การพัฒนาเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลสำหรับอุปกรณ์เคลื่อนย้ายได้

นายสมิทธ์ ธรรมบำรุง

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2554

DEVELOPMENT OF PERSONAL CLOUD FILE SYSTEM FRAMEWORK FOR PORTABLE DEVICES

Mr.Smith Dhumbumroong

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2011

Thesis Title	DEVELOPMENT OF PERSONAL CLOUD FILE SYSTEM FRAME-WORK FOR PORTABLE DEVICES
By	Mr.Smith Dhumbumroong
Field of Study	Computer Science
Thesis Advisor	Assistant Professor Krerk Piromsopa, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

. . . . Dean of the Faculty of Engineering
(Associate Professor Boonsom Lerdhirunwong, Dr.Ing.)

THESIS COMMITTEE

. . . . Chairman
(Assistant Professor Natawut Nupairoj, Ph.D.)

. . . . Thesis Advisor
(Assistant Professor Krerk Piromsopa, Ph.D.)

. . . . Examiner
(Assistant Professor Kultida Rojviboonchai, Ph.D.)

. . . . External Examiner
(Pongtawat Chippimolchai, Ph.D.)

สมิทธ์ ธรรมบำรุง: การพัฒนาเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลสำหรับอุปกรณ์เคลื่อนย้ายได้. (DEVELOPMENT OF PERSONAL CLOUD FILE SYSTEM FRAMEWORK FOR PORTABLE DEVICES) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ.ดร.เกริก ภิรมย์โสภา, 89 หน้า.

งานวิจัยนี้ทำการนำเสนอเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคล โดยคุณลักษณะเด่นของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลอยู่ที่การนำเสนอแฟ้มข้อมูล (file) ต่างๆ ไม่ว่าจะเป็นแฟ้มข้อมูลที่อยู่ภายในเครื่องคอมพิวเตอร์เครื่องเดียวกันแต่ต่างโฟลเดอร์ (folder) หรือแฟ้มข้อมูลที่อยู่ในต่างเครื่อง ในลักษณะที่ผู้ใช้งานนั้นจะเห็นเสมือนกับว่าแฟ้มข้อมูลเหล่านี้นั้นอยู่รวมกันในเมาท์พอยต์ (mount point) ของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคล โดยเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลได้มีการใช้เทคนิกของระบบแฟ้มข้อมูลรวม (unification file system) เพื่อที่จะทำการรวมแฟ้มข้อมูลที่อยู่ในต่างโฟลเดอร์เข้าด้วยกัน นอกจากนี้เฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลถูกออกแบบมาให้สนับสนุนการทำงานแบบนอกสาย (disconnected operation) โดยระบบจะทำการแคช (cache) ไฟล์มาเก็บไว้เพื่อใช้งานในช่วงที่เครือข่าย (network) ใช้งานไม่ได้โดยอัตโนมัติ สถาปัตยกรรมของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลมีลักษณะเป็นแบบมอดูลาร์ (modular) สามารถที่จะปรับเปลี่ยนการทำงานและเพิ่มความสามารถให้กับระบบได้ผ่านทาง IO Module นอกจากนี้เฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลยังได้ทำการเพิ่มกลไกที่มีชื่อว่า Branch Tag ซึ่งเป็นกลไกที่ทำให้ผู้ใช้งานนั้นสามารถที่จะระบุแฟ้มข้อมูลและโฟลเดอร์ที่ต้องการที่จะใช้งานได้โดยตรง ผู้ใช้งานยังสามารถที่จะใช้ Branch Tag เพื่อแสดงแฟ้มข้อมูลที่ถูกระบบซ่อนระหว่างขั้นตอนการรวมโฟลเดอร์ได้อีกด้วย ระบบต้นแบบ (prototype) ของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลที่มีชื่อว่า Simple Protocol Agnostic File System 2 (SPAFS2) ได้ถูกพัฒนาขึ้นบนระบบปฏิบัติการลินุกซ์ (Linux) ผลจากการทดสอบการทำงานของระบบต้นแบบชี้ให้เห็นว่าระบบต้นแบบของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคลนั้นมีประสิทธิภาพในการทำงานเที่ยบเท่าและในบางกรณีก็สูงกว่าระบบแฟ้ม (file system) อื่นๆ ที่มีคุณสมบัติและความซับซ้อนน้อยกว่าระบบต้นแบบของเฟรมเวิร์คระบบแฟ้มคลาวด์ส่วนบุคคล

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิทยาศาสตร์คอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก	....
ปีการศึกษา	2554		

5170487021: MAJOR COMPUTER SCIENCE

KEYWORDS: DISTRIBUTED FILE SYSTEMS / PORTABLE DEVICES / DISCONNECTED OPERATION / FILE ORGANIZATION

SMITH DHUMBUMROONG : DEVELOPMENT OF PERSONAL CLOUD FILE SYSTEM FRAMEWORK FOR PORTABLE DEVICES. ADVISOR : ASST.PROF. KRERK PIROMSOPA, 89 pp.

This work describes Personal Cloud File System Framework, a modular userspace file system framework for accessing and manipulating data on multiple personal computers and portable devices. One of Personal Cloud File System Framework's unique characteristic that distinguishes it from other distributed file systems is the way it presents local and remote files and folders to the user. Personal Cloud File System Framework uses mechanisms similar to those employed by various unification file systems to present a virtually unified view of remote files and folders to the user. From the user's perspective, both local and remote files and folders from different machines appear as if they reside together locally in the Personal Cloud File System Framework's mount point. Personal Cloud File System Framework is also designed to support disconnected operation. When used, files are cached for offline usage automatically. When the cache is full, Personal Cloud File System Framework discards files using a replacement policy specified by the user. Personal Cloud File System Framework also implements Branch Tag, an extension of the fundamental concepts of unification file systems that enables the user to directly specify a file to operate on using the branch's name as well as shows files and folders that have been hidden by the framework during the process of unifying contents of multiple directories. Also, the modular nature of Personal Cloud File System Framework makes it trivial to modify or extend the framework via IO Module. Simple Protocol Agnostic File System 2 (SPAFS2), a prototype of Personal Cloud File System Framework, have been implemented on Linux operating system and preliminary benchmark results have shown that the prototype's performance is equal to and in some cases surpass other file systems that are less complex and have less features than SPAFS2.

Department:	Computer Engineering	Student's Signature	
Field of Study:	Computer Science	Advisor's Signature	
Academic Year:	2011		

Acknowledgements

This dissertation would not have been completed without the help of many people to whom I am forever indebted.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Krerk Piromsopa, who provided invaluable guidance and assistance throughout my time as a student at Chulalongkorn University. Dr. Krerk was always available and willing to help with any problem I was facing.

I would also like to express my utmost gratitude to my thesis committee: Dr. Natawut Nupairoj, Dr. Kultida Rojviboonchai, and last but not least Dr. Pongtawat Chippimolchai. They provided essential guidance, especially in finishing my dissertation, and their invaluable comments, suggestions, and criticisms improved the quality of my dissertation immensely.

During my time at Chulalongkorn University, I have been very fortunate to enjoy the advise, support, and encouragement of my friends and colleagues. Thank you all, my friends.

Finally, I would like to thank my father and mother who always believed in me. I could not have done it without their support.

Contents

List of Tables

List of Figures

CHAPTER I

INTRODUCTION

As the price-performance ratio of personal computers and portable devices continues to increase, it has become increasingly common for people to own more than one personal computer or portable device.

Sooner or later, the majority of people who own multiple machines will realize that managing their personal data that are scattered across multiple personal computers or portable devices is a slow, tedious, and error-prone task, especially if done so manually.

Over the years, various tools to help automate and simplify the task of managing personal data across multiple machines have been created. These tools can be roughly categorized into two groups: file synchronizers and distributed file systems.

A file synchronizer, such as Unison [1] or Dropbox [6], works by synchronizing contents of two or more folders so that in the end the contents of every folder that has been synchronized are identical. Most modern file synchronizers can synchronize both local and remote directories and can deal with conflict during synchronization process either automatically or manually through user intervention.

While using a file synchronizer can guarantee that every machine will always have the same set of files available, it also means that each machine must have enough storage space to store all the data that have been synchronized, which might not be possible especially on portable devices which usually have limited resources.

A distributed file system, such as NFS [14] or SSHFS [19], enables the user to instantly access and use remote files and folders transparently as if they are local ones without having to know about the real location of each file.

The downside of a distributed file system is that most distributed file systems require constant network connection, which renders most distributed file systems unsuitable for portable devices, which do not have constant network connection. While there are some distributed file systems that do support offline operation, also known as disconnected operation, such as Coda [15], they usually require complicated setup procedures as well as its own dedicate servers, which make them unsuitable for personal usage.

In this work, we propose Personal Cloud File System Framework, a modular userspace file system framework, as another approach to help manage personal data that are dispersed across multiple personal computers and portable devices. One of the unique characteristic of Personal Cloud File System Framework that sets it apart from the previously mentioned solutions is the way it presents files and folders from both local and remote machines to the user.

