneously, this support is limited to handling conflicts after they are detected. No concurrency or mutual exclusion mechanism is used to provide inter-node locks.

Unico uses the pthread library implementation of mutex and read/write monitors, condition variables and semaphores to handle concurrency inside a single device [15, 35]. These mechanisms are used to control access to shared data structures, such as the exception list (described in Section 5.2), the LRU list used in the path invalidation process (described in Section 6.7) and the list of known peers. Threads also use condition variables to communicate with other threads. For example, a thread waiting for an object to be received will wait (with a time limit) on a condition variable, which will be signalled by the thread handling received data from a peer.

Whenever possible, portions of code that handle objects that may not be locally available are not protected by locks. This implementation decision seeks to reduce situations where unrelated processes need to wait for monitors locked by

other threads. One particular example where this is evident is in the process of merging incoming commits, described in Algorithm 6.1. In this algorithm, before the merging process starts, the monitor that protects the exception list and latest commit is released, allowing the user to access and modify the system while the merging takes place, as well as additional commits to be received. Once the process creates a merged commit, the lock is reaquired and exception list is compared to its previous version. If other changes have been performed while the merge was being computed, the merged commit is treated as a commit received from a peer, and the local changes result a new commit. After that, a new recursive merge takes place between these commits.

6.3 Communication

Each Unico node runs a TCP server socket listening for connections from other peers. The address of this socket can be discovered by other peers by one of three methods. Nodes may know the fixed address of another node. Additionally, nodes may also optionally use a shared directory, such as a directory in a cloud service, to share information about their address. Finally, once a connection between two nodes is established, both nodes transfer address information about their other peers to each other, as well as the address of the new connection to currently connected peers. Connection persistence is monitored using a simple TCP KEEPALIVE mechanism, and SSL is used for encryption. A Universal Plug-and-Play (UPnP) module allows nodes behind a Network Address Translation (NAT) router to be connected as well.

After the SSL connection is established, peers communicate using messages encoded as JSON objects. The following commands are used:

- HELLO: peers start communication with a HELLO message containing internal node identification and addresses for connection establishment.
- PEER: this command is used to transmit address data about other known peers. Once a connection is established between two nodes, each node will send to the newly connected node a PEER message for each of its known

peers, and send to each of its known peers a PEER message about the newly connected node.

- SYNC: this message is sent to inform other peers about the latest commit in the local node.
- OBJECT: this message transfers the contents of an object to another node. The contents themselves are sent outside the JSON message, both to reduce bandwidth for large objects and because JSON does not handle potentially binary objects properly.
- METADATA: this message sends the content metadata of an object to another node, as described in Section 6.4.
- STATE: this message sends the current state of an object to another node, including availability of content and metadata for the object both locally and in other peers where this information is known.
- REQOBJ, REQMETA, REQSTATE: this message requests that the contents, metadata or state (respectively) of an object be sent. If a node receives a REQOBJ or REQMETA message and it does have the requested data, an OBJECT or METADATA message (respectively) is sent back, otherwise a STATE message is sent instead. The response to a REQSTATE message is always a STATE message.
- INVALID: this message sends the local list of invalidated paths, as described in Section 6.7.

6.4 Metadata and Extended Attributes

As described in Section 5.6, there are three types of metadata. All metadata can be accessed by users using extended attributes. Each type of metadata, however, has a different process for computing and synchronizing information.

User-defined extended attributes, whose name starts with the prefix "`user.`", can be manually specified by a user or application using regular system calls such as `setxattr`. Other structural metadata, such as last modified date and time,

can also be obtained (but not modified directly) as extended attributes using the prefix "`system.stat.`". Both of these are saved in JSON files stored in a hidden directory. For a file called `/a/b/c/file.ext`, the attributes are stored in `/a/b/c/.unicoattr/file.ext.attr`. This file is synchronized using Unico's regular file synchronization process.

Content metadata are associated to objects instead of specific paths. Users and applications can access this data directly, even if the content is not available locally, by using extended attributes with prefix "`system.data.`". Since the contents of an object are immutable, it is assumed that content metadata are also immutable for the same object. Nodes that have access to the content will compute the metadata from the content itself. If the content is not available locally, the metadata can be requested from other nodes by using the command REQMETA described above. Once retrieved, the metadata are stored in the same directory as the object database, in a file named in the same way as the object content file itself, but with the suffix `.attr`.

System metadata, similar to content metadata, are also associated to an object, but are by their vary nature volatile. Users and applications can access this data through extended attributes with prefix "`system.unico.`". Among these attributes, "`system.unico.local`" indicates if the content and/or metadata of the object associated to the file is available locally, while "`system.unico.remote.XXX.state`", where *XXX* is the internal ID of a peer, contains the locally available information about if that peer has the content and/or metadata.

Table 6.1 describes a few examples of extended attributes made available to the user. A user can only directly change extended attributes with prefix "`user.`". Other attributes cannot be changed directly, but their value may be changed if the source of the corresponding information is modified (e.g., if the last modified date is changed by the system call *utime*, the extended attribute "`system.stat.mtime`" will have its value changed accordingly).

Each node stores, for each other node, any known information about availability of the content and of the metadata for the object, as well as the timestamp of when this information was retrieved. The collection of all this information in each node is named the *state* of the object. The timestamp (visible to the user through

Category	Examples

User-defined	`user.author`
	`user.version`
	`user.project`

Structural	`system.stat.ctime` `system.stat.mtime`

Content	`system.data.unico.hash`
	`system.data.unico.type`
	`system.data.exif.ColorSpace`
	`system.data.exif.DateTime`
	`system.data.exif.GPSLongitude`
	`system.data.exif.ExposureTime`

System	`system.unico.local`
	`system.unico.remote.c827013a.nodeid`
	`system.unico.remote.c827013a.received`
	`system.unico.remote.c827013a.state`

Table 6.1: Examples of extended attributes made available to users in Unico. Although user-defined extended attributes are also considered to be structural metadata, they are listed separately for clarity.

the attribute "`system.unico.`*XXX*`.received`", where *XXX* is the internal ID of the source node) is obtained from the source node itself (the node whose state is being computed). Since all object availability information related to a node is linked to the timestamp obtained from that node, Unico uses that timestamp as a version number to resolve any conflicts arising from synchronization. For example, if node A sends node B availability information about an object, and it includes the object state at A and another node C, the state of A will be linked to A's current timestamp, while the state in C will be sent with the timestamp received by A from C (or through a third party). Once B receives these states, it will compare the information about each node with its own information, and use, for each node, the state with the most recent timestamp.

Although a node may request the state of an object from peers, which usually happens in a state prefetching process described in Section 6.6, state information is usually transferred between nodes when a node responds with a failure to a request for object data. In other words, if a node A requests an object from node

B, and node B has no copy of this object, node B answers the request with its own local state info for that object. This strategy is used so that, when an object request cannot be fulfilled by a node, an update in the state information may contain additional data that allows the requestor to obtain the object from other sources.

6.5 Object Location

To simplify and guide the retrieval of object content when necessary, objects are associated to owner nodes. In any given node, the owners of an object are all connected peers (including potentially the local node itself) that are known to have the object content. This information is local to each node, and is based on system metadata transferred among nodes, as described in Section 6.4. As state information is exchanged and updated, the ownership of an object is updated to more closely resemble the most current information about availability. When an object has no known owner, a background process will send individual requests to all other peers to identify known owners.

When an object is needed by a node that does not store its content, Unico needs obtain it from one of its owners. In order to locate a possible source for it, the node uses an object location algorithm. This algorithm uses the state of the object in known peers to choose and rate connected nodes based on their probability of having the content and the expected latency to retrieve it. Once a suitable owner is selected, Unico requests the content from that node. If this node cannot produce the object (by timing out or returning its state, as described in the previous section), the system makes new requests to other owners in sequence until the object can be retrieved. If no suitable node can be identified with the available information, or all suitable nodes have been contacted with no success, Unico makes parallel requests to remaining connected nodes. For example, if node X has (possibly outdated) information that nodes A and B have the content of the required object, and has no information about nodes C and D regarding this object, it contacts A initially (assuming A is considered a best fit for producing the object, as compared to B). If A cannot produce it, the content is requested from B. If this request also fails, simultaneous requests are made to C and D for the content.

The selection of which suitable node should be contacted first is based on a set of criteria. The main factor used in this selection is the expected latency to retrieve the data from each node. This latency is estimated as a weighted average of recent Round Trip Time (RTT) values obtained from requests sent to each node. Another factor used in object selection is the timestamp linked to the object state in each node, described in Section 6.4. This heuristic is based on the assumption that the node that sent its state most recently is more likely to have its state unchanged since then. Although this timestamp is obtained from different nodes, Unico assumes that, although nodes may not have their clocks perfectly synchronized with each other, the timestamps should, for the purposes described here, be reasonably comparable. Even if the clocks are not synchronized, the effects of using unsynchronized timestamps in such a scenario are small, since the timestamp is used as a heuristic tie-breaker when latency is unknown.

The object location algorithm is used for both content and metadata retrieval, with the only differences being the command used to request data and the process of evaluating if the data has been retrieved. Algorithm 6.4 describes the object retrieval process in more detail.

6.6 Prefetching

Unico allows a user to specify paths that are always downloaded even when not specifically requested by a user. In order to identify which paths should be requested, Unico reads a configuration setting containing one or more path specifications. By default, no path is specified for prefetching, so only the root node itself is prefetched. If paths are specified, they follow this syntax, which is adapted from Git's syntax for pathspecs:

- Any specified path matches itself. If the path is matched to a directory, only the directory itself is fetched, not the files and subdirectories it contains.
- The path specification up to the last slash represents a directory prefix. The scope of that specification is limited to that subdirectory.

```
input  : the object to retrieve and the type of request (content or metadata)
output: success or failure
if object is available locally then return success;
P ← all connected peers;
while P ≠ ∅ do
    if object is available locally then return success;
    E ← any peer in P for which the request has been sent;
    if E exists then
        S ← most recently received state of object in E;
        if S was received from E after the process started then
            remove E from P;
        else
            wait until any object or state is received, or Timeout[E] has
            elapsed;
            if current time > Timeout[E] then remove E from P;
        end
    else
        L ← peers in P whose state indicate that object is available;
        if L ≠ ∅ then
            O ← peer in L with shortest RTT;
            send request for object to O;
            Timeout[O] ← current time +4× O's RTT;
        else
            foreach N in P do
                send request for object to N;
                Timeout[N] ← current time +4× N's RTT;
            end
        end
    end
end
return Failure;
```

Algorithm 6.4: Object retrieval process

- The rest of the specification is a pattern for the remainder of the path name. Paths relative to the directory prefix will be matched against that pattern using `fnmatch`; in particular, `*` and `?` can match directory separators.

A prefetching walkthrough is triggered every time a new commit is received and merged to the local commit. All tree objects required to reach a particular path specification are also retrieved.

In order to better inform the object location protocol with up-to-date information about availability of object content in neighbouring peers, the prefetching process also preemptively retrieves the state of potentially needed objects even when no prefetching rule is in place. In particular, when a commit or tree object is received, and it contains references to objects that are not available locally, and that have no known "owner" among connected peers, the prefetching process will, in the background, send a request for the state of that object to all known peers. This request is done with a small delay when the connection is idle, to avoid interference with more time-sensitive data transfer. This process also does not request the state of objects recursively, such as files in a subdirectory not stored locally. If, for example, the tree corresponding to the path `/a/b/c` is received, and it has a reference to a subdirectory `d` whose tree object is not available locally, the state of `d` is preemptively requested from all neighbouring nodes, but not its content or the state of the files and directories contained in it (unless `/a/b/c/d` matches one of the prefetching path specifications described above).

6.7 Path Invalidation

Unico implements a mechanism to reduce the frequency (and by consequence the resource use) of synchronization messages in idle nodes, or nodes that are not actively using files being modified by other peers. The system tracks each file access using an LRU list, implemented with a combination of a queue and a hash table. Entries in this LRU are removed after a configurable amount of time or after the LRU reaches a limit size.

Algorithm 6.5 describes the algorithm used when a new commit is received from another peer, with respect to invalidation. This algorithm is executed automatically after the commit has been merged into the local version of the repository.

```
input: objects corresponding to old (O) and new (N) commits
input: path (P), initially the root directory
if invalidation is disabled then return;
if some revalidation is in process then return;
if P is in the invalidation list then return;
if O = N then return;
if P is not in the LRU then
    add P to the invalidation list;
    trigger a broadcast of the invalidation list to all peers;
else if O or N are not trees or commits then return;
else
    foreach entry E in O or N do
        if E is in O and not in N then do nothing;
        else if E is in N and not in O then
            add (P,E) to the invalidation list;
            trigger a broadcast of the invalidation list to all peers;
        else
            call recursively on object for E in N and O and path (P,E);
        end
    end
end
```

Algorithm 6.5: Invalidation process after a new commit is received

Algorithm 6.6 describes the revalidation process that is executed before any file operation in Unico.

6.7.1 Invalidating Idle Paths

A node adds a path to the invalid path list based on changes received from peers. When a node receives a new commit from a peer, a background process will compare the root directory of this commit with the same directory in the local commit. Any entry that has been modified in the received commit and is not found in the LRU list, i.e., the entry has not been visited in the local node in the recent past, is considered invalid and added to the invalid path list.

The comparison between a received commit and the local commit is performed in a top-down recursive approach. If two entries corresponding to subdirectories

```
input: path being visited
if invalidation is disabled then return;
add path P to the LRU;
if P is not in the invalid path list then return;
remove P from the list of invalid paths;
broadcast list of invalid paths to all connected peers;
add P to list of paths waiting for revalidation;
while P is in list of paths waiting for revalidation do
    wait for commit C from any peer N, or timeout;
    if no commit was received then break loop;
    foreach S in list of paths waiting for revalidation do
        if S is not invalid in N then
            remove S from the list of paths waiting for revalidation;
        end
    end
end
```

Algorithm 6.6: Revalidation process before FUSE file operations

in the root directory do not match, and are found in the LRU list, i.e., the entry has been visited recently in the local node, the directories are recursively compared again, allowing for invalidation in a more detailed path.

For example, a node has recently visited files `/foo` and `/bar/baz/qux`, and no other file. In this case, the LRU will contain entries for both files, as well as for the directories that contain them (`/`, `/bar` and `/bar/baz`). This node receives a new commit that has changes in files `/foo`, `/nor/biz` and `/bar/boo/far`. Once this commit is processed and merged, Algorithm 6.5 is called on the root directory of the repository. This algorithm will identify a change in `/foo`, but since this file is in the LRU it will not be invalidated. The change in `/nor` will also be detected, and since it is not in the LRU this path will be added to the invalid path list.

Finally, the change in `/bar` will be identified by the algorithm, and since it is in the LRU and is a directory, this entry will be recursively compared against the corresponding directory in the local entry. A change in `/bar/boo` is identified, and since it is not in the LRU this path will be invalidated.

After Algorithm 6.5 completes, if the list of invalid paths was changed it is sent to all peers. At this point, no further commits are sent to the node with invalid paths until the paths are revalidated, unless the commit includes changes that are not covered by the invalid path list. In the example above, the invalid path list comprises `/nor` and `/bar/boo`. Other nodes will use this information and compare any commits they have to send against this list. If a modification in this commit includes files that are not in the invalid path list, it is sent to the node. If, however, the commit only changed files within those paths, the commit is withheld.

6.7.2 Revalidating Invalid Paths

Nodes that have a list of invalid paths must take into account that any local information about those paths is potentially out-of-date, since other nodes have stopped sending commits associated to those files. Once these files are accessed in the node with invalid paths, the system must ensure that a recent version of the file is served. To achieve that, all file system operations compare the file or directory where the operation is taking place against the invalid path list, as described in Section 6.1. If the path is marked as invalid, the node needs to retrieve the latest version of the directory or file before serving it to the user.

Once a file in an invalid path is visited (a file operation is performed on it), Unico removes the offending entry in the invalid list, and then broadcasts the pruned invalid path list to all nodes. The operation is then halted until a new commit is received or a configurable timeout is reached. Other nodes, upon receiving a new invalid path list, will respond with their own latest commit. Upon receiving a new commit message, the node that sent the new invalid path list will merge the received commit, as usual. Once this merge is successfully applied, the file operation is resumed and the file is served to the user.

Since other peers may also make use of the invalidation mechanism, the revalidation process must make sure that, when a new commit is received, the path that needs to be served to the user is also not marked as invalid in the peer that sent that commit. If that is the case, the revalidation process will not serve the file to the user, and will instead wait for another commit. Once a peer that does not have

the affected path in its invalidation path sends a synchronization message, the file is considered up-to-date.