Personal Cloud File System Framework uses the same concepts as those used by unification file systems and presents a virtually unified view of files and folders from different locations to the user. From the user's perspective, all of these files and folders from various locations will appear as if they reside together, locally, inside the Personal Cloud File System Framework's mount point. Like most distributed file systems, the user can interact with remote files and folders transparently as if they are local ones.

The modular design of Personal Cloud File System Framework also makes it trivial to extend the framework to support new features such as a new network protocol, or modify how the framework functions.

We also aim to make Personal Cloud File System Framework supports offline operation. The framework should automatically caches files for offline usage and, once the cache is full, discards files according to replacement policy specified by the user.

1.1 Rational Behind the Name

The name was chosen because the ultimate goal of the framework that we have envisioned is to create the system that provides the benefits of cloud storage for personal usage.

By using Personal Cloud File System Framework, the user can combine storage device on all of his machines into one virtual storage device that he can write and read data to and from transparently without having to know about the actual location of the file. The user can also increase or decrease the overall size of the said virtual storage device by simply adding and removing machines from the union.

1.2 Problem Statement

Given the user who owns multiple personal computers or portable devices, how to make it easier for him to:

- Access, use, and manage his personal data that are dispersed across multiple machines.

- Create files on any machines of his choosing.

1.3 Objectives

This work have the following objectives.

1. Design Personal Cloud File System Framework
2. Implement the prototype of Personal Cloud File System Framework
3. Measure the performance of the Personal Cloud File System Framework's prototype and compare the result against other file systems

1.4 Scopes

The scopes of this dissertation is as follows.

1. Personal Cloud File System Framework is designed to manage personal data only.
2. The prototype of Personal Cloud File System Framework only supports Linux operating system.
3. For Network IO Module, we only implement and test SSH Network IO Module.
4. We only measure the Personal Cloud File System Framework's performance in local area network (LAN) environment.

1.5 Organization of the Dissertation

This dissertation is organized as follows. Section 2 discusses related work and we describe the design of Personal Cloud File System Framework in Section 3. The implementation details of the Personal Cloud File System Framework's prototype is given in Section 4. Section 5 presents and analyzes the prototype's benchmark results and we conclude in Section 6.

CHAPTER II

RELATED WORK

In this chapter, we describe distributed file systems and unification file systems. We begins by discussing prominent distributed file systems in Section 2.1. Section 2.2 describes notable unification file systems and we conclude with a brief summary in Section 2.3.

2.1 Distributed File Systems

A distributed file system enables the user to access and operate on remote files and folders transparently as if they are local ones. Distributed file systems have a long history, stretching back to the 1970s and early 1980s with systems such as Datacomputer [17], which supported an FTP-like service for clients which have limited amount of local storage, and Woodstock File Server [18] from XEROX PARC, which made it possible for the user to access single pages of a file. Other distributed file systems such as XDFS [9], LOCUS [13], and Swallow [16] were also developed during this period. The late 1980s and the early 1990s saw the creation of the three most influential distributed file systems in the academia: Sun's Network File System, Andrew File System, and Coda.

In this section, we describe the following prominent distributed file systems that are most closely related the our work: Sun's Network File System, Andrew File System, Coda, SSHFS, YaFS, and SSHFS-MUX.

2.1.1 Sun's Network File System

Sun's Network File System (NFS) [14], a distributed file system developed by Sun Microsystems in 1985, is one of the oldest distributed file system still being used today on UNIX and UNIX-like operating systems.

The primary reason for NFS's longevity and popularity is due to the fact that when the first version of NFS was released back in 1985, Sun disclosed the NFS protocol's specification to the public. Thanks to this, NFS is supported by many operating systems and is still being actively developed today. Also, the ubiquitous nature of NFS, especially on UNIX and UNIX-like operating systems, has led many to consider it to be a *de facto* standard.

NFS servers are stateless. Stateless servers do not store information about the state of client accesses to its files. The main benefit of stateless servers is that should one of the server crashes, no information is lost and the system can recover almost immediately.

However, stateless servers cannot control the concurrent accesses to its files. Since NFS's servers are stateless, this means that NFS cannot maintain the consistency of its file system and different clients can have different and conflicting copies of the same file or directory in their local cache. Also, as a consequences of having stateless servers, NFS is unable to perform files locking and atomic transactions. Another consequence of having stateless servers is the fact that it can take up to 60 seconds before a modification to the file system in one client is perceived by other clients.

2.1.2 Andrew File System

Andrew File System (AFS) [7], a distributed file system developed by researchers at Carnegie-Mellon University (CMU) during the 1980s, is one of the first distributed file system to focus on scalability. Its goal was to create a distributed file system that is capable of sharing data among thousands of machines. AFS achieves this goal by offloading most operations to clients, relying on clients to perform most of the works. When a client open a file, a part of the file are transferred from the server to the client. All subsequent operations are then performs on the local copy. Once the file is modified and closed, it is transferred back to the server.

Another problem that arise as a consequence of AFS supporting a large number of clients is security. To put it simply, with so many clients, it is impossible to trust all of them without compromising the security of the whole system. AFS solves the security problem by using the Kerberos protocol for clients and servers authentication.

AFS also supports replication and provides a location independent namespace which allows data to be moved transparently between servers. These two features can be used to perform load balancing and increase the system's fault tolerance.

2.1.3 Coda

Coda [15], a descendant of AFS, is one of the first major distributed file system to support disconnected operation. Using Venus, a cache manager, the user can set relative importance of files to ensure that his or her desired files will always be cached (or in Coda's terminology, hoarded) for offline usage.

During offline operation, the user can access and use files that have been cached by Venus. Once the connection to the servers is reestablished, Coda propagates changes made to the files during offline operation to the servers. If any conflicts should occur, such as modification to the same file by different clients, Coda provides tools that the user can use to decide how to best resolve the conflicts.

Like AFS, Coda also supports replication and use Kerberos protocol for clients and servers authentication.

2.1.4 SSHFS

Developed by Miklos Szeredi, who is also the author of FUSE [20], Secure Shell File System (SSHFS) [19] is a simple distributed file system implemented as a userspace file system using FUSE. It uses SSH protocol to access data on a remote machine.

While SSHFS lacks many features found in other major distributed file systems, such as replication or striping, it has a relatively advance cache mechanism that help improves its performance. SSHFS is also very simple to install and use. Most Linux distributions already come with SSH daemon installed and configured. Therefore, all the user needs to do in order to use SSHFS to access his data on remote computers which run Linux is to install SSHFS itself. Which is a simple thing to do on a modern Linux distribution that have a package management system. As such, SSHFS is particularly suitable for personal usage.

2.1.5 YaFS

YaFS [8] is an extensible distributed file system framework that supports multiple protocols, storage backends, and is extensible through plugin. YaFS also supports offline operation. However, the offline operation support in YaFS is rather limited. The user can only create new files and access files that have been created during offline operation. During offline operation, the user cannot access files stored on the servers.

Another unique characteristic of YaFS is that it stripes data into chunks before storing them on the servers using unique bandwidth-saving method. This also allows YaFS to store a file that is larger than the server allows. However, the fact that YaFS stripes its data also means that we cannot access the data stored on the servers without using YaFS.

2.1.6 SSHFS-MUX

SSHFS-MUX (SSHFS Multiplex) is one of the tools that are used to build GMount [4], an ad-hoc grid file system. SSHFS-MUX is basically a modified version of SSHFS that have been extended with the ability to unify multiple remote directories. All directories that are to be unified by SSHFS-MUX have Read-Write permission. As a consequence, SSHFS-MUX does not support Whiteout or Copy Up concepts of unification file systems. Since SSHFS-MUX is technically a fork of SSHFS, it also inherits SSHFS's cache mechanism that also help improves its performance.

The limitations of SSHFS-MUX is that it does not support offline operation and only supports SSH protocol.

2.2 Unification File Systems

In this section, we describe unification file systems. As its name implied, an unification file system unifies contents of two or more directories and presents a unified view to the user. The user can access and use files inside the mount point of the unification file system transparently, without having to know about the real location of the files.

We begin by describing 3 fundamental concepts that most modern unification file systems shared in Section 2.2.1 and then proceed to describe Plan 9's Union Directory in Section 2.2.2. 4.4BSD-Lite's Union Mount is presented in Section 2.2.3 and we describe Unionfs in Section 2.2.4.

2.2.1 Fundamental Concepts of Unification File Systems

Although each unification file system have its own unique set of features and capabilities, almost all of the major unification file systems share the following fundamental concepts in order to transparently unify contents of multiple directories and support features such as branch's permission.

2.2.1.1 Branch

Each directory that is going to be unified by an unification file system is called a branch. Each branch have a priority and permission associated with it.

Branch's priority is basically the order of each branch when they are supplied to the unification file system by the user. The first branch in the list of branches is usually referred to as the highest (or the top) branch, while the last branch in the list is referred to as the lowest branch.

Branch's priority is used to resolve any potential conflicts that might arise during file system operations. Specifically, branch's priority is used to decide which files or folders are to be presented to the user when there are files or folders with the same name in the same path on multiple branches.

Traditionally, when a conflict arises between two or more branches, files and folders from the higher branch are shown, while files and folders with the same name from branches which have lower priority are hidden. This leads to a consistent directory namespace where there are no duplicate files or folders.

Each branch also have a permission associated with it. The permission can be either one of: Read-Write (RW), which in most unification file systems is usually the default permission for every branch, and Read-Only (RO). Branch's permission is pretty much self-explanatory: they are used to control write access to each branch.

2.2.1.2 Whiteout and Opaque Directory

Whiteout and Opaque Directory are mechanisms that are used to handle the case when the user try to modify or delete files or folders in a branch that have Read-Only permission.

When the user deletes a file in the branch which have Read-Only permission, instead of returning an error message informing the user that the operation is not permitted and then abort the operation, a whiteout is created for that file instead. When the user list the contents of the directory, files that have whiteout created for them are hidden from the user's view, thus creating an illusion that the files have been deleted while in reality they still exist in their original location. Opaque Directory is the same concept, but apply to a dictionary instead of a file.

2.2.1.3 Copy Up

Copy Up is another mechanism that is used by unification file systems to deal with modification to files or folders in a Read-Only branch. When the user tries to modify a file in a Read-Only branch, the unification file system will copy that file up to the highest branch that have Read-Write permission, and then let the user modify the copy instead of the original file. This ensures that the branch's permission is preserved while at the same time lets the user modifies files in any branches that he wishes.

2.2.2 Plan 9's Union Directory

Developed by Bell Labs for research purposes and as the successor to UNIX, Plan 9 [12] is a distributed operating system with many innovative features. One of the unique feature of Plan 9 is the union directory. Plan 9 creates an union directory by merging multiple directories into one namespace.

Plan 9's Union Directory supports adding and removing directory to the top or bottom of an union directory. It is also unique in that it is one of the few unification file systems that does not hide duplicate files. If the user tries to access the file that have duplicates, the file that is in the first directory from the list of directories that have been merged by Plan 9 is chosen. Another unique characteristic of Plan 9's Union Directory is that it does not unify subdirectory.

Plan 9's Union Directory does not support branch's permission or Whiteout. It implicitly assumes that every branch have Read-Write permission. Plan 9's Union Directory also does not support Copy Up mechanism.

2.2.3 4.4BSD-Lite's Union Mount

4.4BSD-Lite's Union Mount [11], which was implemented on 4.4BSD-Lite operating system, merges directories and their contents to present a unified view. Like Plan 9, 4.4BSD-Lite's Union Mount supports dynamically adding or removing directory from either the top or bottom of the union. Unlike Plan 9, 4.4BSD-Lite's Union Mount unify subdirectory.

When a lookup operation is performed in a lower branch, 4.4BSD-Lite's Union Mount creates the same directory tree in the top branch, called a shadow directory. 4.4BSD-Lite's Union Mount does not support branch's permission. Unlike Plan 9's union directory, however, 4.4BSD-Lite's Union Mount assumes that all branches except the highest one have Read-Only permission. This means that all modifications to a file in lower branches will result in the said file being copied up to the top branch into its corresponding shadow directory. Deleting a file in lower branches will result in the creation of a whiteout for the said file that marks the file as deleted to Union Mounts.

Also, 4.4BSD-Lite's Union Mount only allows file systems that are derived from FFS to be the top branch.

2.2.4 Unionfs

Unionfs [25] is long considered by many to be a *de facto* standard for unification file system on Linux operating system. Unionfs was implemented using stackable file systems technique. Like 4.4BSD-Lite's Union Mount, Unionfs supports Whiteout, Copy Up, and subdirectory unification. Unionfs also allows lower branches to have either Read-Write or Read-Only permission.

Table 2.1: Unification file systems feature comparison.

Feature	**Plan 9's Union Directory**	**4.4BSD-Lite's Union Mount**	**Unionfs**
Recursive unification		✔	✔
Permission preservation on copy up			✔
Multiple writable branches	✔		✔
Dynamic insertion & removal of any branch			✔
Dynamic insertion & removal of the top branch	✔	✔	✔
No file system type restrictions	✔		✔
Creating shadow directories		✔	✔
Copy Up		✔	✔
Whileout		✔	✔
Operating systems supported	Plan 9	4.4BSD-Lite	Linux

Unionfs supports dynamic insertion and removal of any branch not only just the top and the bottom. Unionfs creates shadow directory only on write operations and errors. Like Plan 9's Union Directory, Unionfs does not have restriction on which file system can be used as the top branch. Also, unlike both Plan 9's Union Directory and 4.4BSD-Lite's Union Mount, Unionfs preserves ownership of each file that have been copied up.

Table 2.1 summaries and compares Unionfs's features against 4.4BSD-Lite's Union Mounts and Plan 9's Union Directory.

2.3 Summary

In this chapter, we describe prominent distributed file systems and unification file systems that are closely related to our work.

CHAPTER III

DESIGN

This chapter describes the design of Personal Cloud File System Framework. We begin by outlining the design's goals of Personal Cloud File System Framework in Section 3.1. The high-level overview of the framework is presented in Section 3.2. Section 3.3 describes system's components of Personal Cloud File System Framework. We then describe how the framework handles duplicate files and folders in Section 3.4. The next section, Section 3.5, describes how our Personal Cloud File System Framework handles modification to files and folders inside a branch that have Read-Only permission. The concept of Branch Tag is presented in Section 3.6 and we give a brief summary in Section 3.7.

3.1 Design's Goals

Our design's goals for Personal Cloud File System Framework are as follows.

- Create a framework that helps automate and simplify the management of dispersed personal data by utilizing techniques from unification file systems and distributed file systems to present a unified view of files and folders from multiple locations
- Create a framework that works on both personal computers and portable devices, the letter of which have limited resources and network connectivity
- Create a framework that provides support for multiple network protocols

In order to create a framework that works equally well on both personal computers and portable devices, our framework must support offline operation to ensure that the framework can continue to operate even when there is no network connection.

To support multiple network protocols, we design Personal Cloud File System Framework to be a modular framework in order to facilitate any attempts to modify or extend the framework.

3.2 High-level Overview

As shown in Figure 3.1, Personal Cloud File System Framework have a single client-multiple servers architecture.

Personal Cloud File System Framework does not have its own dedicated server. Instead,

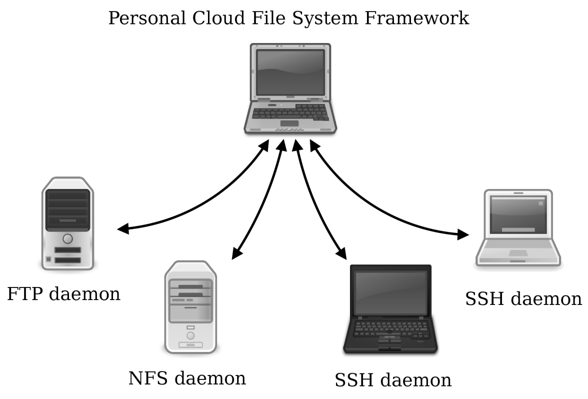

Figure 3.1: The architecture of Personal Cloud File System Framework

Personal Cloud File System Framework uses existing network protocols, such as FTP or SSH, to communicate with remote machines.

3.3 System's Components

Personal Cloud File System Framework have 3 main components: Interface, List Management and Cache Management. Figure 3.2 shows a block diagram that illustrates the relationship among components within the framework.

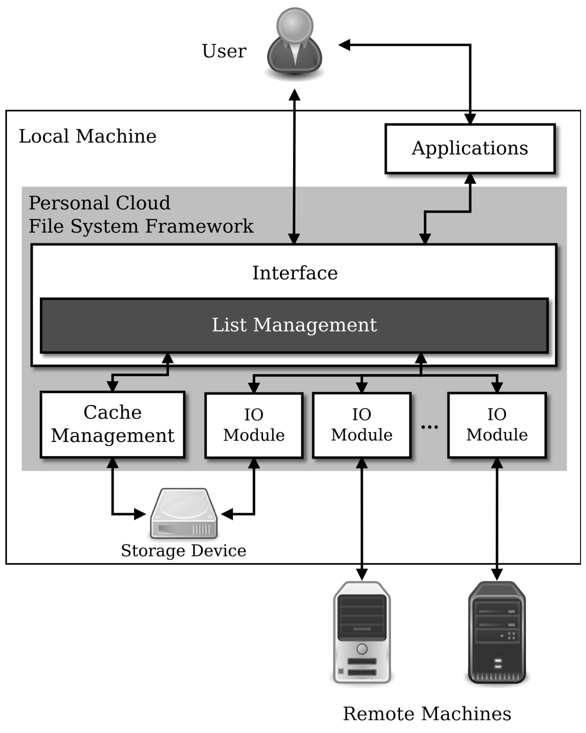

Figure 3.2: Components of Personal Cloud File System Framework

3.3.1 Interface

Interface component of Personal Cloud File System Framework provides standard file system interface that users and applications can interact with. Interface component is also responsible for passing file system calls invoked by the user or application to other components of the framework, such as List Management or Cache Management. Each component then operates on the received file system call and returns the result of the operation back to the user or application through Interface component.