For example, node A has path `/bar/boo` in its invalid path list, and is connected to nodes B and C. Node B has an invalid path list consisting of `/bar`, while node C has no invalid path. If the user in node A reads file `/bar/boo/far`, node A will identify that this file is in an invalid path. The entry `/bar/boo` is removed from the invalid path list, and the new invalid path list is broadcast to all connected peers. At this point the user operation that triggered the invalidation is suspended until a current version can be served to the user.

Upon receiving the new invalid path list from node A, node B will send its commit to node A, where it will be merged. However, since node B's invalid path includes `/bar`, which corresponds to the file being served to the user, the file operation is not resumed. After node C sends a new commit to node A, it will also be merged, but since node C does not have an invalid path related to `/bar/boo/far` it will trigger the operation on this file to be resumed, and the file will be served to the user.

An alternative approach was evaluated, where the node revalidating the path would request the visited file directly instead of a new commit. This approach, however, would incur in additional complexity in managing the state of the data after the revalidation. First, other files revalidated after this process would need to be evaluated to assure that a user would have access to their latest version. In addition to that, revalidated files changed right after this process would need to be merged in a more complex process. So, even though waiting for a new commit may incur in additional processing time, the fact that the state after revalidation is more stable was a deciding factor in choosing the current approach.

6.7.3 Example

Figure 6.1 shows a diagram with an example of how path invalidation behaves in a simple case. This example assumes that files in the `x` directory are initially idle in node B. When it is modified in node A, the modification triggers a synchronization process, which sends a new commit to node B. Upon receiving this commit, node B will identify what file was modified, and as `x` is not in the LRU in this node, will

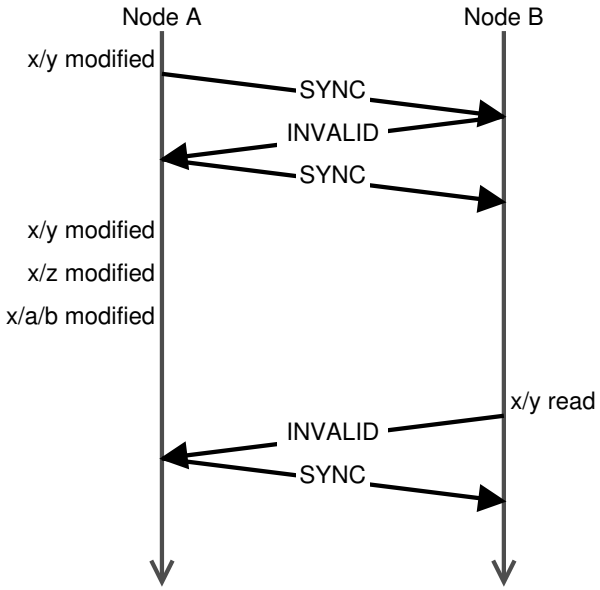


Figure 6.1: Event-response diagram for an example of path invalidation and revalidation for two nodes.

mark this path as invalid, and send this information back to node A. At this point, any access to a file in this directory in node B will trigger a revalidation. Once node A receives the new invalid path list from node B, it will make note of this list, and any modification on a file in the `x` directory will not trigger any synchronization.

After some time has elapsed in the example scenario, node B will read file `x/y`. Since this file is in the invalid path list, this action will trigger this list to be pruned. The new list is broadcast to peers, and once node A receives it, a new synchronization message is received, which allows node B to serve the latest version of the file to the user.

6.7.4 Considerations

In order to keep nodes with a reasonably recent view of the file system even when invalidation is in place, and minimize the impact of a network disconnection, invalid paths can be configured to expire after some time (e.g., thirty minutes). After this time, entries in the invalid path list are removed, even if the file is not visited, triggering a list broadcast and, by consequence, new synchronization messages

from peers. This synchronization message is still subject to the same invalidation checks as before, so the expired entries in the invalid path list may be added again if they are still being changed in other peers and not visited in the local node.

The prefetching process does not interfere with or directly affect path invalidation, but is indirectly influenced by it. Since path invalidation reduces the frequency of commit synchronization, when a remote node changes files in a path that has been invalidated in the local node, the corresponding commit will not be transferred to the local node, and so prefetching will not be triggered. Conversely, prefetching does not affect the LRU list that keeps track of visited files, so a node with prefetching rules will still be able to invalidate a path if it is not visited through the file system.

6.8 Code Implementation

As described above, Unico is implemented on top of Git, using version 2.0.0 as the basis for development. All features are implemented in C.

Git's code uses the notion of builtin commands. When the `git` executable file is run, the first non-option argument is used as a command, and the function corresponding to that command is called with the remaining arguments. The compilation routine for Git also creates links (hard or symbolic links, depending on file system capabilities) for the `git` executable file, corresponding to each builtin function. For example, a link file called `git-commit` is created, and when executed it runs the equivalent of calling `git` with `commit` as its first argument.

Unico creates a new builtin function called `git-fs`. When called, this function mounts the file system in the current directory (as detailed in Section 6.1). This function accepts most of the arguments usually associated with FUSE file systems, such as:

- `-f`: run in foreground;
- `-s`: run file operations in a single thread;
- `-d`: print debugging information on file operations received from the kernel (implies `-f`).

The code for the builtin function and other file system operations, including the exception list, background commit process, invalidation and management of extended attributes, comprises 4229 lines. Code for the synchronization module and prefetching functions adds up to 2178 lines. Additional changes were made in Git's own files, totaling around 300 lines added or changed in 20 files.

Chapter 7

Experimental Results

This chapter describes the results of several experiments that were performed to evaluate how Unico handles different scenarios, profiles and configuration settings.

7.1 Testing Environment

In order to evaluate Unico, particularly the efficiency of storage and bandwidth utilization, we created a testbed with six virtual machines distributed across two different hosts in two different subnets. All nodes run Linux and are presented in Table 7.1. Except when otherwise noted, the scenarios described below run in a subset of four or these nodes, namely nodes B, F, M and N.

Host	Operating System	Network
Node B	64-bit Debian 8.2	globally-reachable IP
Node C	64-bit Mint 17	NAT
Node D	64-bit Mint 17	NAT
Node F	64-bit Ubuntu 14.04.3	globally-reachable IP
Node M	64-bit Ubuntu 14.04.3	globally-reachable IP
Node N	32-bit Ubuntu 14.04.3	globally-reachable IP

Table 7.1: List of nodes used in the experiments.

7.1.1 Profiles

All the scenarios are tested using different sets of prefetching policies, in order to evaluate the impact of each policy in the use of bandwidth, storage and access latency. Each node may be set to a push-all profile (by using a prefetching rule of "$\star$"), where all content is preemptively requested after every modification, or an on demand profile (by using no prefetching rule at all), where content is requested only when needed. In total six profiles are tested:

- *PRE-IMM*: Full prefetch with immediate commits. In this profile, commits are created and synchronized one second after a change, including all content. This profile is similar to that of systems like Dropbox [16] and BitTorrent Sync [5].
- *PRE-PER*: Full prefetch with periodic commits. In this profile, commits are created and synchronized automatically every 60 seconds (unless specified otherwise), including all content. This profile is similar to that of systems like OriFS [39].
- *PRE-INV*: Full prefetch with path invalidation. In this profile, commits are created automatically one second after a change, but commits and content are only synchronized to nodes that use the associated files.
- *OD-IMM*: On demand content retrieval with immediate commits. In this profile, commits are created and synchronized one second after a change, but content is retrieved on demand. This is the default approach used by Unico if path invalidation is not enabled.
- *OD-PER*: On demand content retrieval with periodic commits. In this profile, commits are created and synchronized automatically every 60 seconds (unless specified otherwise), but content is retrieved on demand.
- *OD-INV*: On demand content retrieval with path invalidation. In this profile, commits are created automatically one second after a change, but commits are only synchronized to nodes that use the associated files, and content is retrieved on demand. This approach allows the evaluation of path invalidation in Unico.

7.1.2 File Operations

Except when otherwise noted, the file operations used in these experiments are defined as:

- *Content read*: a read-only operation using the content of the file itself. To evaluate this operation, a read will consist of calling the `file` command, a program in Linux that reads the content of the file and describes its content type (e.g., image, text, etc.) based on predetermined rules.
- *Metadata read*: a read-only operation using only the extended attributes of the file. This operation will consist of calling the `getfattr` command, a program in Linux that reads and prints extended attributes of a file. The command is run with arguments `-d` (dump values of all attributes) and `-msystem.data` (list only attributes with the provided prefix, which corresponds to attributes associated to content metadata).
- *File change*: an operation that changes the content of a file. Unless otherwise noted, the change will not alter the size of the file itself, to allow the experiments to evaluate bandwidth and storage resource utilization with limited variability for external causes. This operation will usually consist of a `dd` script that replaces 256 bytes in a random position of the file with a random string of the same size. The executed command is (where FILE is the modified file and POS is a random number between zero and the file size minus 256):

```
dd if=/dev/urandom of=FILE bs=1 seek=POS \
  count=256 conv=notrunc status=none
```

These operations are performed in individual files. If a scenario describes an operation being performed on a set of files (e.g., files in a subdirectory), by default it will be performed on a randomly chosen file in the set. If the operation is repeated more than once, a different file is chosen for each new operation run. For example, if an experiment describes that, at a specific frequency, 10 file changes are performed, in practice 10 different files are randomly chosen from the specified

set, and the file operation is performed for each of these files in sequence (i.e., no parallel changes).

7.1.3 Metrics

Except when otherwise noted, the measured metrics in these experiments are:

- *Data transfer*: a measure of network resource utilization, this metric corresponds to the total amount of data transferred by the application in each node, in KB (kilobytes). This metric is usually listed as the sum of two values: content (the amount of transferred data that corresponds to file content itself) and overhead (any other type of transferred data, including metadata, requests, synchronization messages and peer discovery). These two values are also usually divided in incoming and outgoing traffic for each node.

- *Storage increase*: a measure of storage resource utilization, this metric corresponds to the difference between the amount of data stored in the object database after the execution of a scenario, and the same value before this execution, in KB (kilobytes).

- *Access latency*: this metric computes the amount of time required to execute a specific operation, in seconds. When the operation is not explicitly described, it corresponds to the time required to read a file or its metadata, as described above. Only the lowest 70% of latency values for each metric are considered for the average, to reduce the impact of variability related to network load in each scenario.

- *Conflict rate*: this metric measures the incidence of file (or metadata) reads that correspond to an older version of the file. The metric is quantified by the time elapsed since the version currently used by the consumer was last modified. For example, if the consumer has knowledge of a specific version of a file locally, but other nodes have already modified this file seven seconds ago, this metric will be defined as 7. This metric is measured as 0 (zero) if the consumer accesses the latest version of the file.

Profile	Full Content	Metadata
PRE-IMM	0.010	0.009
PRE-PER	0.010	0.008
PRE-INV	1.145	1.115
OD-IMM	0.040	0.015
OD-PER	0.010	0.008
OD-INV	1.110	1.110

Table 7.2: Comparison of access latency (in seconds) between content and metadata retrieval in a single-file producer/consumer experiment.

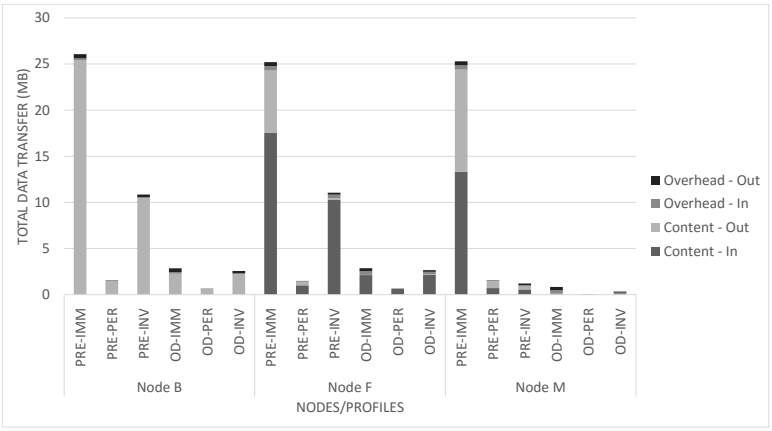
Profile	Full Content	Metadata
PRE-IMM	1.2	1.2
PRE-PER	25.2	26.9
PRE-INV	0.2	0.1
OD-IMM	1.1	1.2
OD-PER	24.6	24.6
OD-INV	0.0	0.2

Table 7.3: Comparison of conflict rate (as defined in Section 7.1.3) between content and metadata retrieval in a single-file producer/consumer experiment.

7.2 Producer/Consumer with One File

In a first scenario, a single node (producer) updates a file continuously (every three seconds), sending updates to all other nodes based on each profile. Another node (consumer) reads the file that has been updated by the node every 15 seconds, causing it to be transferred if needed. In a variation of this scenario, instead of the entire file, the consumer will read all extended attributes. We compare the latency of file access and conflict rate in the consumer and the use of bandwidth and storage in cases where all nodes use each of the profiles listed above. For this experiment, node B was used as the producer, while node F was the consumer. In total, this scenario was run with twelve variations, one for each profile with full file content retrieval, and one with extended attribute listing.

This section discusses the results of these experiments, while measurements are presented as follows. Figures 7.1 and 7.2 compare network and storage utilization



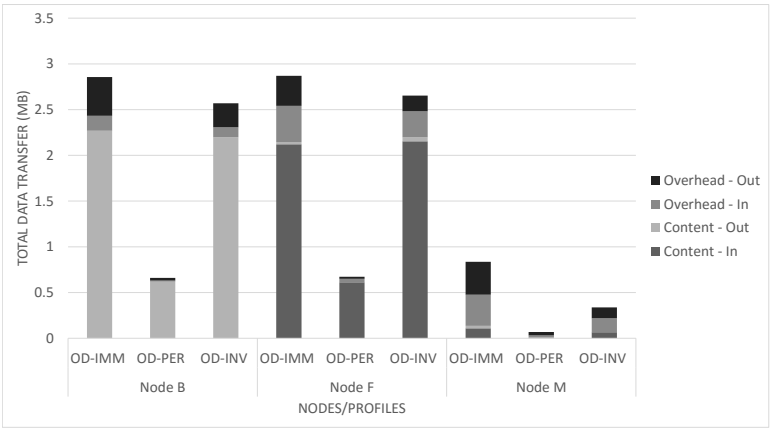


Figure 7.1: Total amount of data transfer for a single-file producer/consumer experiment for full content access. The top graph shows all profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

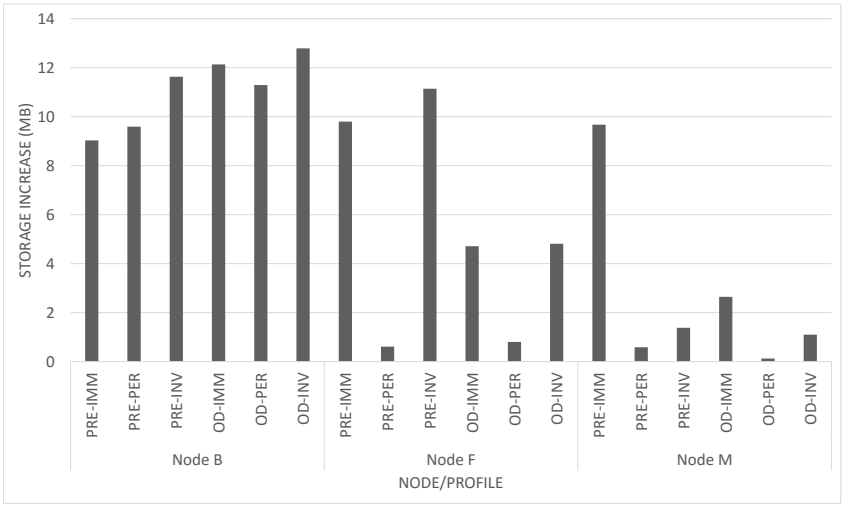


Figure 7.2: Storage resource utilization for a single-file producer/consumer experiment.