3.3.2 List Management

List Management component stores and manages a list of branches that has been supplied by the user when the framework was mounted. The list of branches that List Management component manages also consists of other information about each branch, namely:

- Branch's priority
- Branch's protocol
- Branch's permission
  - Can be either Read-Write (RW) or Read-Only (RO)
- Path to directory

These branch's information must also be given by the user when mounting the framework.

List Management uses the list of branches and branch's information to conduct files or folders lookup and handle duplicate files or folders. List Management is also responsible for performing operations on either local or remote files or folders through IO Module, which is a self-contained sub-component of List Management that implements methods necessary to operate on either local or remote files and folders using various network protocols.

3.3.3 Cache Management

Cache Management component stores and manages files that have been locally cached. It caches files up to the amount specified by the user and uses replacement policy specified by the user to decide which file to discard when the cache is full. Cache Management cannot directly access local or remote files or folders. Instead, it uses List Management to access and modify both local and remote files and folders.

3.4 How the Framework Handles Duplicate Files or Folders

This section and the next describe how Personal Cloud File System Framework maintains its mount point's namespace consistency by using concepts of branch's priority, whiteout (opaque directory) and copy up, which are the same concepts used by unification file systems to make sure that there are no files or folders with duplicate name in its mount point. For more information on unification file systems' fundamental concepts, please refer to Section 2.2.1.

We start with the ideal case: when there are no duplicate files or folders name in the framework's mount point and thus no conflict. Figure 3.3 shows this ideal state.

In this case, as there are no files or folders with the same name in any of the branches, we can simply shows files and folders from all branches together in the framework's mount point.

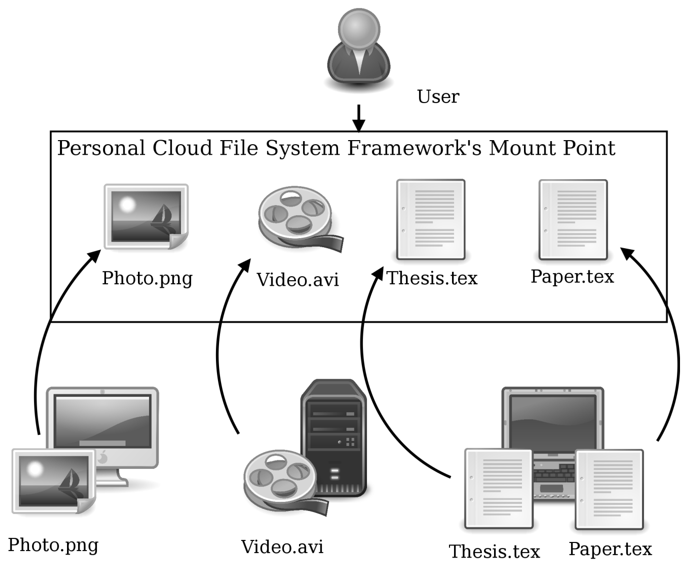

Figure 3.3: Personal Cloud File System Framework's mount point when there are no duplicate files or folders.

When two or more machines have files or folders with the same name as shown in Figure 3.4, where there is a file named `Photo.png` on two machines, in this case which file should Personal Cloud Figure System Framework shows in its mount point?

In order to make that decision, we need to have more data. The data we need is the branch's priority of each branch, implicitly supplied by the user when he mounts Personal Cloud File System Framework.

When the user mounts Personal Cloud File System Framework, he must give a list of branches that are to be unified by the framework. Usually, the first branch in the list of branches

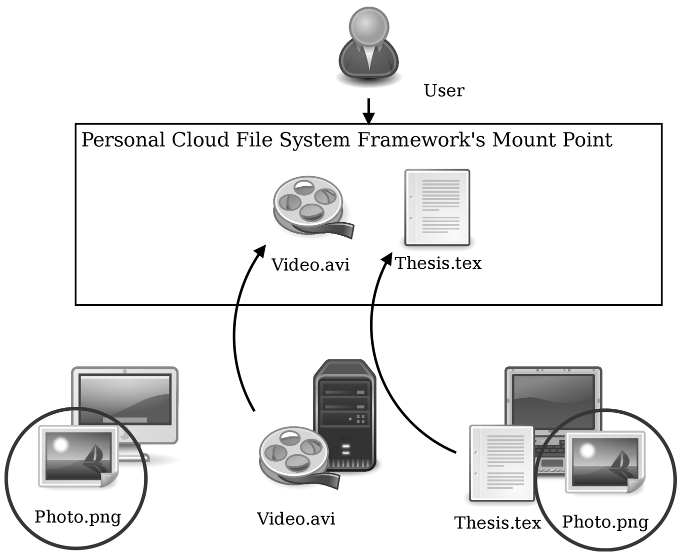

Figure 3.4: When two machines have files with the same name, which one should Personal Cloud File System Framework shows in its mount point?

that the user supplied to the framework is given the highest priority, the second branch in the list is given a lower priority than the first branch and so on.

Figure 3.5 shows how Personal Cloud File System Framework uses the concept of branch's priority to decide which duplicates to shows in the framework's mount point. As we can see from the figure, since the `Photo.png` file from the machine on the left handed side is in the 1st branch, which have higher priority than the 3rd branch, Personal Cloud File System Framework shows the `Photo.png` file from the machine on the left handed side in the framework's mount point and hides another file with the same name from the machine on the right handed side.

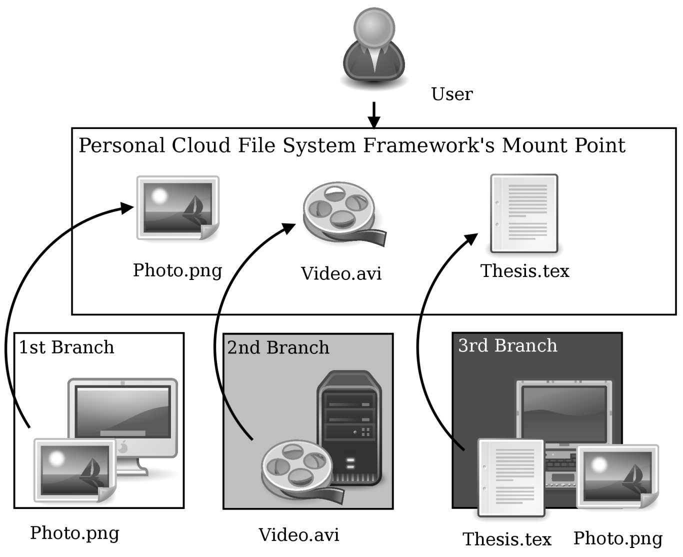

Figure 3.5: Using branch's priority to handle duplicate files

What if the user wishes to access the `Photo.png` file from the machine on the right handed side? In that case, he needs to unmount Personal Cloud File System Framework, rearrange priority of each branch so that machine on the right handed side is assigned with higher priority than that of machine on the left handed side, and then remount the framework.

As shown in Figure 3.6, once the user have rearrange each branch's priority so that machine on the right handed side is now the 1st branch, Personal Cloud File System Framework will shows the `Photo.png` file from machine on the right handed side and hides the same file from machine on the left handed side.

While this method works well enough, it is unfortunately very cumbersome. In Section 3.6, we present another method to view and operate on hidden files and folders in lower branches that does not require the user to remount the framework.

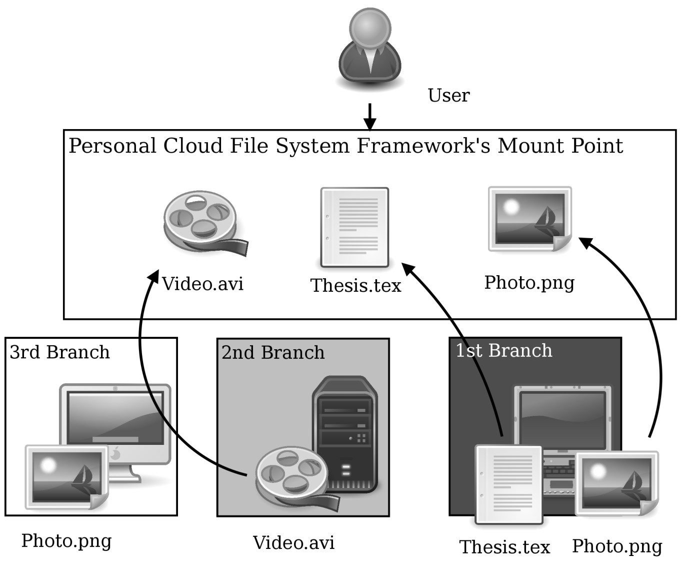

Figure 3.6: Alternative way of arranging branch's priority to resolve duplicate files.

3.5 How the Framework Handles Modification to Read-Only Branches

What if the user wishes to modify files or folders in branch that have Read-Only permission? We could just abort the operation and return error messages to the user, informing him that the operation is not permitted.

However, as the reader shall discover in this section, by applying concepts of whiteout and copy up from unification file systems, Personal Cloud File System Framework enables the user to modify files and folders inside a Read-Only branch as much as he wants while still respects each branch's permission.

3.5.1 Delete Files or Folders Inside Read-Only Branch

If the user attempts to delete a file inside a Read-Only branch, such as the `Video.avi` file in the 2nd Branch in Figure 3.7, instead of deleting the file as the user instructed, Personal Cloud File System Framework would create a whiteout for the given file.

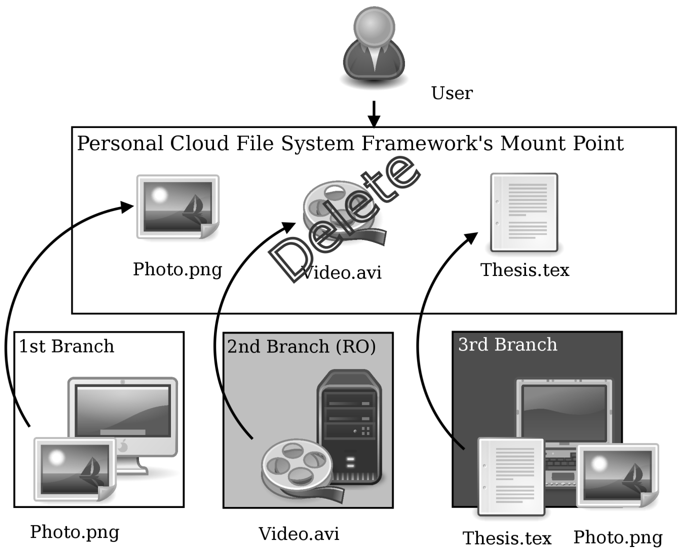

Figure 3.7: The user deletes the Video.avi file in the 2nd branch, which have Read-Only (RO) permission

Once the whiteout have been created for the `Video.avi` file, as shown in Figure 3.8, when the user lists contents of Personal Cloud File System Framework's directory tree, the framework will not show the `Video.avi` file or any other files which have whiteouts created for them.

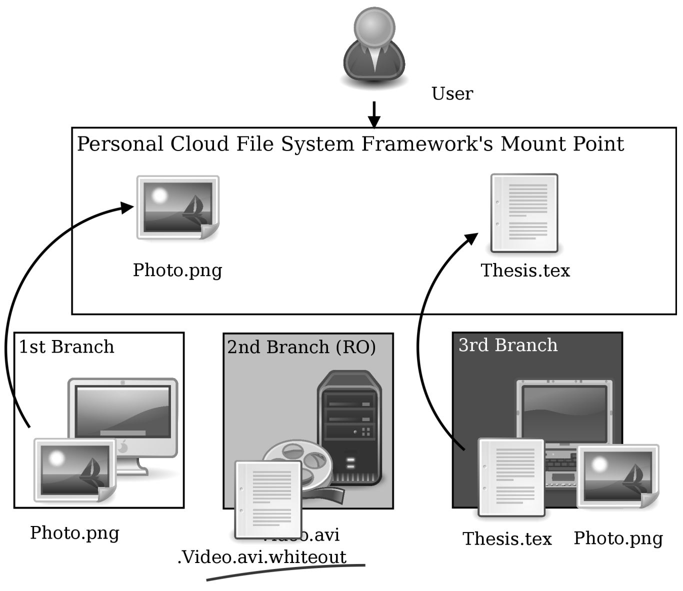

Figure 3.8: Personal Cloud File System Framework creates a whiteout for that file instead of deletes it

3.5.2 Modify Files or Folders Inside Read-Only Branch

If the user wishes to modify files inside a Read-Only branch, for example the `Thesis.tex` file in the 3rd branch in Figure 3.9, Personal Cloud File System Framework will copy the `Thesis.tex` file that the user wishes to modify up to the first branch of the framework, which always have Read-Write permission, and let the user modifies the copy instead of the original file. Figure 3.10 shows the result of a copy up operation.

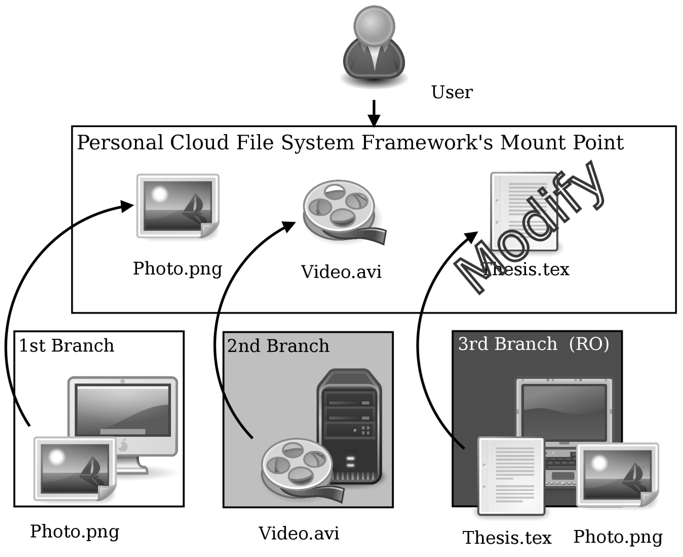

Figure 3.9: When the user modifies the Thesis.tex file in 3rd branch, which have Read-Only (RO) permission

As can be seen in Figure 3.10, Personal Cloud File System Framework lets the user modifies the copy instead of the original `Thesis.tex` file, thereby ensuring that each branch permission is observed.

3.6 Branch Tag

Branch Tag is Personal Cloud File System Framework's extension of the basic unification file system concepts. It is used to directly specify a file to operate on using the branch's name. It can also be used to show files and folders in lower branches that have been hidden by Personal Cloud File System Framework. Branch Tag was inspired by quFile's raw view [24]. In quFile, the user can access what is called a "raw view" of a file by referring to the file with the suffix `.quFile`, which will then show all current versions of the file. The user can then choose which version of the file he wish to access.

Branch Tag implements a similar functionality in Personal Cloud File System Framework. In Personal Cloud File System Framework, during a normal mode of operation, when there are files with the same name in more than one branches, we use branch's priority to control which file

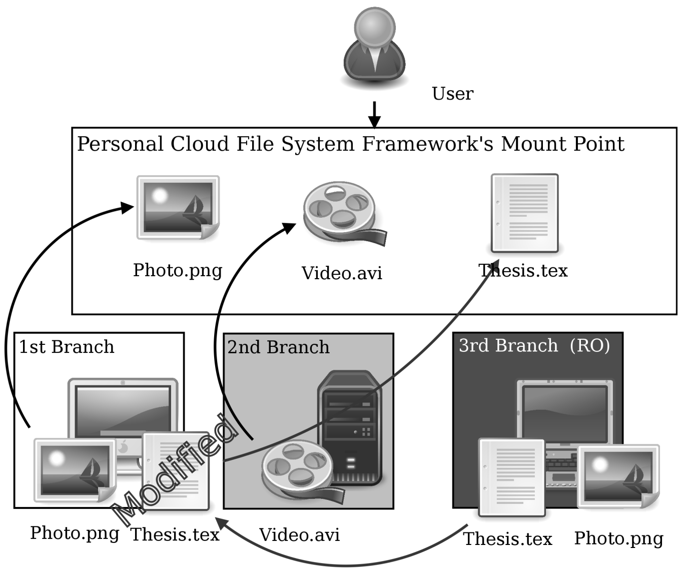

Figure 3.10: Personal Cloud File System Framework copy the file up to the highest branch, which always have Read-Write (RW) permission, before letting the user modify the copy of the file

to be hidden and which file to be shown. Figure 3.11 shows such a case, where the `Photo.png` file from the machine on the right handed side is hidden because it is in the lowest priority branch.

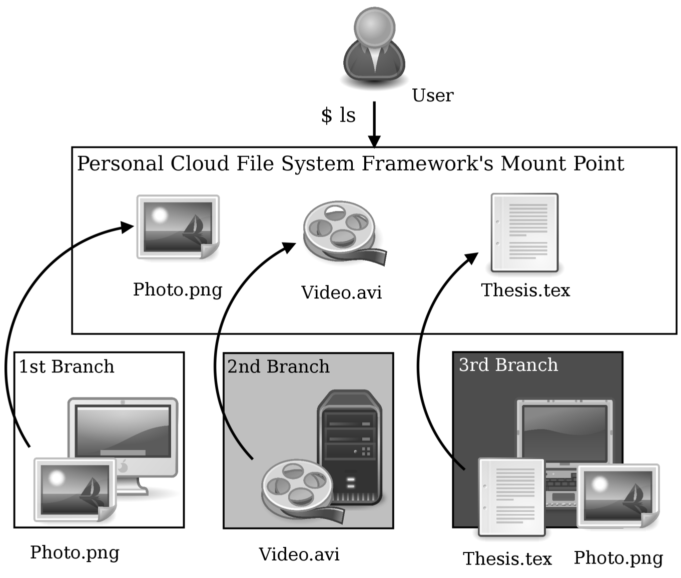

Figure 3.11: Personal Cloud File System Framework's mount point without Branch Tag

Now if the user passes the "`ls +raw`" command instead of the usual "`ls`" to the shell, Personal Cloud File System Framework will return contents of its mount point, but this time all files and folders inside the mount point will have branch's name appended to its name, as shown in Figure 3.12.

With this, the user not only know which branch a file belongs to, but can also see files and folders that have been previously hidden.