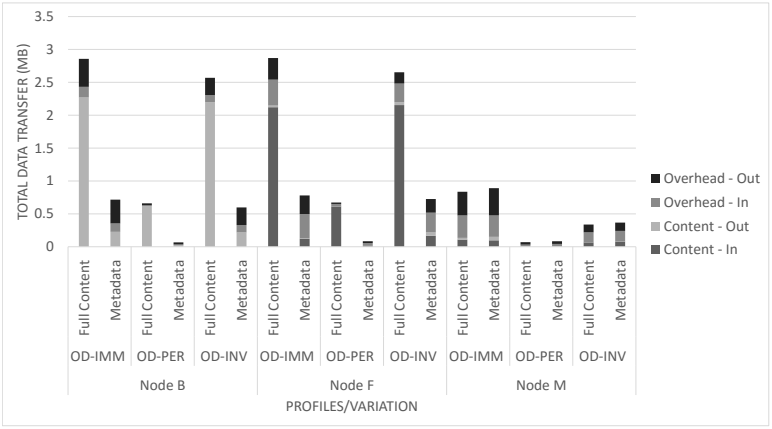


Figure 7.3: Comparison of total amount of data transfer between content and metadata retrieval for a single-file producer/consumer experiment.

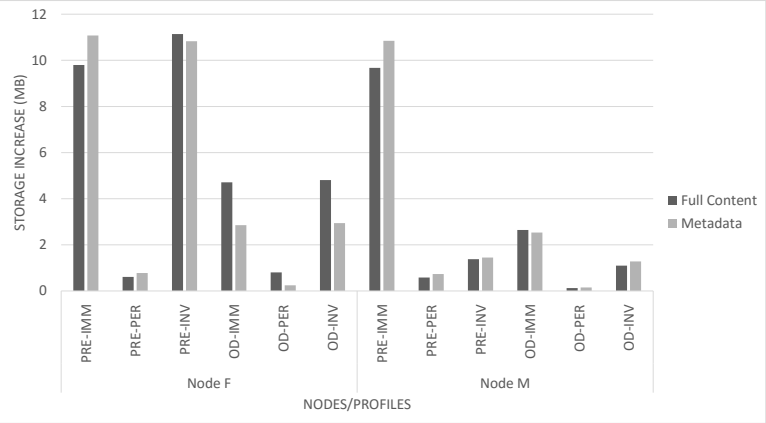


Figure 7.4: Comparison of storage resource utilization in the consumer between content and metadata retrieval for a single-file producer/consumer experiment.

between profiles when full file content is retrieved in a read. Figures 7.3 and 7.4 compare the same metrics between full content reading and extended attribute listing in the consumer node. The data that generated these graphs is presented in Appendix A, in Tables A.1 to A.3. Access latency is described in Table 7.2, while conflict rate is presented in Table 7.3. Node B produces data, while node F consumes it. Nodes M and N are idle. To aid the visualization of other nodes' data, node N is not shown in some graphs, as results are similar to node M.

As shown in Tables 7.2 and 7.3, a fully synchronized approach (PRE-IMM) accounts for shorter access latency and low conflict rate, as expected, since the file content is usually available in the consumer node when the read operation happens. However, this comes at the expense of significantly higher network and storage resource utilization, as can be seen in Figures 7.1 and 7.2. When a periodic synchronization is used, however (PRE-PER), network and storage resources are saved, with a 94% reduction both in data transfer and storage increase in the consumer, at the expense of a significant increase in conflict rate.

The same figures and tables show that, when an on demand approach is used, an equivalent reduction of network resources is observed, with 89% less data being transferred in the consumer when comparing the profile OD-IMM with PRE-IMM in Figure 7.1. Storage resources are also reduced by 52% as compared to the equivalent profile using full prefetching, as can be seen in Figure 7.2. Although latency is affected, the conflict rate remains low.

Path invalidation resulted in a reduction in conflict rate, as the consumer is forced to revalidate the consumed object and as such gets access to the latest version of the file. This advantage, though, comes at the expense of higher access latency both for full prefetching (PRE-INV) and on demand data transfer (OD-INV). Since the consumer accesses files that are often marked as invalid, a revalidation is needed in most file reads, and as such the latency is increased by the need to synchronize and merge the latest version of the file.

Invalidation resulted in a small decrease of data transfer and storage in the consumer node in case of on demand object retrieval. The decrease is small, though, due to the need to recursively compare paths in all received commits, which requires the node to transfer changed tree objects and evaluate these changes locally. The results, however, are significantly better in idle nodes, where invalidation results in a significant reduction in network and storage resource use. This can be seen in Figures 7.1 and 7.2 by comparing scenarios OD-IMM and OD-INV for nodes F (consumer) and M (idle).

Since node B is the producer, and as such had access to the latest version of the affected file at all times irrespective of the profile in use, its storage and incoming network utilization was not significantly affected by changes in its own profile; however its outgoing network utilization peaked when other nodes prefetched its content, changing from around 2.9MB of data transfer when all nodes used on demand fetching to around 26MB when full prefetching was in place.

In this scenario, nodes M and N are not directly involved in any file read or modification. However, the use of different profiles can affect the use of resources in M and N as well. In a full prefetching profile, these nodes incur a significant amount of data transfer, comparable to a consumer node, as can be seen in Figure 7.1. In fact, node M transferred around 25MB of data on a full prefetch profile, compared to 836KB when on demand was in place. Path invalidation also im-

Profile	Full Content	Metadata
PRE-IMM	0.012	0.011
PRE-PER	0.012	0.010
PRE-INV	0.290	0.333
OD-IMM	0.069	0.065
OD-PER	0.101	0.056
OD-INV	0.376	0.317

Table 7.4: Comparison of access latency (in seconds) between content and metadata retrieval in a multiple-file producer/consumer experiment.

proved the use of resources in these nodes, by further reducing data transfer in more than half.

As expected, the use of metadata instead of the file content in prefetching profiles did not significantly affect the results, since the content is retrieved anyway by all nodes, irrespective of how the data is accessed by the consumer. However, Table 7.2 shows a slight improvement in access latency, since retrieving extended attributes is generally faster than reading the file content. Furthermore, in on demand profiles, the use of metadata in the consumer significantly reduced the use of network and storage resources and the latency, as shown in Figures 7.3 and 7.4. Since nodes are not required to obtain and store the file content, the use of extended attributes allows certain application to retrieve the latest version of the information it needs faster and with reduced resource waste in any profile.

7.3 Producer/Consumer with Multiple Files

A new scenario was created as a variation of the first scenario. In this scenario, a directory with ten files is used for data producing and consuming. As in the first scenario, a single node (producer) updates all ten files in the directory continuously (every five seconds), sending updates to all other nodes based on each profile. Another node (consumer) reads two random files in the same set, causing it to be transferred if needed. The same metrics are gathered, with nodes B and F reprising their roles as producer and consumer, respectively. In total, twelve variations of this scenario were run: each profile ran once with full content retrieval, and once with extended attribute reading.

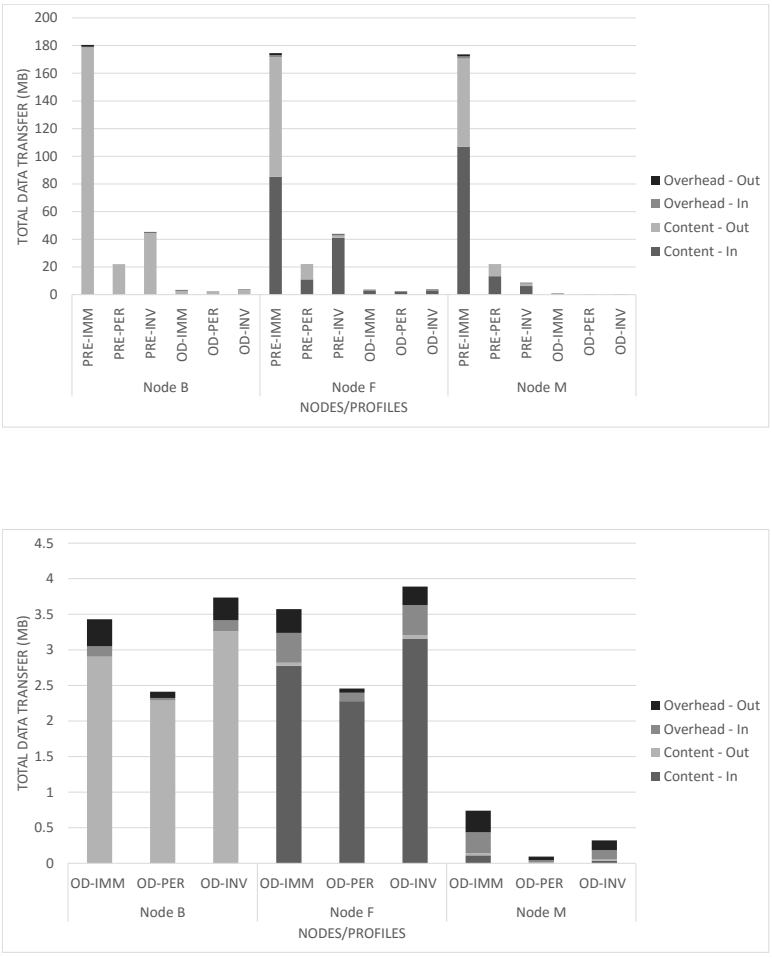


Figure 7.5: Total amount of data transfer for a multiple-file producer/consumer experiment for full content access. The top graph shows all profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

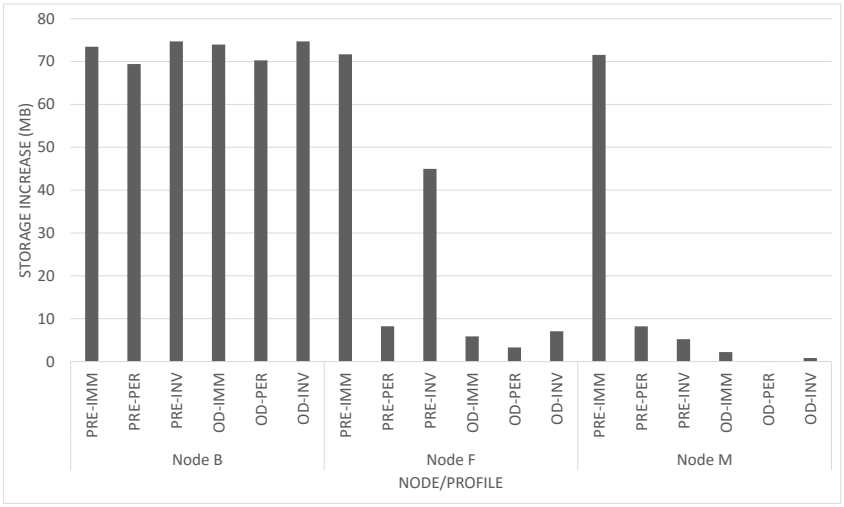


Figure 7.6: Storage resource utilization for a multiple-file producer/consumer experiment.

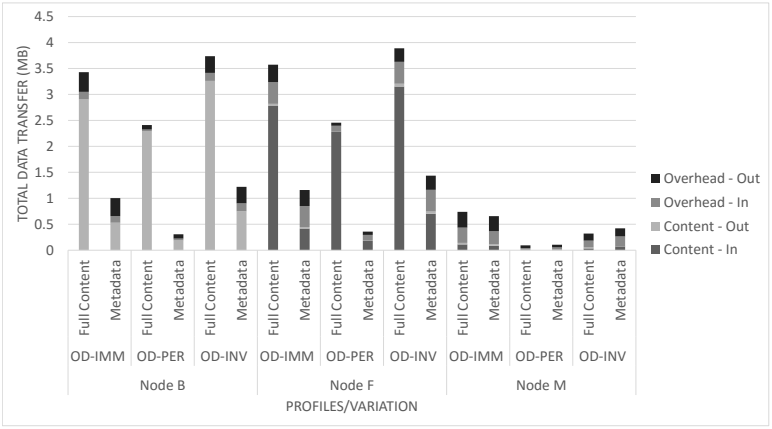


Figure 7.7: Comparison of total amount of data transfer between content and metadata retrieval for a multiple-file producer/consumer experiment.

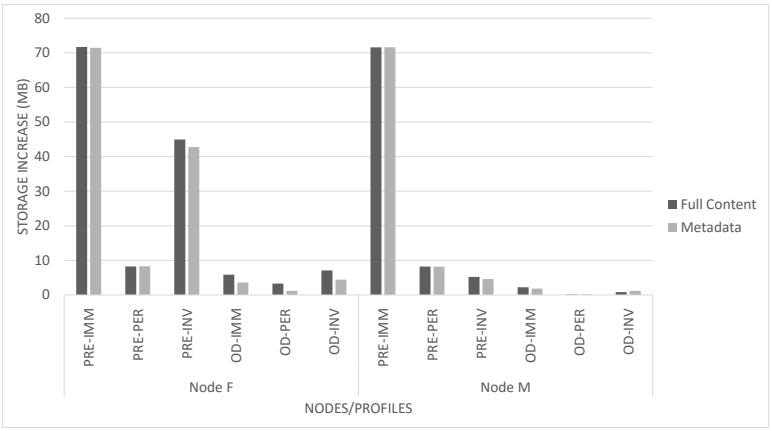


Figure 7.8: Comparison of storage resource utilization in the consumer between content and metadata retrieval for a multiple-file producer/consumer experiment.

Profile	Full Content	Metadata
PRE-IMM	0.6	0.0
PRE-PER	27.3	35.4
PRE-INV	0.0	0.0
OD-IMM	0.0	0.0
OD-PER	24.9	27.0
OD-INV	0.0	0.0

Table 7.5: Comparison of conflict rate (as defined in Section 7.1.3) between content and metadata retrieval in a multiple-file producer/consumer experiment.

This section discusses the results of these experiments, while measurements are presented as follows. Figures 7.5 and 7.6 compare network and storage utilization between profiles when full file content is retrieved in a read. Figures 7.7 and 7.8 compare the same metrics between full content reading and extended attribute listing in the consumer node. The data that generated these graphs is presented in Appendix A, in Tables A.4 to A.6. Access latency is described in Table 7.4, while conflict rate is presented in Table 7.5. Node B produces data, while node F con-

sumes it. Nodes M and N are idle. To aid the visualization of other nodes' data, node N is not shown in some graphs, as results are similar to node M.

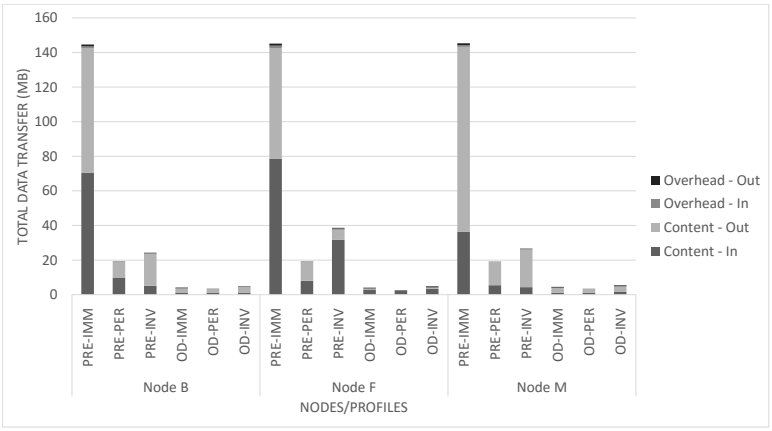
As expected, the results of this scenario are equivalent in most of the metrics to those of the first scenario. It becomes more evident in this case that on demand synchronization drastically reduces the use of network resources, as can be seen in Figure 7.5. Data transfer is reduced by 98% both in the producer and the consumer, when comparing the immediate synchronization profiles for full prefetching (PRE-IMM) and on demand (OD-IMM).

These results also highlight how path invalidation achieves one of its main objectives: when this feature is turned on, idle nodes do not waste resources on those files, even when full prefetching is in place. In node M, for example, as shown in Figures 7.5 and 7.6, invalidation accounted for a 96% reduction in data transfer and 94% reduction in storage increase for full prefetching profiles, namely profile PRE-INV when compared to PRE-IMM. Resources are also more efficiently used in on demand profiles (OD-INV compared to OD-IMM), although the reduction is not as drastic.

7.4 Producer/Consumer with Switching Devices

In order to evaluate how the system behaves when a user changes devices, a new scenario was created where two separate producers change the same set of files. In this scenario, one producer modifies the files for the first half of the experiment; after a while, the first producer stops modifying files, and a second producer takes over, modifying the same set of files. The file set is the same as the one used in the scenario presented in Section 7.3. Nodes B and M are the producer for the first and second half of the experiment, respectively, while node F is again the consumer. The same six profiles and two variations (full content and metadata retrieval) are repeated, with the same metrics being evaluated.