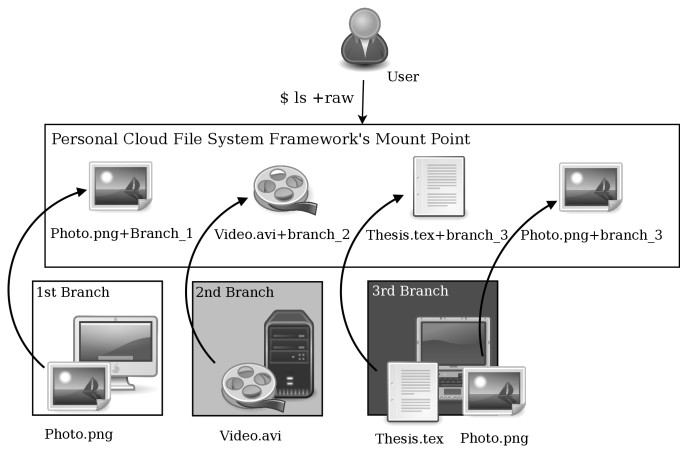

Figure 3.12: Personal Cloud File System Framework's mount point with Branch Tag enable

Now, if the user wishes to specify a file in a specific branch to operate on, he can do so by appending a plus sign (+) follows by the branch's name after the file name.

For example, as Figure 3.13 shows, to open the `Photo.png` file in the 3rd branch, the user simply have to refer to the `Photo.png` file as `Photo.png+branch_3`.

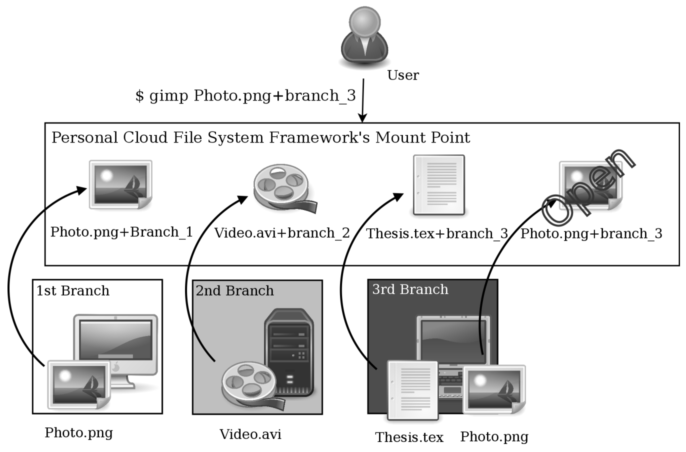

Figure 3.13: Using Branch Tag to access hidden files directly

We can also use Branch Tag to create a new file or folder on a specific branch. For example, if we wish to create a new file called `hello_world` on the 2nd branch, we can simply do so by using the following command.

```
$ touch hello_world+branch_2
```

In conclusion, Branch Tag allows the user to have greater control over how he can access or create data in our framework. Branch Tag is also one of the key mechanism that we planned to use to implement Cache Management component of Personal Cloud File System Framework in the future.

3.7 Summary

In this chapter, we describe various aspects of the design of Personal Cloud File System Framework, as well as how various components of the framework interact with one another. We also show how Personal Cloud File System Framework applies the concepts of unification file systems to solve problems that might arise during the process of unifying multiple directories. We also present Branch Tag, a mechanism that enables the user to easily specify a file to operate on using the branch's name as well as show files and folders in lower branches that have been hidden.

CHAPTER IV

IMPLEMENTATION

This chapter describes the implementation of Simple Protocol Agnostic File System 2 (SPAFS2), the prototype of Personal Cloud File System Framework. We begin by describing SPAFS2 and its various components in Section 4.1. We then describe how SPAFS2 parses branch's list in Section 4.2, follows by Section 4.3 which describes how SPAFS2 maintains its mount point's namespace consistency. Section 4.4 describes how SPAFS2 conducts files and folders lookup. Section 4.5 describes how SPAFS2 implements Branch Tag. Section 4.6 describes the limitation of SPAFS2 and we describe the past attempts at implementing SPAFS2's Cache Management component in Section 4.7. We conclude the chapter with a brief summary in Section 4.8

4.1 Simple Protocol Agnostic File System 2 (SPAFS2)

We use Python programming language [23] to implement Simple Protocol Agnostic File System 2 (SPAFS2), the prototype of Personal Cloud File System Framework. We choose to use Python because it facilitates rapid prototyping, has a very comprehensive standard library, and is cross-platform.

Similar to the design of Personal Cloud File System Framework, SPAFS2 consists of 2 components: Interface and List Management. The reason for the absent of Cache Management component in SPAFS2 is given in Section 4.7.

4.1.1 Interface

We implement the Interface component of SPAFS2 as a userspace file system, using Python binding of FUSE [20].

Table 4.1 shows the list of file system operation methods that SPAFS2's Interface component implements. Note that all link and symlink related methods as well as support for advance features such as access control list (ACL) are not implemented because most network protocol does not support them.

The file system operation methods shown in Table 4.1 also doubles as a basis for which the

Table 4.1: List of file system operation methods that SPAFS2 implements.

Method	Description
getattr	Get a file or folder attributes
readdir	Read the content of a directory
mkdir	Create a directory
rmdir	Remove a directory
read	Read a file
mknod	Create a new file node
write	Write to an existing file
rename	Rename a file or folder
truncate	Truncate a file
unlink	Delete a file
chmod	Change the permission bits of a file
chown	Change the owner and group of a file
utime	Change the access and/or modification times of a file
statfs	Get file system's statistics

interface of IO Module is derived from.

4.1.2 List Management

Currently, we implement List Management as a part of Interface component. This means that aside from providing file system interface to the user and application, Interface component, which is a userspace file system, also stores and manages list of branches and branch's information. Interface component is also responsible for other tasks such as maintaining mount point's namespace consistency, the creation of whiteouts, etc.

4.1.3 IO Module

The IO Module sub-component is implemented as a class inside a separate Python module that is imported and used by Interface component of SPAFS2.

Two IO Modules were created. The first one is Local Disk IO Module, which implements support for manipulating local files and folders. The second IO Module is SSH Network IO Module, which implements support for manipulating remote files and folders through SSH protocol using Paramiko library [10].

Table 4.2 presents the interface of IO Module. Aside from the file system operation methods, which are roughly the same as those that are implemented in Interface component of SPAFS2, there are 6 new methods that are unique to the IO Module.

The `connect` and `close_connect` methods deal with establishing connection with or terminating connection to a remote machine. The `path_exists` method is used to check whether the file or folder exists in a given path. The `makedirs` method is used to recursively create folders up to the leaf node of the given path. The `create_whiteout` method is, as its name implied, used to create a whiteout for a given file or folder and lastly, the `open` method is used to open a file and return the file object.

Table 4.2: IO Module's interface

Method	**Description**
connect	Establishing a new connection
close_connection	Close the existing connection
path_exists	Check if a path exists
makedirs	Recursively create directories up to the leaf node
create_whiteout	Create a whiteout for a given file or folder
open	Open and return a file object
lstat	Get a file or folder attributes
listdir	Read the content of a directory
mkdir	Create a directory
rmdir	Remove a directory
read	Read a file
mknod	Create a new file node
write	Write to an existing file
rename	Rename a file or folder
truncate	Truncate a file
unlink	Delete a file
chmod	Change the permission bits of a file
chown	Change the owner and group of a file
utime	Change the access and/or modification times of a file
statfs	Get file system's statistics

Each of SPAFS2's IO Module have its own root path that every file operations will originate from. Each IO Module can only have one root path. The root path can be on the local or remote machine.

Also, each IO Module have its own unique protocol associated with it.

4.2 How Simple Protocol Agnostic File System 2 Parses Branch's List

The user mounts SPAFS2 by giving a list of branches separated by a hash symbol (#) using the command similar to the following example.

```
$ spafs2 -o branch=PROTOCOL:ADDRESS:PERMISSION#... MOUNTPOINT
```

The order of the list of branches that the user gives here is important, as it is used by SPAFS2 to determine each branch's priority.

SPAFS2 then parses the list of branches that is given by the user and separates each item in the list of branches into the following components using a semicolon symbol (;) as a separator.

PROTOCOL Inform SPAFS2 which IO Module to use for this branch

ADDRESS Passed onto the IO Module as an argument

PERMISSION Branch's permission (optional)

- Can be either Read-Write (RW) (default) or Read-Only (RO)

The components of each item in the list of branches are then added to a Python's dictionary according to the order in which they are given. This dictionary is then later used to construct another dictionary called the `IO_modules` dictionary.

Each item of the `IO_modules` dictionary is a key-value pair where the key is the index number of each branch and the value is the tuple, containing the instant of the IO Module class as specified by the PROTOCOL component and the branch's permission as specified by the PERMISSION component. The instant of the IO Module class of for each branch is created using the ADDRESS component as an argument.

In this sense, each branch have its own instant of the IO Module class that are, in effect, isolated from each other. This means that we are able to use the same IO Module class in more than one branches.

The resultant `IO_modules` dictionary is the workhorse data structure of SPAFS2 and is used in every file system operations in SPAFS2.