This section discusses the results of these experiments, while measurements are presented as follows. Figures 7.9 and 7.10 compare network and storage utilization between profiles when full file content is retrieved in a read. Figures 7.11 and 7.12 compare the same metrics between full content reading and extended attribute listing in the consumer node. The data that generated these graphs is presented in



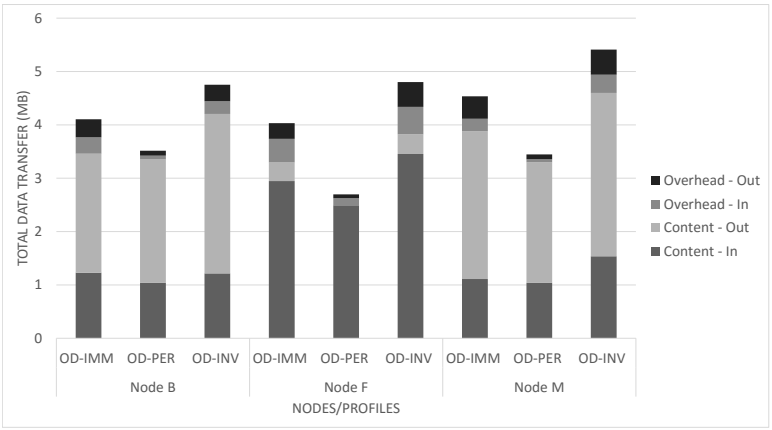


Figure 7.9: Total amount of data transfer for a producer/consumer experiment with switching devices for full content access. The top graph shows all profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

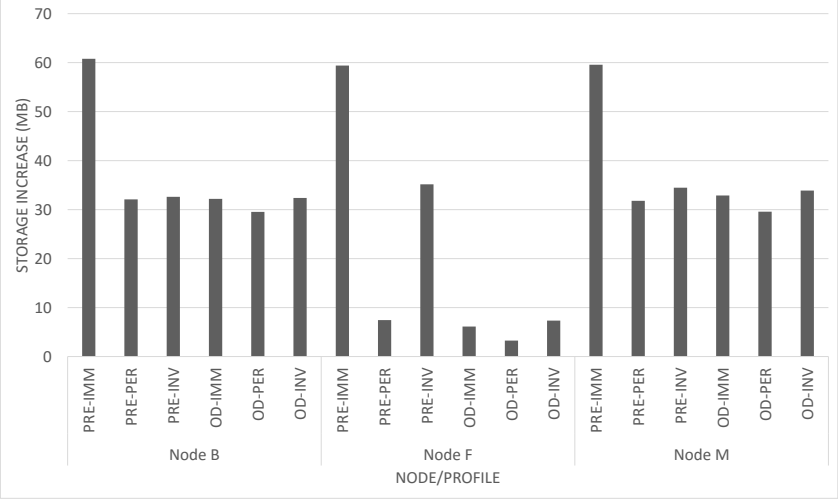


Figure 7.10: Storage resource utilization for a producer/consumer experiment with switching devices.

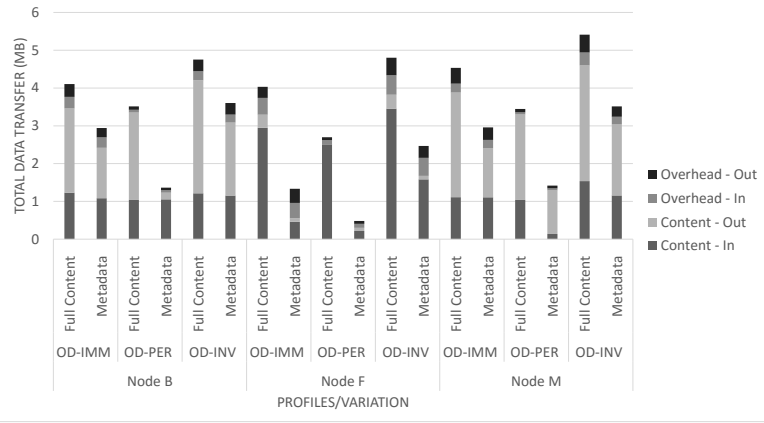


Figure 7.11: Comparison of total amount of data transfer between content and metadata retrieval for a producer/consumer experiment with switching devices.

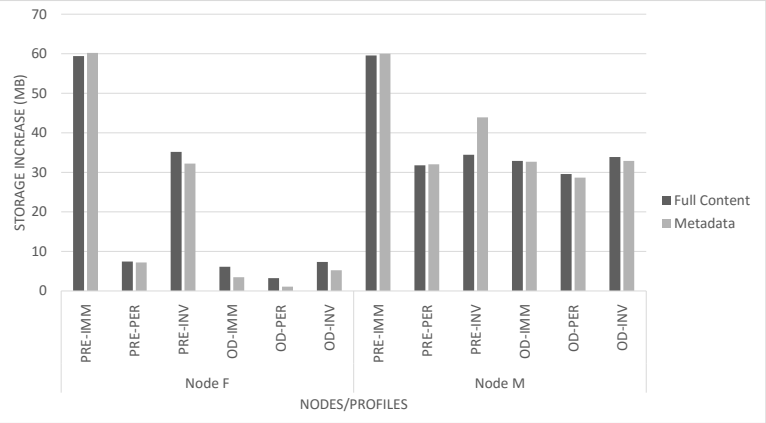


Figure 7.12: Comparison of storage resource utilization in the consumer between content and metadata retrieval for a producer/consumer experiment with switching devices.

Profile	Full Content	Metadata
PRE-IMM	0.011	0.010
PRE-PER	0.011	0.010
PRE-INV	0.232	0.092
OD-IMM	0.076	0.060
OD-PER	0.047	0.040
OD-INV	0.208	0.278

Table 7.6: Comparison of access latency (in seconds) between content and metadata retrieval in a producer/consumer experiment with switching devices.

Appendix A, in Tables A.7 to A.9. Access latency is described in Table 7.6, while conflict rate is presented in Table 7.7. Nodes B and M produce data, while node F consumes it. Node N is idle. To aid the visualization of other nodes' data, node N is not shown in some graphs.

Since node M is no longer idle in this scenario, its use of network and storage resources is larger than that found for the same node in Section 7.3 as can be observed in Figures 7.9 and 7.10. Conversely, since node B is no longer active for

Profile	Full Content	Metadata
PRE-IMM	0.1	0.2
PRE-PER	16.3	105.4
PRE-INV	0.0	32.5
OD-IMM	0.2	0.0
OD-PER	17.7	102.3
OD-INV	0.3	0.0

Table 7.7: Comparison of conflict rate (as defined in Section 7.1.3) between content and metadata retrieval in a producer/consumer experiment with switching devices.

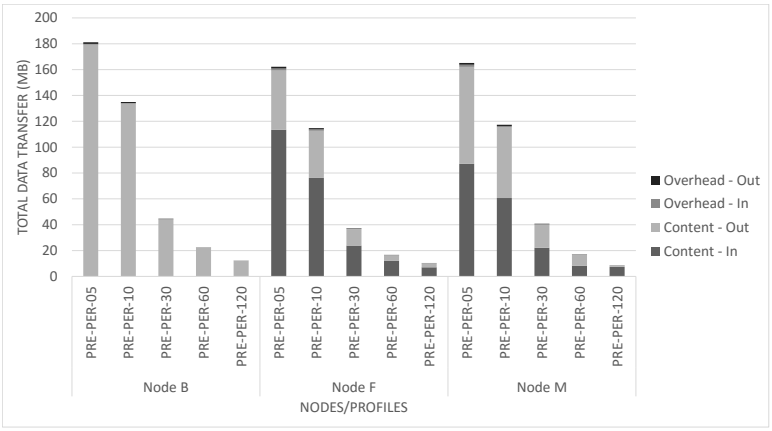
the entire scenario's execution time, its use of resources is reduced. This behaviour, though, is not found in the full prefetching profile (PRE-IMM). Since all data is transferred to all nodes in this case, nodes that become idle will continue to make use of network and storage resources as long as the repository is modified.

Tables 7.6 and 7.7 demonstrate that, although the source of up-to-date information has been changed, the consumer is still able to obtain the data required to read its files with the same latency and conflict rates as those found in Section 7.3, described in Tables 7.4 and 7.5.

7.5 Evaluation of Synchronization Frequency

In order to evaluate how different synchronization frequencies affect resource utilization and conflicts, the scenario presented in Section 7.3 was repeated with different values for the periodic profiles, both with prefetch and on demand retrieval. The rate of commit creation was set to 5, 10, 30, 60 and 120 seconds, and the same set of operations as scenario 2 was performed. The profiles PRE-PER and OD-PER, described in Section 7.1.1, are adapted into ten profiles, namely PRE-PER-*NN* and OD-PER-*NN*, where *NN* is the rate of commit creation, in seconds.

This section discusses the results of these experiments, while measurements are presented as follows. Figures 7.13 and 7.14 compare network and storage utilization between profiles when full file content is retrieved in a read. Figures 7.15 and 7.16 compare the same metrics between full content reading and extended attribute listing in the consumer node. The data that generated these graphs is



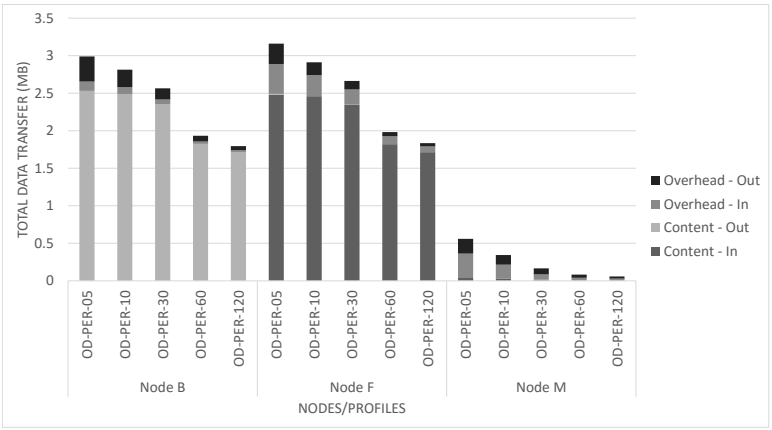


Figure 7.13: Total amount of data transfer for an evaluation of synchronization frequency for periodic commits for full content access. The top graph shows only prefetch profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

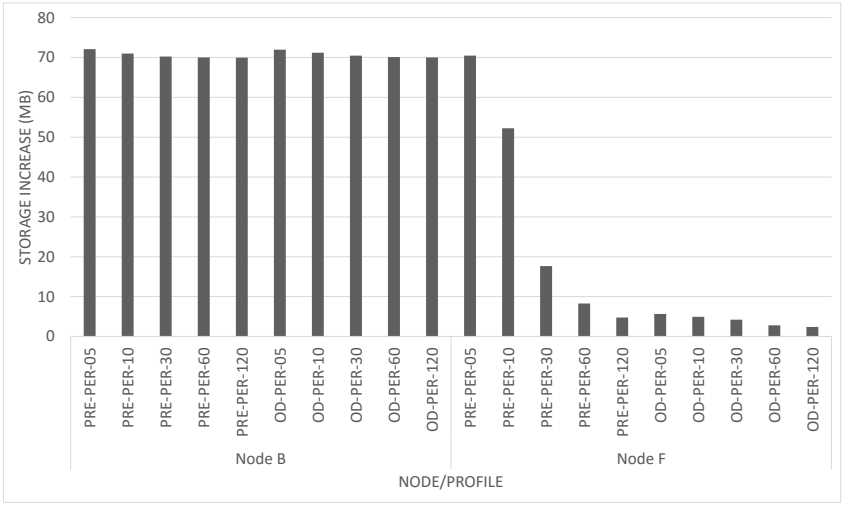


Figure 7.14: Storage resource utilization for an evaluation of synchronization frequency for periodic commits.

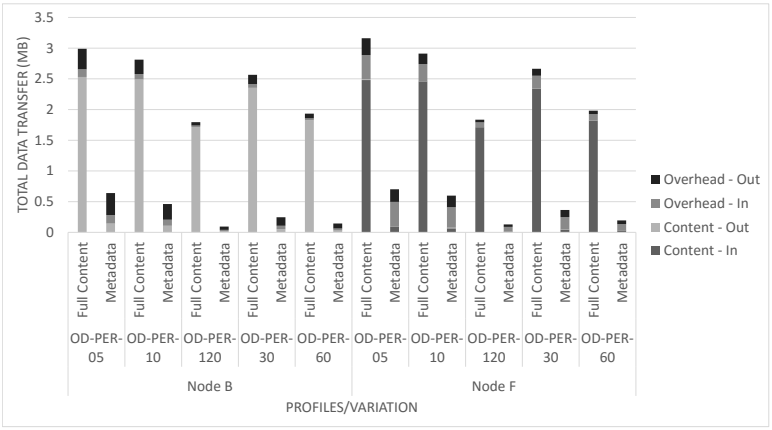


Figure 7.15: Comparison of total amount of data transfer between content and metadata retrieval for an evaluation of synchronization frequency for periodic commits.

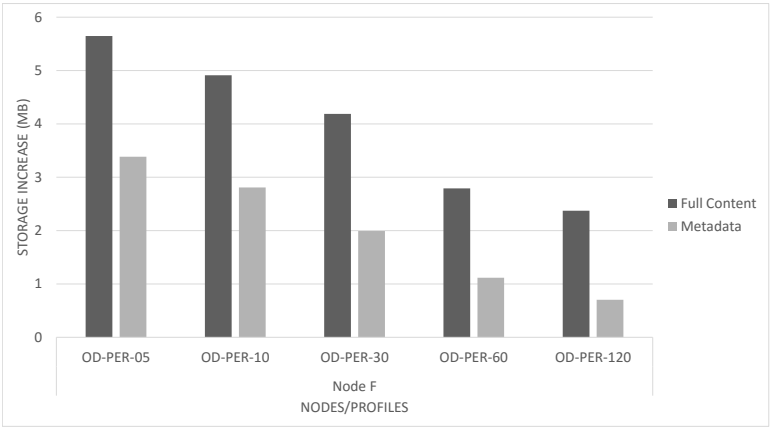


Figure 7.16: Comparison of storage resource utilization in the consumer between content and metadata retrieval for an evaluation of synchronization frequency for periodic commits.

Profile	Full Content	Metadata
PRE-PER-05	0.010	0.009
PRE-PER-10	0.010	0.009
PRE-PER-30	0.010	0.009
PRE-PER-60	0.010	0.009
PRE-PER-120	0.010	0.009
OD-PER-05	0.054	0.048
OD-PER-10	0.060	0.041
OD-PER-30	0.074	0.058
OD-PER-60	0.040	0.041
OD-PER-120	0.034	0.042

Table 7.8: Comparison of access latency (in seconds) between content and metadata retrieval in an evaluation of synchronization frequency for periodic commits.

Profile	Full Content	Metadata
PRE-PER-05	1.3	1.1
PRE-PER-10	1.6	2.0
PRE-PER-30	11.6	9.9
PRE-PER-60	27.2	26.6
PRE-PER-120	52.7	47.6
OD-PER-05	0.8	1.4
OD-PER-10	2.3	3.0
OD-PER-30	12.1	9.9
OD-PER-60	37.7	24.5
OD-PER-120	41.1	45.9

Table 7.9: Comparison of conflict rate (as defined in Section 7.1.3) between content and metadata retrieval in an evaluation of synchronization frequency for periodic commits.

presented in Appendix A, in Tables A.10 to A.12. Access latency is described in Table 7.8, while conflict rate is presented in Table 7.9. Node B produces data, while node F consumes it. Nodes M and N are idle. To aid the visualization of other nodes' data, node N is not shown in some graphs, as results are similar to node M.

Figure 7.13 shows that the amount of data transfer used by Unico decreases as the interval between updates is increased. This result is more accentuated for data transfer with full prefetching (PRE-PER), but is also evident with on demand (OD-PER). Storage resources are also saved in the consumer node by decreasing the synchronization frequency, as shown in Figures 7.14 and 7.16. However, the use of a higher interval for synchronization also incurs in significant rates of conflict, as shown in Table 7.9.

7.6 Scalability on Number of Nodes

All scenarios above are tested with a set of four nodes. In order to identify the potential impact of the number of nodes in the overall performance of the system, a scenario equivalent to the one presented in Section 7.2 was repeated with two, four and six nodes. The profiles described in crefsec:profiles are employed in three variations each, with a suffix indicating the number of nodes in place. Nodes B

Profile	Full Content	Metadata
PRE-IMM-6N	0.008	0.006
PRE-IMM-4N	0.008	0.006
PRE-IMM-2N	0.008	0.007
PRE-PER-6N	0.008	0.006
PRE-PER-4N	0.008	0.006
PRE-PER-2N	0.008	0.006
PRE-INV-6N	1.062	1.061
PRE-INV-4N	1.064	1.046
PRE-INV-2N	1.027	1.020
OD-IMM-6N	0.062	0.056
OD-IMM-4N	0.062	0.053
OD-IMM-2N	0.062	0.052
OD-PER-6N	0.008	0.006
OD-PER-4N	0.008	0.006
OD-PER-2N	0.008	0.006
OD-INV-6N	1.080	1.070
OD-INV-4N	1.075	1.068
OD-INV-2N	1.032	1.023

Table 7.10: Comparison of access latency (in seconds) between content and metadata retrieval in an evaluation of scalability on number of nodes.

and F reprise their role as producer and consumer. The two-node variation (e.g., OD-IMM-2N) is limited to these two nodes; the four-node variation (e.g., OD-IMM-4N) includes the same nodes M and N as before, while two additional nodes, C and D, join them in the six-node variation (e.g., OD-IMM-6N).

This section discusses the results of the experiments, while measurements are presented as follows. Figures 7.17 and 7.18 compare network and storage utilization between profiles when full file content is retrieved in a read. The data that generated these graphs is presented in Appendix A, in Tables A.13 to A.16. Access latency is described in Table 7.10, while conflict rate is presented in Table 7.11. Node B produces data, while node F consumes it. Nodes M, N, C and D are idle. For clarity, nodes C, D and N are not shown in some graphs, as results are similar to node M.

This experiment demonstrates that, as expected, in a full prefetching setting (PRE-IMM), the network traffic generated by data synchronization increases dramatically as the number of nodes increases, as shown in Figure 7.17. This is ex-

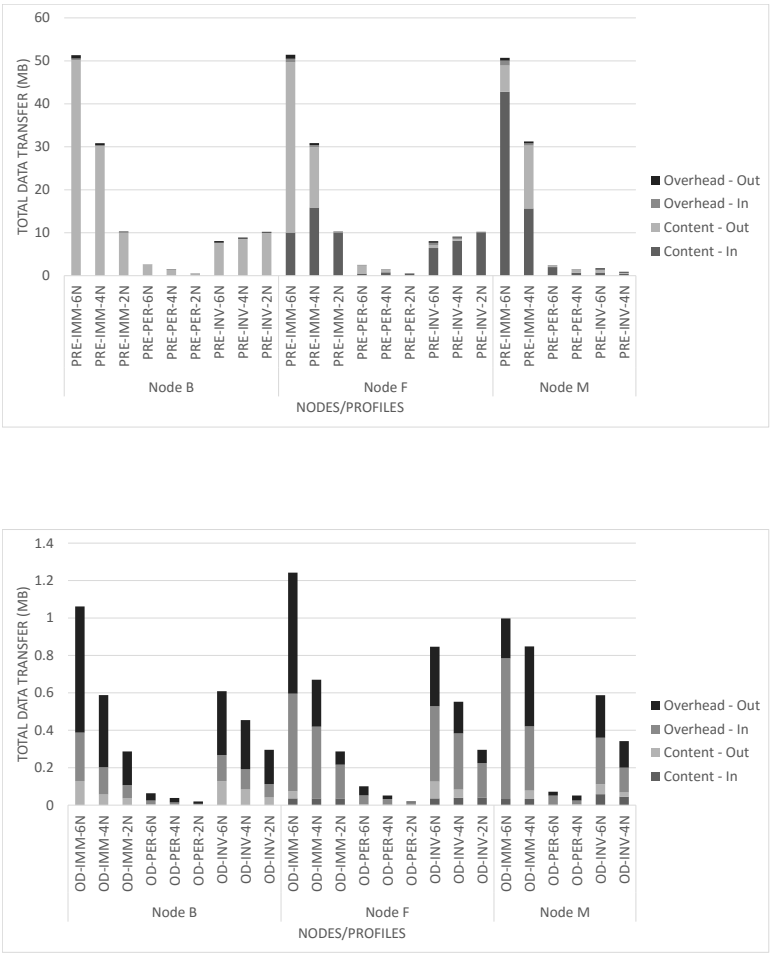


Figure 7.17: Total amount of data transfer for an evaluation of scalability on number of nodes for full content access. The top graph shows only prefetch profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

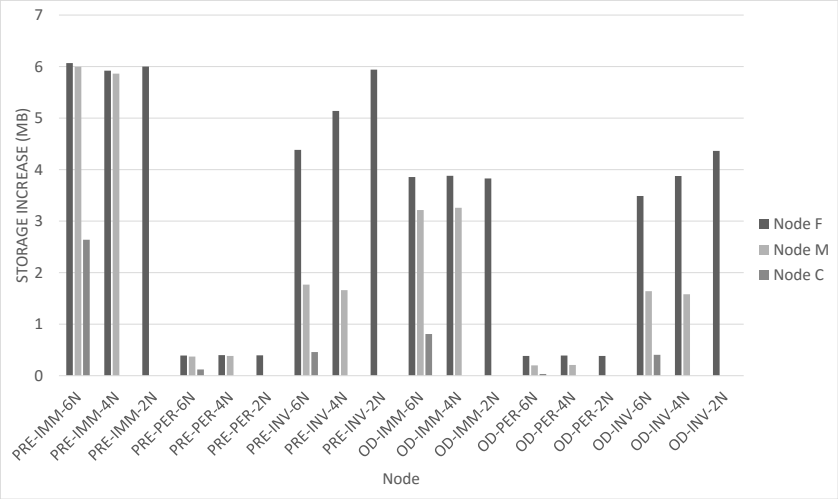


Figure 7.18: Storage resource utilization for an evaluation of scalability on number of nodes.

Profile	Full Content	Metadata
PRE-IMM-6N	1.3	1.3
PRE-IMM-4N	1.2	1.2
PRE-IMM-2N	1.2	1.2
PRE-PER-6N	25.1	24.8
PRE-PER-4N	27.5	27.7
PRE-PER-2N	28.2	25.5
PRE-INV-6N	0.0	0.0
PRE-INV-4N	0.0	0.0
PRE-INV-2N	0.0	0.0
OD-IMM-6N	1.3	1.3
OD-IMM-4N	1.3	1.3
OD-IMM-2N	1.3	1.3
OD-PER-6N	25.6	25.2
OD-PER-4N	27.8	27.7
OD-PER-2N	25.4	25.4
OD-INV-6N	0.0	0.0
OD-INV-4N	0.0	0.0
OD-INV-2N	0.0	0.0

Table 7.11: Comparison of conflict rate (as defined in Section 7.1.3) between content and metadata retrieval in an evaluation of scalability on number of nodes.

plained by the need to transfer all data to all nodes, even data that is not needed in those nodes. The on demand profile (OD-IMM) follows a similar pattern, but at a smaller proportion, since not all nodes need all data. Access latency and conflict rate, described in Tables 7.10 and 7.11, are not statistically affected by the change in number of nodes. Storage resources, shown in Figure 7.18, are also not significantly affected by the number of nodes in most profiles.

In profiles involving path invalidation, results show an interesting pattern. The use of storage resources, particularly in the consumer node, is higher in a scenario with two nodes when compared to scenarios with four or six nodes. This is evident when comparing storage resources for node F in PRE-INV and OD-INV profiles in Figure 7.18. Further investigation of this behaviour indicates that, as the number of nodes increases, the load of handling the process of invalidation and merging spreads among different nodes. Because of that, a scenario where only two nodes are available forces these two nodes to process all invalidation and merging on their own. A similar behaviour is observed in Figure 7.17 for network traffic, although to a lesser degree and only for the prefetching profiles (PRE-INV).

7.7 Scalability on Number of Modified Files

All scenarios above are tested with a set of four nodes. In order to identify the potential impact of the number of nodes in the overall performance of the system, a scenario equivalent to the one presented in Section 7.3 was repeated, but instead modifying 5, 50 and 90 nodes. Nodes B and F reprise their role as producer and consumer. Only the full content scenario is tested for all six profiles described in Section 7.1.1.

This section discusses the results of the experiments, while measurements are presented as follows. Figures 7.19 and 7.20 compare network and storage utilization between profiles when full file content is retrieved in a read. The data that generated these graphs is presented in Appendix A, in Tables A.17 to A.19. Access latency is described in Table 7.12, while conflict rate is presented in Table 7.13. Node B produces data, while node F consumes it. Nodes M and N are idle. To aid the visualization of other nodes' data, node N is not shown in some graphs, as results are similar to node M.

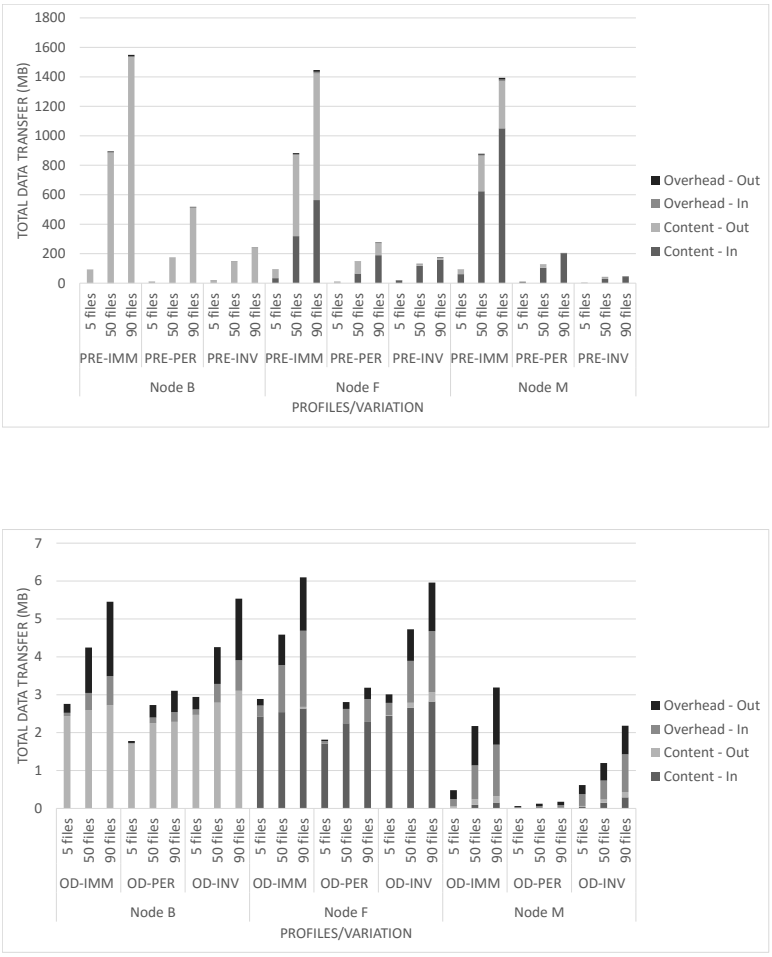


Figure 7.19: Total amount of data transfer for an evaluation of scalability on number of modified files. The top graph shows only prefetching profiles, while the bottom graph shows only on demand profiles, with a different range for better visualization.

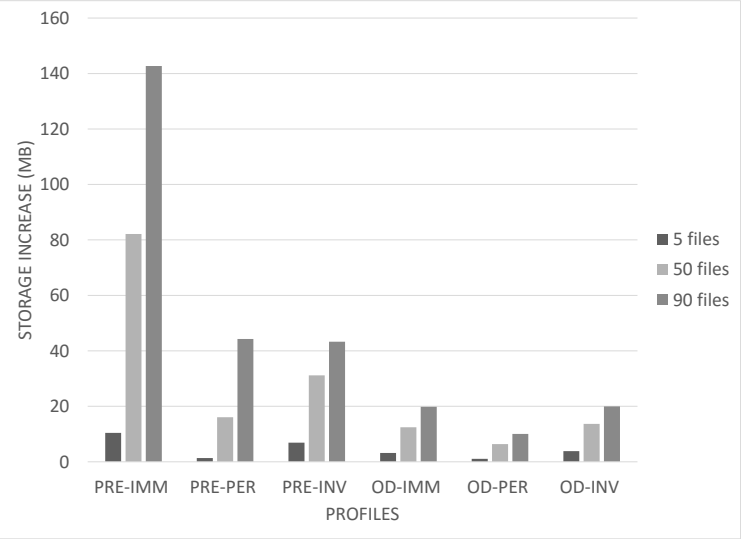


Figure 7.20: Storage resource utilization for an evaluation of scalability on number of modified files on the consumer.

Profile	5 files	50 files	90 files
PRE-IMM	0.009	0.009	0.009
PRE-PER	0.009	0.009	0.009
PRE-INV	0.274	0.320	0.306
OD-IMM	0.073	0.060	0.058
OD-PER	0.048	0.057	0.057
OD-INV	0.340	0.332	0.370

Table 7.12: Comparison of access latency (in seconds) for content retrieval in an evaluation of scalability on number of modified files.

Profile	5 files	50 files	90 files
PRE-IMM	0.0	0.0	0.0
PRE-PER	29.6	26.1	26.1
PRE-INV	0.0	0.0	0.0
OD-IMM	0.0	0.0	0.4
OD-PER	28.9	22.7	23.6
OD-INV	0.0	0.0	0.0

Table 7.13: Comparison of conflict rate (as defined in Section 7.1.3) for content retrieval in an evaluation of scalability on number of modified files.

The results shown in Figure 7.19 demonstrate that, as expected, in full prefetching profiles, the network traffic is roughly proportional to the number of files being modified. In on demand profiles, however, network traffic grows significantly slower, allowing it to scale much better in this scenario. In particular, while in the PRE-IMM profile a tenfold increase in the number of modified files generated an increase in traffic at the same rate in the consumer, the same change in modified files generates less than double the traffic in the OD-IMM profile. A similar behaviour is seen in the use of storage resources, as shown in Figure 7.20.

A comparison of access latency and conflict rate presented in Tables 7.12 and 7.13 shows that these metrics are not directly affected by the number of modified files in any profile.

7.8 Discussion

The results of these experiments, particularly the comparison between the PRE-IMM and OD-IMM profiles, show that, as expected, a full synchronization, as used in other distributed platforms, provide a shorter access latency to the files; however this latency also results in a less efficient use of bandwidth and storage resources. Devices where these resources are scarce or expensive (particularly, but not limited to, mobile devices) will greatly benefit from a trade-off approach to these limitations, based on limiting data transfer to files actually used; however other devices may also benefit from this approach, since a more efficient use of network connectivity and storage allows these resources to be used for other tasks.

These results also demonstrate that, although a periodic synchronization strategy (described by the PRE-PER profile), as used by OriFS, reduces the use of network and storage resources, it can introduce significant conflict rates, if data is accessed in one node shortly after it is created in another node.

Path invalidation behaves as expected. Idle nodes can save storage and network resources by receiving updates at a significantly lower rate. Nodes that require recently updated data have access to the most recent version of that data with little or no conflict, although with some impact in access latency. Nodes that produce data can also save resources through the resulting reduction of outgoing data transfer.

The access of metadata through extended attributes allows an application to have faster access to information while saving network and storage resources and reducing the latency associated to retrieving the data and processing it. This type of file access finds some level of improvement in all profiles, although it is more evident in on demand profiles.

An on demand approach also provides better scalability when a large number of files is modified. Even though data transfer and storage utilization are increase with a larger number of modified files, this increase is significantly smaller than the one resulting from full prefetching of all the data.

Chapter 8

Conclusion

This chapter provides a summary of the dissertation and discusses the possible research directions and future work on Unico and on demand data synchronization.

8.1 Summary

Many computer users have access to several devices with different characteristics. These devices need to have consistent access to the user's entire data corpus, including the latest version of any files changed or created in other devices. Several strategies have been proposed to address this requirement, including centralized servers, cloud storage technologies and peer-to-peer data transfer. This dissertation proposes a different approach, where data is transferred on demand between peers as it is needed. Devices have a consistent view of the user's data, but storage is limited to files required in each device.

The results presented in this dissertation show how the described objectives have been achieved. Unico allows a user to reduce the use of storage and network resources by avoiding the transfer of files to peers that don't require them. Data is transparently replicated on demand, without the need for explicit user intervention. Although latency and availability is affected by the delayed transfer of data, policy configurations allow a user to fine-tune the system to their specific needs.

File operations that depend on specific metadata can be performed locally in nodes without retriving the file content. Content metadata are available to users

and applications through extended attributes. This dissertation demonstrate that the use of metadata when possible allow an application to reduce its use of storage and network resources, as well as latency, without detriment to data consistency.