4.3 Resolving Conflict and Maintain Namespace Consistency

As previously described in Section 3.4, It is inevitable that during the course of unifying contents of two or more directories, we will end up in a situation where the resultant unified directory contains files and folders with the same name. The act of trying to resolve the aforementioned situation so that in the end all files and folders in the mount point have a unique name is called resolving conflict or maintaining consistency of the file system's namespace.

The key insight we have gained during the development of SPAFS2 is that in order to maintain file system's namespace consistency and prevent any occurrence of conflicts during files lookup operation, we only need to keep the namespace of file system's root directory consistent and the rest will take care of themselves. In a sense, we are using the directory's structure of the file system's root directory itself to maintain consistency for all sub-directories.

Algorithm 1 Algorithm used in SPAFS2's readdir method for resolving conflict and maintaining namespace consistency.

```
dir_entries_list.append(".", "..")
for index, (branch_io_module, permission) in IO_modules do
    if path = "/" then
        branch_dir_entries_list ← branch_io_module.listdir(".")
    else
        branch_dir_entries_list ← branch_io_module.listdir(path)
    end if
    for entry in branch_dir_entries_list do
        if entry not in dir_entries_list then
            whiteout_file ← "." + entry + ".white_out"
            if not branch_io_module.path_exists(whiteout_file) then
                dir_entries_list.append(entry)
                return dir_entries_list
            end if
        end if
    end for
end for
```

To do so, we extend SPAFS2's `readdir` method to use Algorithm 1, which will only add a file or folder to the *dir_entries_list*, the list of files and folders to be shown to the user, if the said file or folder fulfills the following criteria.

1. Does not currently exists in the *dir_entries_list*
2. Does not have a whiteout created for it

Since the order in which files and folders get added to the *dir_entries_list* follows the order of the branches in *IO_modules* dictionary, we can guarantee that when there are files and folders of the same name in two or more branches the files and folders in the higher up branches will always be the ones that are shown to the user.

4.4 How Simple Protocol Agnostic File System 2 Conducts Files and Folders Lookup

When the user performs operations on a file or folder, SPAFS2 performs a check on each branch, starting from the first one all the way down to the last, to see which branch contains the file or folder that the user wishes to operate on. Once the branch that contains the file or folder is found, SPAFS2 terminates the loop and operates on the file or folder.

Algorithm 2 which is used in SPAFS2's `read` method is one such example. It received a path to file from the user, iterates over each branch in `IO_modules` dictionary, checking with

Algorithm 2 Algorithm used in SPAFS2's read method

```
1: for $index$, ($branch\_io\_module$, $permission$) in $IO\_modules$ do
2:   if $branch\_io\_module.path\_exists(path)$ then
3:     $branch\_io\_module.read(path)$
4:     return
5:   end if
6: end for
```

each branch's instant of IO module class to see which one contains the path to file. Once the match is found, SPAFS2 performs a read operation on the file in the branch, and then exit the loop.

Using this algorithm, we can make sure that the read operation will always occur on the file that have been shown to the user without the need to implement any other checking mechanisms.

Algorithm 3 Algorithm used in SPAFS2's mknod method

```
1: $dir\_component$ $\leftarrow$ extract_dir_component($path$)
2: if $dir\_component$ =“/” then
3:   $IO\_modules$[1st_branch].io_module.mknod($path$)
4: else
5:   for $index$, ($branch\_io\_module$, $permission$) in $IO\_modules$ do
6:     if $branch\_io\_module$.path_exists($path$) then
7:       if $permission$ = “rw” then
8:         $branch\_io\_module$.mknod($path$)
9:         return
10:      else
11:        if not $IO\_modules$[1st_branch].io_module.path_exists($dir\_component$) then
12:          $IO\_modules$[1st_branch].io_module.makedirs($dir\_component$)
13:        end if
14:        $IO\_modules$[1st_branch].io_module.mknod($path$)
15:        return
16:      end if
17:    end if
18:  end for
19: end if
```

Similar algorithm is also used when creating a new file or folder. Algorithm 3, which is used in SPAFS2's `mknod` method is one such algorithm. It checks if the directory component of the path supplied by the user is the root path or not. If it is the root path, then SPAFS2 defaults to creating the new file in the first branch.

If the path given is not the root path, SPAFS2 performs a check similar to one used in Algorithm 2 but instead of finding the exact path to file, it finds which branch contains the directory component of the path and then create the new file there.

Lastly, if it turns out that the branch that contains directory component have Read-Only permission, then SPAFS2 will use the copy up concept and creates the new file, along with the directory component if there is any, in the first branch.

This also means that in SPAFS2, the first branch always have Read-Write permission.

Algorithm 4 Algorithm used in SPAFS2's rmdir method

```
for $index$, ($branch\_io\_module$, $permission$) in $IO\_modules$ do
    if $branch\_io\_module$.path_exists($path$) then
        if $permission$ = "rw" then
            $branch\_io\_module$.read($path$)
            return
        else
            $branch\_io\_module$.create_whiteout($path$)
            return
        end if
    end if
end for
```

Algorithm 4, which is used in `rmdir` method and is derived from Algorithm 2, is an example of the algorithm that handles whiteout creation. It basically works the same way as Algorithm 2 does, but includes the if statement that checks the permission of the branch. If the branch that contains folder that the user wishes to delete have Read-Only permission, then instead of deleting the file, a whiteout is created for it.

Algorithm 5 Algorithm used in SPAFS2's write method

```
for $index$, ($branch\_io\_module$, $permission$) in $IO\_modules$ do
    if $branch\_io\_module$.path_exists($path$) then
        if $permission$ = "rw" then
            $branch\_io\_module$.write($path$)
            return
        else
            $dir\_component \leftarrow$ extract_dir_component($path$)
            if not $IO\_branches$[1st_branch].io_module.path_exists($dir\_component$) then
                $IO\_branches$[1st_branch].io_module.makedirs($dir\_component$)
            end if
            $src\_path \leftarrow branch\_io\_module$.open($path$)
            $dst\_path \leftarrow IO\_modules$[1st_branch].io_module.open($path$)
            copyfile($src\_path$, $dst\_path$)
            $IO\_modules$[1st_branch].io_module.write($dst\_path$)
            return
        end if
    end if
end for
```

Another type of algorithm that is also derived from Algorithm 2 is the algorithm that supports Copy Up when we try to modify the existing file in a Read-Only branch.

Algorithm 5 is an example of such algorithm. It checks to see which branch contains the file that the user wishes to write to and if the permission of the branch is Read-Write allows the user to write the file. If the permission of the branch is Read-Only then the algorithm copies the file up to the first branch, and then lets the user writes to the copy instead of the original file.

4.5 How Simple Protocol Agnostic File System 2 Implements Branch Tag

As previously described in Section 3.6, Branch Tag is a mechanism that enables the user to directly specify a file to operate on using the branch's name by appending a plus sign (+) follows by the branch's name to the file name. For example, the following command deletes the `rc.conf` file in the 4th branch.

```
$ rm rc.conf+branch_4
```

Implementing Branch Tag mechanism in SPAFS is a pretty straight forward process. Algorithm 6 shows the algorithm we used to implements Branch Tag functionality in SPAFS2's `read` method.

Algorithm 6 Branch Tag algorithm used in SPAFS2's read method

```
1: if $branch\_tag$ then
2:     $splited\_path \leftarrow path$.split("+")
3:     if length_of($split\_path$) $> 1$ then
4:         $branch\_name \leftarrow splited\_path$[-1]
5:         $IO\_modules[branch\_name]$.io_module.read($path$[0])
6:         return
7:     end if
8: end if
```

The algorithm begins by checking whether the user enables Branch Tag functionality. If the user enables Branch Tag, then the algorithm splits all paths that have been given by the user using the plus sign (+) as a separator and stores the resultant list into the $splited\_path$ variable. The algorithm then checks the length of the $splited\_path$ list. If the length of the list is greater than 1, then the algorithm uses the last item in the $splited\_path$ list as the branch's name and reads data from the given path using the instant of IO Module class from the branch that have the same name as the one that user specified.

By using this algorithm, we can implements Branch Tag support into SPAFS2's `read` method without having to modify other part of the `read` method's code. We implements Branch Tag into other SPAFS2's methods using algorithms that are similar to Algorithm 6.

Another use of Branch Tag is to show files and folders in lower branches which have been hidden because they have the same name as the files and folders in the higher branches. Again, we extend SPAFS2's `readdir` method to support this feature of Branch Tag by using another variation of Algorithm 6 that checks if the user passes `+raw` as an argument for a `ls` command. If the user uses the command "`ls +raw`" to list contents of a directory in SPAFS2 instead of the normal "`ls`" command, SPAFS2 will append branch's name to each file or folder's name before adding them to the $dir\_entries\_list$ variable.

This method not only show which file or folder belongs to which branch, it also shows all files and folders in lower branches that have been hidden during the normal mode of operation. This is because with branch's name appended to every file and folder's name in every branch, all files and folders' name are now unique and thus SPAFS2 can show the contents from all branches without having to worry about duplicate files or folders.