When object content is needed in a node, a heuristic process of object location allows that node to select a suitable peer to request that content from. Peers known to have the content are given priority, and data is requested from one of these nodes based on expected latency to request it.

A path invalidation process allows an idle node to reduce synchronization frequency and, as a consequence, reduce the need for storage and network resources in that node. Results show that this process reduces the amount of data transfer and storage requirements, most notably in idle nodes, while maintaining consistency across all devices.

Although Unico is implemented as a fully-functioning prototype, further development of the proposed strategies as full applications would allow a user to benefit from these results in a more practical scenario. The application of existing research into a more robust application, including a more user-friendly interface, extended authentication and support for other operating systems and mobile platforms, should result in a better user experience as compared to existing solutions.

Experimental results show that a full synchronization strategy provides shorter access latency, but implies a less efficient use of bandwidth and storage resources. Devices where these resources are scarce or expensive, including mobile devices, may benefit from an on demand approach, since a more efficient use of network connectivity and storage allows these resources to be used for other tasks. Compared to the strategies taken by systems like Dropbox and OriFS, an on demand approach to file synchronization can save up to 98% of network traffic and 92% of storage utilization in some scenarios. Path invalidation allows idle nodes to save additional storage and network resources, in some cases by 94% and 96% respectively, by receiving updates at a significantly lower rate, while also allowing active nodes to save their own network resources through the resulting reduction of outgoing data transfer.

Experiments also show that the access of metadata through extended attributes allows an application to have faster access to information while saving network and storage resources and reducing the latency associated to retrieving the data and

processing it. An on demand approach also provides better scalability when a large number of files is modified. Even though data transfer and storage utilization are increase with a larger number of modified files, this increase is significantly smaller than the one resulting from full prefetching of all the data.

8.2 Future Research Directions

The lessons learned through the research in this dissertation open up a few directions for further research in the area. One such direction is the application of automated prefetching rules based on use cases and relations between files. For example, when a user often opens a set of files together, such as includes linked by a programming source file, the system could preemptively retrieve other files in the set when one of the files is requested. Other rules could be determined based on the user's behaviour related to criteria like time of day and specific types of files (e.g., JPEG images).

The implementation of a system like Unico in mobile devices needs to account for the availability and cost of local resources. In particular, information like charging status, battery level, space availability and network connectivity type (e.g., Wi-Fi, 4G) should dictate how Unico stores and transfers files. Additionally, this type of policy adaptation should also affect connected peers. Nodes with abundant storage and cheap connectivity should avoid retrieving objects from peers in restricted or metered connections if an alternative is available.

Better strategies for reducing storage resource utilization would also require some mechanism to remove local copies of objects in some nodes. This mechanism needs to be based on a reasonable expectation that the object will not be needed in those specific nodes in the near future, and that the object can be obtained from another source should it be needed. There should also be a negotiation phase that ensures that no peer deletes the last version of an object. Additional research into options to remove old unwanted versions of a file can also be considered, as well as ways to store objects into external media for backup purposes, including potentially the last copy of older versions of a file, using strategies such as those used in Git Annex [28].

Path invalidation is another field of research that could benefit from further research. Several improvements could be made in the way that nodes invalidate and revalidate paths. In particular, idleness in a producer could be used to revalidate paths in other nodes (or at least resynchronize the data), generating potentially improved results in latency when a user switches devices.

OriFS implements a new type of object called "large blob", which splits content of very large files into smaller chunks [39]. This type of object allows the Git-based system to better handle large files, as well as to improve deduplication of data segments. The application of this type of object into a system like Unico is viable and probably beneficial, and is a potential direction for future research.

Further improvements in use of network resources, particularly latency, can potentially be obtained by evaluating the use of deltas when transferring commits and trees. Instead of sending trees as a mapping from name to object, as usual, this process could send the difference between the tree and its previous version. The receiver would then applied this delta to its own version and generate the new tree. This process would require more detailed knowledge of what data is available in other peers, so that a delta does not trigger more data transfer than it intends.

Bibliography

[1] Android Open Source Project. Developing — Android Open Source Project. URL https://source.android.com/source/developing.html. [accessed November 2015]. → pages 26

[2] Apple Inc. What is iCloud? - iCloud Help. URL https://help.apple.com/icloud/#/mmfc0efea4. [accessed July 2016]. → pages 8

[3] R. M. Baecker, J. Grudin, W. Buxton, and S. Greenberg. *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann, Jan. 1995. → pages 24

[4] D. Belson *et al.* The State of the Internet, 2nd Quarter. Technical report, Akamai Technologies, 2013. URL http://www.akamai.com/dl/documents/akamai_soti_q213.pdf?WT.mc_id=soti_Q213. → pages 1

[5] BitTorrent, Inc. BitTorrent Sync, 2015. URL https://www.getsync.com/. [accessed March 24, 2016]. → pages 3, 13, 64

[6] J. Blair. Introducing Samba. *Linux Journal*, 1998(51es), July 1998. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=327422.327434. → pages 7, 21

[7] P. J. Braam. The Coda Distributed File System. *Linux Journal*, (50), June 1998. → pages 8, 39

[8] S. Chacon and B. Straub. *Pro Git*. Apress, 2nd edition edition, Nov. 2014. → pages 24, 26, 28

[9] B. Chen. *A Serverless, Wide-Area Version Control System*. PhD thesis, Massachusetts Institute of Technology (MIT), 2004. → pages 10

[10] D. Chisnall. Examining the legendary HURD kernel. *InformIT, Pearson Education*, Mar. 2008. → pages 21

[11] Conifer Systems LLC. Cascade: Overview, 2010. URL http://www.conifersystems.com/cascade/overview/. [accessed Feb. 6, 2012]. → pages 10

[12] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 9 1994. ISBN 9781556156601. → pages 2, 20

[13] cvsFS. cvsFS - mount a CVS-Tree, 2009. URL http://cvsfs.sourceforge.net/. [accessed Feb. 6, 2012]. → pages 10

[14] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3): 143–155, Mar. 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL http://doi.acm.org/10.1145/365230.365252. → pages 21

[15] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, Sept. 1965. ISSN 0001-0782. doi: 10.1145/365559.365617. URL http://doi.acm.org/10.1145/365559.365617. → pages 47

[16] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet Measurement Conference*, IMC '12, pages 481–494, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1705-4. doi: 10.1145/2398776.2398827. URL http://doi.acm.org/10.1145/2398776.2398827. → pages 2, 8, 64

[17] Drupal. Migrating Drupal.org to Git. URL http://drupal.org/community-initiatives/git. [accessed November 2015]. → pages 26

[18] D. E. Eastlake and P. E. Jones. US Secure Hash Algorithm 1 (SHA1). Internet RFC 3174, September 2001. → pages 28

[19] J. Farina, M. Scanlon, and M.-T. Kechadi. BitTorrent Sync: First Impressions and Digital Forensic Implications. *Digital Investigation*, 11, Supplement 1(0):S77–S86, 2014. ISSN 1742-2876. doi: 10.1016/j.diin.2014.03.010. URL http://www.sciencedirect.com/science/article/pii/S1742287614000152. → pages 3

[20] Gemius SA. Features, Architecture and Requirements: MooseFS network file system. http://www.moosefs.org/. [accessed November 2015]. → pages 8

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450. → pages 8

[22] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, Mar. 1988. ISSN 0001-0782. doi: 10.1145/42392.42398. URL http://doi.acm.org/10.1145/42392.42398. → pages 7

[23] G. D. Gonçalves, I. Drago, A. V. Borges, A. P. Couto, and J. M. de Almeida. Analysing costs and benefits of content sharing in cloud storage. In *Proceedings of the 2016 Workshop on Fostering Latin-American Research in Data Communication Networks*, LANCOMM '16, pages 43–45, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4426-5. doi: 2940116.2940128. URL http://doi.acm.org/2940116.2940128. → pages 24

[24] Google, Inc. Google Drive – Cloud Storage & File Backup for Photos, Docs & More. URL https://www.google.com/drive/. [accessed July 2016]. → pages 8

[25] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, Anaheim, CA, June 1990. USENIX. → pages 12

[26] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the Workshops on Data Warehousing and Data Mining: Advances in Database Technologies*, ER '98, pages 254–265, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-65690-1. → pages 12

[27] D. Hardt. The OAuth 2.0 Authorization Framework. Internet RFC 6749, October 2012. ISSN 2070-1721. → pages 23

[28] J. Hess *et al.* git-annex. https://git-annex.branchable.com/. [accessed March 1, 2014]. → pages 11, 95

[29] T. Hildmann and O. Kao. Deploying and extending on-premise cloud storage based on owncloud. In *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 76–81, June 2014. doi: 10.1109/ICDCSW.2014.18. → pages 13

[30] M. E. Hoskins. Sshfs: Super easy file access over ssh. *Linux Journal*, 2006 (146):4–, June 2006. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1134782.1134786. → pages 21

[31] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988. ISSN 0734-2071. doi: 10.1145/35037.35059. URL http://doi.acm.org/10.1145/35037.35059. → pages 8

[32] Ilan Shamir. Sync Security and Privacy Brief Now Available, Dec. 2015. URL http://blog.bittorrent.com/2015/12/14/sync-security-and-privacy-brief-now-available/. [accessed August 2016]. → pages 23

[33] J. Janak, J. W. Lee, and H. Schulzrinne. GRAND: Git Revisions As Named Data. In *SIGCOMM Information Centric Networking Workshop*, New York, NY, USA, 2011. ACM. URL http://janakj.org/papers/grand.pdf. (submitted). → pages 10

[34] R. Kapitza, P. Baumann, and H. P. Reiser. Using object replication for building a dependable version control system. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 86–99, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-68639-8, 978-3-540-68639-2. → pages 10

[35] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, Aug. 1974. ISSN 0001-0782. doi: 10.1145/361082.361093. URL http://doi.acm.org/10.1145/361082.361093. → pages 47

[36] J. B. Layton. User Space File Systems. *Linux Magazine*, June 2010. URL http://www.linux-mag.com/id/7814/. → pages 21, 22, 34

[37] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient Batched Synchronization in Dropbox-Like Cloud Storage Services. In D. Eyers and K. Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 307–327. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45064-8. doi: 10.1007/978-3-642-45065-5_16. URL http://dx.doi.org/10.1007/978-3-642-45065-5_16. → pages 18, 36

[38] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005. → pages 11

[39] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, History, and Grafting in the Ori File System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522721. URL http://doi.acm.org/10.1145/2517349.2522721. → pages 1, 3, 12, 24, 47, 64, 96

[40] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Thomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, Ottawa, ON, Canada, 6 2007. → pages 2

[41] E. Mouw. *Linux Kernel Procfs Guide*, revision 1.1 edition, June 2001. → pages 21

[42] P. Mukherjee. *A fully Decentralized, Peer-to-Peer Based Version Control System*. PhD thesis, TU Darmstadt, Mar. 2011. URL http://tuprints.ulb.tu-darmstadt.de/2488/. → pages 10

[43] One Laptop per Child. Git - OLPC. URL http://wiki.laptop.org/go/Git. → pages 26

[44] B. O'Sullivan. Making Sense of Revision-control Systems. *ACM Queue*, 7 (7), Aug. 2009. → pages 9, 10, 24, 25

[45] ownCloud. ownCloud.org. URL https://owncloud.org/. [accessed July 2016]. → pages 13

[46] Perl.org. Getting and Working With the Perl Source. URL http://dev.perl.org/perl5/source.html. [accessed November 2015]. → pages 26

[47] O. Peters and S. ben Allouch. Always connected: a longitudinal field study of mobile communication. *Telematics and Informatics*, 22(3):239 – 256, 2005. ISSN 0736-5853. doi: 10.1016/j.tele.2004.11.002. URL http://www.sciencedirect.com/science/article/pii/S0736585304000607. → pages 13, 16

[48] D. Quick and K.-K. R. Choo. Forensic Collection of Cloud Storage Data: Does the Act of Collection Result in Changes to the Data or Its Metadata? *Digital Investigation*, 10(3):266–277, Oct. 2013. ISSN 1742-2876. doi: 10.1016/j.diin.2013.07.001. URL http://dx.doi.org/10.1016/j.diin.2013.07.001. → pages 3, 8

[49] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A Platform for Content-based Partial Replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association. → pages 12

[50] D. M. Ritchie. The Evolution of the Unix Time-sharing System. In *Language Design and Programming Methodology Conference*, Sydney, Australia, Sept. 1979. → pages 20

[51] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. In *Fourth ACM Symposium on Operating Systems Principles*, New York, NY, USA, Oct. 1973. Association for Computing Machinery. → pages 20

[52] D. Santry, M. Feeley, N. Hutchinson, and A. Veitch. Elephant: the file system that never forgets. In *Hot Topics in Operating Systems, 1999. Proceedings of the 7^{th} Workshop on*, pages 2–7, Rio Rico, AZ, USA, Mar. 1999. → pages 8, 21

[53] M. Satyanarayan, J. J. Kistler, and E. H. Siegel. Coda: A resilient distributed file system. In *IEEE Workshop on Workstation Operating Systems*, Cambridge, MA, Nov. 1987. → pages 8

[54] R. Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, pages 101–102. IEEE Computer Society, 2001. → pages 11

[55] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. Internet RFC 3010, December 2000. → pages 7, 21

[56] K. Shvachko, K. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Incline Village, NV, May 2010. → pages 8

[57] G. Silberschatz. *Operating System concepts*. Addison-Wesley Publishing Company, 1994. → pages 7

[58] Sun Microsystems, Inc. Lustre File System–High-Performance Storage Architecture and Scalable Cluster File System. White paper, Sun Microsystems, Inc., Dec. 2007. → pages 8

[59] The GNOME Project. Git - Gnome Wiki! URL http://live.gnome.org/Git. [accessed November 2015]. → pages 26

[60] L. Torvalds. Google Tech Talk: Git, 2007. URL http://www.youtube.com/watch?v=4XpnKHJAok8. [accessed August 25, 2011]. → pages 26

[61] L. Torvalds and J. C. Hamano. Git Users Manual, 2011. URL http://www.kernel.org/pub/software/scm/git/docs/user-manual.html#trust. [accessed August 25, 2011]. → pages 29

[62] Upthere. Upthere – What is Upthere. https://www.upthere.com/what-is-upthere/. [accessed November 2015]. → pages 9

[63] J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994. → pages 19

[64] S. A. Weil. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, University of California Santa Cruz, Santa Cruz, Dec. 2007. → pages 8

[65] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. → pages 8

[66] D. A. Wheeler. Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) / Revision-Control Systems, 2004. URL http://www.dwheeler.com/essays/scm.html. → pages 10, 25

[67] J. Yu, W. Wu, and H. Li. DMooseFS: Design and implementation of distributed file system with distributed metadata server. In *Cloud Computing Congress (APCloudCC), 2012 IEEE Asia Pacific*, pages 42–47, Shenzhen,

China, Nov. 2012. IEEE. doi: 10.1109/APCloudCC.2012.6486509. → pages 8

Appendix A

Network and Storage Utilization Details

This chapter describes in more details the results of network and storage utilization metrics for the scenarios described in Chapter 7.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	Full Content	PRE-IMM	176/	451	0/	25439	26067
		PRE-PER	14/	31	0/	1531	1577
		PRE-INV	115/	274	0/	10462	10851
		OD-IMM	163/	424	0/	2270	2856
		OD-PER	12/	30	0/	619	661
		OD-INV	108/	262	0/	2198	2569
	Metadata	PRE-IMM	174/	455	0/	25144	25772
		PRE-PER	16/	34	0/	1836	1886
		PRE-INV	122/	291	0/	8968	9381
		OD-IMM	128/	361	0/	227	716
		OD-PER	14/	31	0/	21	66
		OD-INV	112/	267	0/	218	597
Node F	Full Content	PRE-IMM	482/	425	17532/	6765	25203
		PRE-PER	38/	31	1017/	405	1491
		PRE-INV	387/	211	10314/	142	11054
		OD-IMM	395/	329	2122/	24	2870
		OD-PER	39/	22	612/	0	672
		OD-INV	284/	171	2154/	43	2653
	Metadata	PRE-IMM	511/	362	22892/	2109	25874
		PRE-PER	44/	37	819/	809	1710
		PRE-INV	384/	228	8611/	352	9574
		OD-IMM	368/	282	123/	7	780
		OD-PER	42/	29	13/	1	84
		OD-INV	301/	207	166/	51	726

Table A.1: Total data transfer per profile in a single-file producer/consumer experiment (Section 7.2), in KB, for nodes B and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	Full Content	PRE-IMM	463/	433	13316/	11082	25294
		PRE-PER	32/	35	711/	812	1590
		PRE-INV	171/	164	572/	303	1210
		OD-IMM	341/	359	109/	27	836
		OD-PER	25/	37	5/	3	69
		OD-INV	151/	118	67/	1	337
	Metadata	PRE-IMM	432/	425	10917/	14092	25866
		PRE-PER	44/	42	1015/	712	1813
		PRE-INV	167/	177	458/	126	928
		OD-IMM	325/	416	98/	53	891
		OD-PER	34/	41	6/	3	84
		OD-INV	163/	125	78/	1	367
Node N	Full Content	PRE-IMM	488/	301	13847/	1409	16046
		PRE-PER	42/	29	1221/	200	1493
		PRE-INV	152/	176	245/	224	797
		OD-IMM	425/	212	94/	4	735
		OD-PER	33/	20	5/	0	57
		OD-INV	125/	117	44/	21	308
	Metadata	PRE-IMM	425/	301	11086/	3550	15361
		PRE-PER	48/	39	1626/	102	1814
		PRE-INV	190/	166	582/	205	1143
		OD-IMM	373/	135	66/	1	574
		OD-PER	38/	27	6/	0	70
		OD-INV	154/	132	52/	25	364

Table A.2: Total data transfer per profile in a single-file producer/consumer experiment (Section 7.2), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Total
Full Content	OD-IMM	12132	4708	2644	2484	21968
	OD-INV	12788	4808	1100	1052	19748
	OD-PER	11292	804	124	132	12352
	PRE-IMM	9032	9796	9672	5772	34272
	PRE-INV	11628	11140	1380	1240	25388
	PRE-PER	9592	608	584	580	11364
Metadata	OD-IMM	12396	2852	2532	1908	19688
	OD-INV	12868	2940	1280	1288	18376
	OD-PER	11308	244	152	144	11848
	PRE-IMM	10100	11076	10844	5952	37972
	PRE-INV	11820	10828	1448	1632	25728
	PRE-PER	10116	776	732	724	12348

Table A.3: Total storage increase per profile in a single-file producer/consumer experiment (Section 7.2). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	Full Content	PRE-IMM	607/	1344	0/	178551	180501
		PRE-PER	64/	138	0/	21548	21750
		PRE-INV	210/	435	0/	44698	45344
		OD-IMM	146/	380	0/	2905	3430
		OD-PER	36/	87	0/	2288	2411
		OD-INV	155/	321	0/	3260	3736
	Metadata	PRE-IMM	559/	1235	0/	172041	173836
		PRE-PER	64/	136	0/	21548	21748
		PRE-INV	210/	431	0/	40477	41117
		OD-IMM	127/	347	0/	530	1004
		OD-PER	34/	83	0/	189	307
		OD-INV	153/	319	0/	750	1221
Node F	Full Content	PRE-IMM	1642/	1414	85240/	86284	174580
		PRE-PER	177/	150	11058/	10466	21851
		PRE-INV	855/	482	41135/	1480	43953
		OD-IMM	419/	334	2777/	43	3573
		OD-PER	117/	59	2279/	1	2456
		OD-INV	422/	261	3153/	53	3889
	Metadata	PRE-IMM	1578/	1305	91066/	74346	168294
		PRE-PER	174/	153	8082/	12542	20952
		PRE-INV	843/	511	36440/	3007	40801
		OD-IMM	407/	306	418/	27	1159
		OD-PER	114/	64	180/	2	359
		OD-INV	419/	272	702/	44	1437

Table A.4: Total data transfer per profile in a multiple-file producer/consumer experiment (Section 7.3), in KB, for nodes B and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	Full Content	PRE-IMM	1623/	1424	106932/	63646	173625
		PRE-PER	171/	157	13335/	8090	21754
		PRE-INV	281/	345	6240/	1752	8619
		OD-IMM	294/	304	111/	30	739
		OD-PER	33/	51	7/	4	94
		OD-INV	133/	136	34/	19	321
	Metadata	PRE-IMM	1539/	1323	100304/	64607	167772
		PRE-PER	171/	142	14535/	6801	21650
		PRE-INV	316/	365	5576/	1476	7733
		OD-IMM	252/	290	86/	28	656
		OD-PER	44/	51	8/	3	105
		OD-INV	193/	156	72/	1	422
Node N	Full Content	PRE-IMM	1624/	1315	144535/	8226	155700
		PRE-PER	179/	147	18673/	2962	21961
		PRE-INV	250/	334	2534/	1978	5096
		OD-IMM	374/	218	99/	10	701
		OD-PER	44/	33	7/	0	84
		OD-INV	157/	151	147/	2	457
	Metadata	PRE-IMM	1419/	1233	128033/	8409	139094
		PRE-PER	173/	152	19550/	1276	21150
		PRE-INV	288/	352	3559/	614	4813
		OD-IMM	334/	176	81/	0	590
		OD-PER	45/	39	8/	1	93
		OD-INV	157/	174	45/	25	401

Table A.5: Total data transfer per profile in a multiple-file producer/consumer experiment (Section 7.3), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Total
Full Content	OD-IMM	73956	5872	2220	2160	84208
	OD-INV	74700	7080	824	1128	83732
	OD-PER	70276	3288	144	144	73852
	PRE-IMM	73468	71696	71552	62464	279180
	PRE-INV	74692	44952	5232	3512	128388
	PRE-PER	69408	8252	8216	8152	94028
Metadata	OD-IMM	74112	3632	1832	1708	81284
	OD-INV	74492	4404	1196	1108	81200
	OD-PER	70276	1200	180	164	71820
	PRE-IMM	73368	71452	71588	54900	271308
	PRE-INV	74824	42732	4588	3504	125648
	PRE-PER	69848	8292	8188	8184	94512

Table A.6: Total storage increase per profile in a multiple-file producer/consumer experiment (Section 7.3). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	Full Content	PRE-IMM	963/	1065	70366/	72234	144628
		PRE-PER	119/	130	9908/	9239	19395
		PRE-INV	202/	337	5184/	18522	24245
		OD-IMM	306/	338	1230/	2230	4104
		OD-PER	72/	91	1038/	2314	3515
		OD-INV	238/	309	1216/	2989	4752
	Metadata	PRE-IMM	1014/	1131	70746/	77171	150062
		PRE-PER	111/	132	7901/	10323	18468
		PRE-INV	448/	674	5272/	28610	35004
		OD-IMM	278/	247	1085/	1333	2942
		OD-PER	65/	66	1054/	177	1362
		OD-INV	211/	307	1154/	1931	3603
Node F	Full Content	PRE-IMM	1351/	1170	78664/	63935	145121
		PRE-PER	160/	144	8084/	10745	19133
		PRE-INV	707/	418	31850/	5665	38640
		OD-IMM	443/	293	2947/	349	4031
		OD-PER	131/	72	2494/	1	2698
		OD-INV	514/	464	3455/	369	4802
	Metadata	PRE-IMM	1400/	1224	72115/	72000	146740
		PRE-PER	164/	138	9271/	8950	18523
		PRE-INV	810/	973	27648/	4982	34412
		OD-IMM	403/	374	459/	97	1333
		OD-PER	106/	75	215/	85	481
		OD-INV	479/	313	1584/	90	2466

Table A.7: Total data transfer per profile in a producer/consumer experiment with switching devices (Section 7.4), in KB, for nodes B and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	Full Content	PRE-IMM	899/	1127	36433/	106837	145296
		PRE-PER	107/	138	5431/	13415	19091
		PRE-INV	237/	341	4339/	21712	26629
		OD-IMM	235/	419	1112/	2769	4534
		OD-PER	56/	89	1038/	2262	3445
		OD-INV	342/	471	1537/	3062	5411
	Metadata	PRE-IMM	926/	1199	36411/	109327	147863
		PRE-PER	109/	130	9317/	9235	18791
		PRE-INV	1271/	723	7469/	16346	25808
		OD-IMM	217/	330	1108/	1302	2957
		OD-PER	54/	72	138/	1152	1417
		OD-INV	203/	273	1158/	1880	3514
Node N	Full Content	PRE-IMM	1274/	1123	91661/	34118	128176
		PRE-PER	167/	141	14552/	4576	19436
		PRE-INV	237/	288	6196/	1669	8390
		OD-IMM	325/	258	96/	38	716
		OD-PER	42/	49	8/	1	100
		OD-INV	342/	191	214/	1	749
	Metadata	PRE-IMM	1386/	1173	105881/	26654	135093
		PRE-PER	157/	142	10256/	8238	18793
		PRE-INV	544/	703	10718/	1170	13135
		OD-IMM	301/	249	89/	9	647
		OD-PER	30/	43	7/	1	82
		OD-INV	181/	181	57/	51	469

Table A.8: Total data transfer per profile in a producer/consumer experiment with switching devices (Section 7.4), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Total
Full Content	OD-IMM	32196	6132	32884	2224	73436
	OD-INV	32368	7336	33880	2080	75664
	OD-PER	29536	3256	29580	156	62528
	PRE-IMM	60788	59400	59568	50368	230124
	PRE-INV	32612	35168	34460	3876	106116
	PRE-PER	32088	7436	31780	7352	78656
Metadata	OD-IMM	32516	3480	32692	1896	70584
	OD-INV	31992	5244	32896	1188	71320
	OD-PER	29492	1084	28676	156	59408
	PRE-IMM	62024	60212	60008	54852	237096
	PRE-INV	34864	32212	43920	10996	121992
	PRE-PER	31856	7212	32040	7120	78228

Table A.9: Total storage increase per profile in a producer/consumer experiment with switching devices (Section 7.4). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	Full Content	OD-PER-05	124/	333	0/	2532	2989
		OD-PER-10	89/	234	0/	2490	2814
		OD-PER-120	24/	55	0/	1717	1795
		OD-PER-30	62/	149	0/	2354	2565
		OD-PER-60	32/	77	0/	1825	1933
		PRE-PER-05	590/	1288	0/	179265	181144
		PRE-PER-10	403/	871	0/	133641	134914
		PRE-PER-120	41/	86	0/	12047	12173
		PRE-PER-30	140/	297	0/	44186	44623
		PRE-PER-60	70/	147	0/	22088	22305
	Metadata	OD-PER-05	140/	358	0/	143	640
		OD-PER-10	100/	254	0/	107	461
		OD-PER-120	23/	54	0/	17	95
		OD-PER-30	55/	139	0/	53	246
		OD-PER-60	33/	82	0/	29	144
		PRE-PER-05	590/	1289	0/	181949	183828
		PRE-PER-10	387/	833	0/	127319	128539
		PRE-PER-120	38/	80	0/	12046	12164
		PRE-PER-30	135/	289	0/	45177	45601
		PRE-PER-60	65/	136	0/	21083	21284
Node F	Full Content	OD-PER-05	396/	273	2481/	11	3161
		OD-PER-10	282/	170	2459/	0	2911
		OD-PER-120	77/	43	1714/	0	1834
		OD-PER-30	203/	113	2344/	3	2663
		OD-PER-60	103/	56	1821/	1	1982
		PRE-PER-05	1585/	1350	113358/	45888	162180
		PRE-PER-10	1097/	941	76216/	36475	114729
		PRE-PER-120	103/	80	7032/	3211	10426
		PRE-PER-30	364/	325	23905/	12664	37258
		PRE-PER-60	187/	165	11950/	4333	16634
	Metadata	OD-PER-05	406/	205	91/	0	703
		OD-PER-10	333/	189	67/	10	599
		OD-PER-120	75/	40	14/	0	129
		OD-PER-30	198/	117	45/	5	365
		OD-PER-60	110/	60	24/	1	195
		PRE-PER-05	1585/	1356	114655/	41992	159587
		PRE-PER-10	1043/	907	61880/	34092	97922
		PRE-PER-120	92/	84	7425/	3619	11221
		PRE-PER-30	358/	311	32196/	3365	36230
		PRE-PER-60	163/	144	10942/	5728	16978

Table A.10: Total data transfer per profile in an evaluation of synchronization frequency for periodic commits (Section 7.5), in KB, for nodes B and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	Full Content	OD-PER-05	315/	197	47/	0	559
		OD-PER-10	189/	128	27/	0	344
		OD-PER-120	27/	29	3/	0	59
		OD-PER-30	71/	80	10/	5	166
		OD-PER-60	35/	42	4/	2	83
		PRE-PER-05	1626/	1337	87059/	75104	165127
		PRE-PER-10	1124/	926	60641/	54650	117342
		PRE-PER-120	98/	91	7428/	1015	8632
		PRE-PER-30	384/	325	22092/	17980	40781
		PRE-PER-60	189/	168	8368/	8513	17239
	Metadata	OD-PER-05	265/	202	49/	5	521
		OD-PER-10	212/	185	48/	11	457
		OD-PER-120	26/	27	2/	0	55
		OD-PER-30	80/	63	9/	0	152
		OD-PER-60	42/	50	4/	4	100
		PRE-PER-05	1634/	1340	92148/	69102	164224
		PRE-PER-10	1046/	911	74493/	37581	114032
		PRE-PER-120	101/	85	6029/	5616	11831
		PRE-PER-30	366/	303	18843/	16019	35531
		PRE-PER-60	178/	148	9967/	7808	18101
Node N	Full Content	OD-PER-05	254/	287	46/	31	619
		OD-PER-10	165/	194	27/	23	409
		OD-PER-120	28/	29	2/	2	61
		OD-PER-30	79/	72	10/	1	161
		OD-PER-60	36/	37	4/	1	78
		PRE-PER-05	1557/	1382	132352/	32512	167804
		PRE-PER-10	1059/	946	104352/	16443	122799
		PRE-PER-120	103/	90	6029/	4216	10437
		PRE-PER-30	379/	324	33552/	4719	38974
		PRE-PER-60	207/	175	16755/	2139	19276
	Metadata	OD-PER-05	209/	258	45/	37	549
		OD-PER-10	149/	168	27/	13	357
		OD-PER-120	29/	35	2/	2	68
		OD-PER-30	64/	81	9/	6	160
		OD-PER-60	42/	38	4/	0	84
		PRE-PER-05	1564/	1390	123148/	37007	163109
		PRE-PER-10	1068/	895	80595/	17977	100535
		PRE-PER-120	99/	85	9435/	1608	11227
		PRE-PER-30	353/	309	23192/	9670	33524
		PRE-PER-60	165/	147	16141/	2431	18884

Table A.11: Total data transfer per profile in an evaluation of synchronization frequency for periodic commits (Section 7.5), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Total
Full Content	PRE-PER-05	72100	70464	70388	70216	283168
	PRE-PER-10	70992	52244	52168	52140	227544
	PRE-PER-30	70248	17644	17512	17484	122888
	PRE-PER-60	69996	8256	8180	9396	95828
	PRE-PER-120	69952	4740	4664	4664	84020
	OD-PER-05	71972	5648	1680	1632	80932
	OD-PER-10	71200	4912	988	948	78048
	OD-PER-30	70460	4188	364	360	75372
	OD-PER-60	70112	2792	152	144	73200
	OD-PER-120	69996	2372	88	80	72536
Metadata	PRE-PER-05	71980	71432	71356	71048	285816
	PRE-PER-10	71140	49856	49744	49728	220468
	PRE-PER-30	70192	17564	17504	17440	122700
	PRE-PER-60	70008	8268	8156	8128	94560
	PRE-PER-120	69968	4764	4664	4648	84044
	OD-PER-05	72120	3384	1660	1632	78796
	OD-PER-10	71136	2808	1552	920	76416
	OD-PER-30	70368	1992	328	332	73020
	OD-PER-60	70128	1116	168	164	71576
	OD-PER-120	70016	704	84	92	70896

Table A.12: Total storage increase per profile in an evaluation of synchronization frequency for periodic commits (Section 7.5). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	Full Content	PRE-IMM-6N	318/	783	0/	50103	51204
		PRE-PER-6N	23/	45	0/	2510	2577
		PRE-INV-6N	155/	373	0/	9540	10069
		OD-IMM-6N	223/	599	0/	2096	2918
		OD-PER-6N	20/	38	0/	505	563
		OD-INV-6N	147/	355	0/	2130	2632
		PRE-IMM-4N	205/	497	0/	30146	30848
		PRE-PER-4N	13/	27	0/	1507	1548
		PRE-INV-4N	116/	279	0/	10494	10889
		OD-IMM-4N	153/	397	0/	2058	2608
		OD-PER-4N	11/	25	0/	503	539
		OD-INV-4N	109/	266	0/	2083	2458
		PRE-IMM-2N	67/	165	0/	10049	10281
		PRE-PER-2N	4/	9	0/	502	515
		PRE-INV-2N	79/	198	0/	11952	12228
		OD-IMM-2N	74/	187	0/	2036	2296
		OD-PER-2N	5/	13	0/	502	521
		OD-INV-2N	75/	191	0/	2041	2307
	Metadata	PRE-IMM-6N	322/	791	0/	50208	51322
		PRE-PER-6N	24/	47	0/	2511	2581
		PRE-INV-6N	152/	366	0/	7538	8056
		OD-IMM-6N	259/	674	0/	128	1062
		OD-PER-6N	19/	39	0/	5	63
		OD-INV-6N	140/	341	0/	128	609
		PRE-IMM-4N	204/	496	0/	30146	30847
		PRE-PER-4N	12/	27	0/	1507	1547
		PRE-INV-4N	114/	274	0/	8494	8882
		OD-IMM-4N	146/	385	0/	57	588
		OD-PER-4N	11/	24	0/	3	38
		OD-INV-4N	108/	263	0/	84	454
		PRE-IMM-2N	67/	165	0/	10049	10281
		PRE-PER-2N	4/	9	0/	502	515
		PRE-INV-2N	77/	193	0/	9952	10221
		OD-IMM-2N	71/	180	0/	36	287
		OD-PER-2N	5/	12	0/	2	20
		OD-INV-2N	72/	183	0/	41	296
Node C	Full Content	PRE-IMM-6N	1001/	918	30886/	18131	50935
		PRE-PER-6N	59/	57	1304/	1102	2523
		PRE-INV-6N	379/	246	909/	306	1840
		OD-IMM-6N	593/	519	33/	1	1145
		OD-PER-6N	41/	40	2/	0	83
		OD-INV-6N	355/	203	109/	4	671
	Metadata	PRE-IMM-6N	1009/	894	29986/	19432	51322
		PRE-PER-6N	55/	56	1304/	1102	2518
		PRE-INV-6N	340/	257	601/	313	1511
		OD-IMM-6N	598/	514	33/	0	1146
		OD-PER-6N	37/	39	2/	0	78
		OD-INV-6N	329/	208	108/	6	651

Table A.13: Total data transfer per profile in an evaluation of scalability on number of nodes (Section 7.6), in KB, for nodes B and C.

		Profile	Overhead In/Out		Content In/Out		Total
Node D	Full Content	PRE-IMM-6N	874/	990	21772/	26848	50485
		PRE-PER-6N	55/	62	1305/	1103	2524
		PRE-INV-6N	334/	320	785/	431	1870
		OD-IMM-6N	553/	787	38/	54	1433
		OD-PER-6N	37/	50	2/	2	91
		OD-INV-6N	301/	283	84/	29	698
	Metadata	PRE-IMM-6N	873/	980	20771/	27849	50473
		PRE-PER-6N	53/	59	1205/	902	2219
		PRE-INV-6N	339/	295	592/	324	1551
		OD-IMM-6N	464/	604	38/	5	1111
		OD-PER-6N	40/	50	2/	2	94
		OD-INV-6N	283/	285	84/	31	682
Node F	Full Content	PRE-IMM-6N	835/	949	10050/	39812	51646
		PRE-PER-6N	53/	56	503/	2006	2618
		PRE-INV-6N	550/	418	8245/	1091	10303
		OD-IMM-6N	479/	599	2036/	20	3133
		OD-PER-6N	51/	54	502/	2	610
		OD-INV-6N	414/	346	2037/	88	2885
		PRE-IMM-4N	533/	447	16262/	13720	30962
		PRE-PER-4N	31/	27	803/	701	1562
		PRE-INV-4N	383/	196	10148/	345	11072
		OD-IMM-4N	389/	254	2036/	0	2679
		OD-PER-4N	30/	18	502/	0	551
		OD-INV-4N	296/	152	2038/	44	2529
		PRE-IMM-2N	165/	67	10049/	0	10281
		PRE-PER-2N	9/	4	502/	0	515
		PRE-INV-2N	198/	79	11952/	0	12228
		OD-IMM-2N	187/	74	2036/	0	2296
		OD-PER-2N	13/	5	502/	0	521
		OD-INV-2N	191/	75	2041/	0	2307
	Metadata	PRE-IMM-6N	831/	934	10050/	39608	51422
		PRE-PER-6N	51/	54	503/	1905	2512
		PRE-INV-6N	531/	382	6445/	693	8051
		OD-IMM-6N	522/	647	36/	38	1243
		OD-PER-6N	48/	49	2/	2	101
		OD-INV-6N	403/	318	36/	90	847
		PRE-IMM-4N	531/	446	15861/	14020	30859
		PRE-PER-4N	30/	25	803/	701	1560
		PRE-INV-4N	388/	216	8148/	345	9096
		OD-IMM-4N	383/	251	36/	0	670
		OD-PER-4N	30/	19	2/	0	51
		OD-INV-4N	300/	169	39/	44	552
		PRE-IMM-2N	165/	67	10049/	0	10281
		PRE-PER-2N	9/	4	502/	0	515
		PRE-INV-2N	193/	77	9952/	0	10221
		OD-IMM-2N	180/	71	36/	0	287
		OD-PER-2N	12/	5	2/	0	20
		OD-INV-2N	183/	72	41/	0	296

Table A.14: Total data transfer per profile in an evaluation of scalability on number of nodes (Section 7.6), in KB, for nodes D and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	Full Content	PRE-IMM-6N	1160/	622	42304/	6310	50396
		PRE-PER-6N	67/	39	2005/	401	2511
		PRE-INV-6N	258/	298	854/	260	1670
		OD-IMM-6N	713/	241	32/	2	989
		OD-PER-6N	50/	23	2/	0	74
		OD-INV-6N	232/	264	50/	61	607
		PRE-IMM-4N	512/	456	14953/	15529	31450
		PRE-PER-4N	29/	27	703/	802	1561
		PRE-INV-4N	140/	172	445/	123	880
		OD-IMM-4N	342/	422	33/	44	840
		OD-PER-4N	20/	27	2/	2	51
		OD-INV-4N	119/	132	50/	16	316
	Metadata	PRE-IMM-6N	1139/	644	42803/	6111	50697
		PRE-PER-6N	66/	39	2005/	301	2410
		PRE-INV-6N	241/	273	753/	461	1728
		OD-IMM-6N	752/	213	33/	0	998
		OD-PER-6N	49/	21	2/	0	71
		OD-INV-6N	248/	227	59/	53	588
		PRE-IMM-4N	511/	460	15654/	14628	31253
		PRE-PER-4N	29/	26	703/	802	1560
		PRE-INV-4N	145/	188	445/	123	902
		OD-IMM-4N	344/	427	33/	45	848
		OD-PER-4N	20/	27	2/	2	51
		OD-INV-4N	133/	142	45/	22	342
Node N	Full Content	PRE-IMM-6N	932/	857	42403/	6210	50402
		PRE-PER-6N	55/	53	2205/	201	2514
		PRE-INV-6N	297/	318	1075/	240	1930
		OD-IMM-6N	600/	415	34/	1	1049
		OD-PER-6N	38/	32	2/	0	72
		OD-INV-6N	276/	273	72/	40	662
		PRE-IMM-4N	583/	434	28781/	600	30398
		PRE-PER-4N	34/	26	1505/	0	1564
		PRE-INV-4N	153/	144	568/	200	1065
		OD-IMM-4N	424/	236	33/	0	693
		OD-PER-4N	24/	16	2/	0	41
		OD-INV-4N	130/	104	61/	6	301
	Metadata	PRE-IMM-6N	926/	857	44306/	4707	50796
		PRE-PER-6N	56/	51	2005/	301	2413
		PRE-INV-6N	245/	275	1076/	138	1734
		OD-IMM-6N	506/	449	33/	1	989
		OD-PER-6N	38/	34	2/	0	73
		OD-INV-6N	218/	243	66/	45	572
		PRE-IMM-4N	587/	431	28480/	1201	30699
		PRE-PER-4N	32/	25	1504/	0	1561
		PRE-INV-4N	191/	159	568/	200	1119
		OD-IMM-4N	421/	230	33/	0	684
		OD-PER-4N	25/	16	2/	0	43
		OD-INV-4N	134/	101	66/	1	302

Table A.15: Total data transfer per profile in an evaluation of scalability on number of nodes (Section 7.6), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Node C	Node D	Total
Full Content	PRE-IMM-6N	4880	6068	5996	6060	2639	2611	28254
	PRE-IMM-4N	4796	5920	5864	5876			22456
	PRE-IMM-2N	4828	6000					10828
	PRE-PER-6N	3472	392	372	380	121	118	4855
	PRE-PER-4N	3436	400	384	380			4600
	PRE-PER-2N	3500	396					3896
	PRE-INV-6N	4756	4384	1768	1836	458	461	13663
	PRE-INV-4N	4844	5140	1660	1764			13408
	PRE-INV-2N	4804	5940					10744
	OD-IMM-6N	4880	3856	3216	3132	809	769	16662
	OD-IMM-4N	4924	3880	3260	3268			15332
	OD-IMM-2N	4812	3828					8640
	OD-PER-6N	3412	384	200	200	34	30	4260
	OD-PER-4N	3472	392	208	208			4280
	OD-PER-2N	3548	384					3932
	OD-INV-6N	4844	3488	1640	1672	405	400	12449
	OD-INV-4N	4804	3876	1580	1644			11904
	OD-INV-2N	4832	4364					9196
Metadata	PRE-IMM-6N	4876	6020	5948	6008	2637	2597	28086
	PRE-IMM-4N	4860	6012	5944	5944			22760
	PRE-IMM-2N	4808	5964					10772
	PRE-PER-6N	3512	392	372	372	122	124	4894
	PRE-PER-4N	3512	400	384	380			4676
	PRE-PER-2N	3484	396					3880
	PRE-INV-6N	4824	4512	1768	1768	473	496	13841
	PRE-INV-4N	4780	5116	1684	1776			13356
	PRE-INV-2N	4828	5980					10808
	OD-IMM-6N	4824	3560	3212	3296	819	789	16500
	OD-IMM-4N	4920	3568	3224	3236			14948
	OD-IMM-2N	4808	3536					8344
	OD-PER-6N	3484	312	204	204	32	36	4272
	OD-PER-4N	3508	312	208	208			4236
	OD-PER-2N	3500	308					3808
	OD-INV-6N	4848	3168	1644	1624	386	381	12051
	OD-INV-4N	4764	3600	1636	1664			11664
	OD-INV-2N	4784	4060					8844

Table A.16: Total storage increase per profile in an evaluation of scalability on number of nodes (Section 7.6). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.

		Profile	Overhead In/Out		Content In/Out		Total
Node B	5 files	PRE-IMM	336/	741	0/	90534	91611
		PRE-PER	36/	75	0/	9022	9134
		PRE-INV	118/	259	0/	19783	20160
		OD-IMM	86/	236	0/	2436	2758
		OD-PER	21/	52	0/	1707	1780
		OD-INV	143/	331	0/	2468	2942
	50 files	PRE-IMM	2746/	5975	300/	885263	894284
		PRE-PER	542/	1134	0/	173029	174705
		PRE-INV	647/	1265	0/	146581	148493
		OD-IMM	461/	1196	0/	2587	4244
		OD-PER	148/	331	0/	2249	2728
		OD-INV	495/	968	0/	2791	4255
	90 files	PRE-IMM	6536/	10947	0/	1531914	1549397
		PRE-PER	1761/	3689	0/	512617	518068
		PRE-INV	1094/	2206	0/	238517	241817
		OD-IMM	765/	1965	0/	2725	5455
		OD-PER	256/	564	0/	2285	3105
		OD-INV	807/	1625	0/	3103	5535
Node F	5 files	PRE-IMM	909/	777	34030/	57169	92884
		PRE-PER	95/	81	3012/	6008	9195
		PRE-INV	518/	276	18556/	224	19574
		OD-IMM	287/	175	2428/	0	2889
		OD-PER	72/	36	1706/	0	1814
		OD-INV	330/	223	2440/	16	3008
	50 files	PRE-IMM	7469/	6356	319111/	549366	882302
		PRE-PER	1284/	1317	65161/	79657	147420
		PRE-INV	2385/	1641	117458/	9762	131246
		OD-IMM	1238/	803	2543/	1	4585
		OD-PER	378/	182	2247/	0	2808
		OD-INV	1107/	832	2653/	134	4725
	90 files	PRE-IMM	12931/	11722	563973/	857705	1446331
		PRE-PER	3503/	3812	189541/	79222	276077
		PRE-INV	3266/	2286	159421/	10656	175630
		OD-IMM	2008/	1411	2636/	44	6099
		OD-PER	606/	297	2282/	0	3185
		OD-INV	1607/	1281	2821/	249	5958

Table A.17: Total data transfer per profile in an evaluation of scalability on number of modified files (Section 7.7), in KB, for nodes B and F.

		Profile	Overhead In/Out		Content In/Out		Total
Node M	5 files	PRE-IMM	897/	787	60994/	30407	93085
		PRE-PER	94/	85	6810/	2710	9699
		PRE-INV	148/	218	726/	614	1707
		OD-IMM	193/	239	20/	31	482
		OD-PER	25/	33	2/	3	63
		OD-INV	313/	240	37/	27	617
	50 files	PRE-IMM	7327/	6441	623454/	240926	878148
		PRE-PER	1233/	1147	103987/	20237	126603
		PRE-INV	1158/	1269	28239/	11782	42449
		OD-IMM	894/	1039	95/	146	2174
		OD-PER	48/	68	4/	6	125
		OD-INV	502/	460	148/	87	1197
	90 files	PRE-IMM	12629/	11654	1050859/	318203	1393345
		PRE-PER	3310/	3121	198501/	431	205362
		PRE-INV	1747/	1819	41431/	2980	47978
		OD-IMM	1357/	1509	153/	171	3190
		OD-PER	70/	91	6/	10	177
		OD-INV	993/	756	291/	142	2182
Node N	5 files	PRE-IMM	938/	775	87090/	4003	92807
		PRE-PER	97/	81	8719/	800	9698
		PRE-INV	172/	201	1841/	502	2716
		OD-IMM	236/	151	20/	0	406
		OD-PER	28/	24	2/	0	54
		OD-INV	118/	110	35/	1	264
	50 files	PRE-IMM	7494/	6265	787836/	55147	856741
		PRE-PER	1635/	1097	106896/	3120	112747
		PRE-INV	1175/	1191	26472/	4044	32882
		OD-IMM	1086/	642	95/	0	1824
		OD-PER	56/	50	4/	0	111
		OD-INV	584/	427	224/	12	1247
	90 files	PRE-IMM	13418/	11192	1201892/	108902	1335404
		PRE-PER	4914/	2866	204780/	552	213112
		PRE-INV	1913/	1709	52023/	722	56366
		OD-IMM	1722/	968	152/	0	2842
		OD-PER	82/	62	6/	0	150
		OD-INV	888/	634	396/	14	1932

Table A.18: Total data transfer per profile in an evaluation of scalability on number of modified files (Section 7.7), in KB, for nodes M and N.

	Profile	Node B	Node F	Node M	Node N	Total
5 files	PRE-IMM	9356	10432	10340	10356	40484
	PRE-PER	7908	1336	1276	1280	11800
	PRE-INV	9356	6884	1244	1456	18940
	OD-IMM	9352	3184	1704	1708	15948
	OD-PER	7960	1068	200	200	9428
	OD-INV	9308	3836	3496	960	17600
50 files	PRE-IMM	77092	82144	82124	79016	320376
	PRE-PER	68052	16064	15996	15984	116096
	PRE-INV	75820	31152	10296	9388	126656
	OD-IMM	75116	12456	6844	6884	101300
	OD-PER	67884	6396	392	392	75064
	OD-INV	75864	13672	4728	4776	99040
90 files	PRE-IMM	136944	142700	142224	136156	558024
	PRE-PER	122472	44284	44204	44204	255164
	PRE-INV	135996	43300	15868	18460	213624
	OD-IMM	135120	19848	10420	10348	175736
	OD-PER	122744	10064	608	608	134024
	OD-INV	136924	19984	9280	7772	173960

Table A.19: Total storage increase per profile in an evaluation of scalability on number of modified files (Section 7.7). Each cell corresponds to the amount of storage utilization (in KB) that was added after the experiment was run with the specific scenario.