4.6 Simple Protocol Agnostic File System 2's Limitation

One limitation of SPAFS2 worth mention is that the rename operation only works in a certain circumstance. Specifically, it only works on files and folders inside the same branch. This limitation arises due in part mostly to the difficulty of moving files and folders from one remote branch to either a local or another remote branch.

4.7 Attempts at Implementing a Functional Cache Management Component in Simple Protocol Agnostic File System 2

After successfully implemented Interface component, List Management functionality, as well as 2 proof-of-concept IO Modules, the next logical step in the development of SPAFS2 is to try and implement Cache Management component.

The first attempt at implementing Cache Management component in SPAFS2 tries to implement it as a part of Interface component, like what we did with List Management component. However, we quickly ran into problems with this approach.

The first problem we had was the performance impact when caching large files. This problem should be easily solved by creating a new thread to handle file caching in the background, but then we ran into another issue in that creating a new thread in Python FUSE often leads to race condition and then deadlock.

Another solution that we have tried was starting a new process to handle files caching, but we ran into yet another problem here. Basically, after a connection to remote machine has been established, any attempts to pass SFTP objects, such as file objects, between processes will result in Paramiko throwing exceptions. After further investigation, we have found that this is the known limitation of Paramiko library.

In conclusion, with the current implementation of Interface component using Python FUSE, it is unlikely that we can implement Cache Management component as a part of Interface component the same way we had done with List Management component largely because of performance impact that caching large files will incur as well as other issues that prevented us from creating a new thread or process to cache files.

Due to time constraints, we are unable to investigate other approaches to implement Cache Management component for SPAFS2. But from what we have known so far, it seems to us that in order to avoid all of the aforementioned problems, Cache Management component of SPAFS2 needs to be a separate process from Interface component that handles files caching through Interface component just like any other applications.

This also means that for Cache Management component as a separate process from Interface component to work, SPAFS2 itself needs some mechanism to specify which branch to write data to. That mechanism is, of course, the Branch Tag which we have described in Section 3.6 and implemented in SPAFS2 as shown in Section 4.5.

4.8 Summary

In this chapter, we present Simple Protocol Agnostic File System 2 (SPAFS2), the prototype of Personal Cloud File System Framework. We also shows how various components of SPAFS2 are implemented and what tools were used to implement them. Various algorithms that were used in SPAFS2 are also described and we explain the problems we faced when trying to implement the Cache Management component of SPAFS2.

CHAPTER V

PERFORMANCE EVALUATION

In this chapter, we measure and analyze the performance of Simple Protocol Agnostic File System 2 (SPAFS2), the current prototype of Personal Cloud File System Framework, against other file systems. We begin by describing our experimental setup in Section 5.1. The benchmark tool used in the experiment is presented in Section 5.2. We present and analyze the benchmark results in Section 5.3 and we conclude the chapter with a summary in Section 5.4.

5.1 Experimental Setup

We use two machines in the experiment.

- A desktop machine with 2.0GHz AMD Athlon 64 3200+ processor, 2GB of RAM, and 250GB SATA hard disk drive formatted with the ext4 file system.
- An ASUS Eee PC model 901 with 1.6GHz Intel Atom N270 processor, 1GB of RAM, and two solid state drives, one 4GB and another 16GB. Both drives are formatted with the ext4 file system.

Both machines run Arch Linux and were connected wirelessly through an 802.11b/g wireless router. Figure 5.1 shows network diagram of the experimental setup.

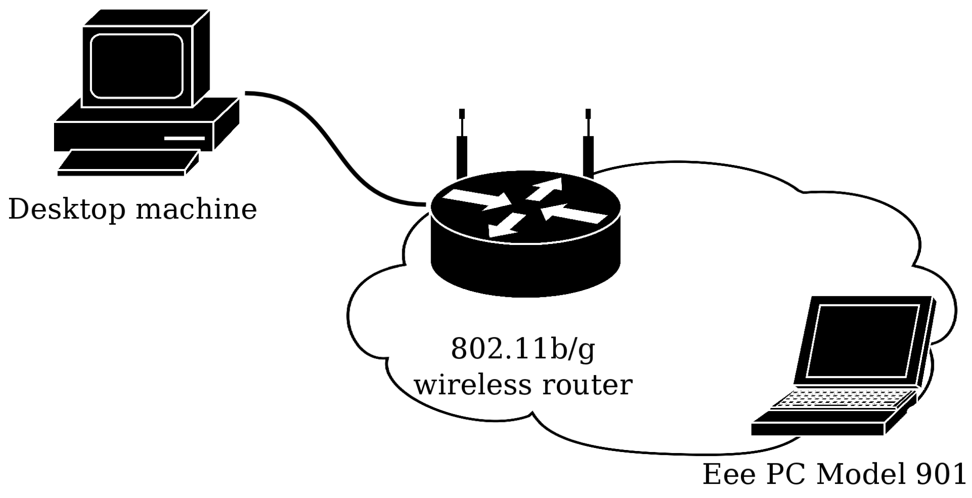

Figure 5.1: Network diagram of the experimental setup.

SPAFS2 as well as other file systems that are being compared against it are deployed on the netbook, while the desktop simply acts as a server, running SSH daemon.

We divide the experiment into two parts. The first part of the experiment compares SPAFS2's Local Disk IO module performance against the following file systems.

- ext4
- DummyFS
- SPAFS
- PyUnionFS

The ext4 or fourth extended file system is a journaling file system for Linux operating system. It is the successor to the ext3 file system and supports features such as extents and delayed allocation. The ext4 file system also has an improved design and better performance compared to the ext3 file system. Most major Linux distributions of today, such as Ubuntu and Fedora, use the ext4 file system as the default file system [2, 22].

For more details on DummyFS, SPAFS and PyUnionFS, please consult Section A.1, Section A.2 and Section A.3, respectively.

The second part of the experiment compares SPAFS2's SSH Network IO module performance against the following distributed file systems.

- SSHFS
- PySSHFS

Secure Shell File System (SSHFS) [19] is a distributed file system that uses SSH protocol to communicate with remote computers. For more details on SSHFS, please refer to Section 2.1.4

More information on PySSHFS can be found in Section A.4.

5.2 Benchmark Tool

We choose to use Filebench [21] because it is a very flexible and versatile benchmark tool, capable of generating various different kinds of workloads via workload personality.

For our experiment, we use the following pre-defined micro-benchmark workload personalities:

Create Files Create 10,000 files, write up to 16KB of data and then close all of them.

Random Reads Perform random reads on a 1GB file for 60 seconds.

Random Writes Perform random writes on a 1GB file for 60 seconds.

Delete Files Delete 10,000 files. Each file is no larger than 16KB.

All of the workloads were run on the netbook.

5.3 Results and Analysis

Section 5.3.1 presents and analyzes the benchmark results of SPAFS2's Local Disk IO Module, while Section 5.3.2 presents and analyzes the benchmark results of SPAFS2's SSH Network IO Module.

5.3.1 Benchmark Results and Analysis of SPAFS2's Local Disk IO Module Performance

As we can see from the results in Table 5.1, SPAFS2 performance in general is not only on par with DummyFS but actually surpasses it in few types of workloads, namely Creates Files and Random Writes. It also outperforms its predecessors, SPAFS and PyUnionFS, despite having more complex code base, modular architecture, and more features.

Table 5.1: Micro-benchmark results of SPAFS2's Local Disk IO Module and other file systems

		Create Files	Random Reads	Random Writes	Delete Files
ext4	ops/sec	749.483	1351.467	398.140	526.249
	latency (ms)	55.4	0.5	2.3	29.8
DummyFS	ops/sec	173.237	491.753	91.115	227.240
	latency (ms)	274.3	1.8	10.3	68.7
SPAFS	ops/sec	4.665	178.439	109.523	2.710
	latency (ms)	10274.1	5.3	8.8	5901.1
PyUnionFS	ops/sec	105.9	475.4	102.2	103.1
	latency (ms)	451.2	1.8	9.5	153.0
SPAFS2	ops/sec	249.7	476.1	315.4	188.7
	latency (ms)	189.5	1.8	2.8	83.5

The reason that SPAFS2 out performs SPAFS is simple: SPAFS2 does not need to use database in order to function, while SPAFS was designed from the ground up to store branch's data in the database and uses these data inside the database to conduct files lookup. This design choice results in every file system operations invoking at least one database query. This of course leads to terrible performance even during file system operations that only involve reading data from the database and abysmal performance during file system operations that involve writing data to the database, where not only does SPAFS have to write data to files in the file system, it also have to compute and add or update metadata of each and every files that have been written to the database.

In contrast, SPAFS2 learns from the mistake made by SPAFS and abandons the use of database to conduct files lookup altogether. Instead, SPAFS2 relies on using only built-in data structures of Python, specifically list and dictionary, to store data necessary to conduct files lookup. This results in SPAFS2 easily performs 2 to 60 times faster than SPAFS, as the benchmark results clearly show.

In the case of PyUnionFS, we believe that SPAFS2 performs better than PyUnionFS because PyUnionFS operates on a list of paths, while SPAFS2 operates on a list of objects inside a dictionary. This might sound trivial, but it means that for every file operations PyUnionFS have to perform more works than SPAFS2. This is because while SPAFS2 can use paths that are supplied by the user or application instantly, PyUnionFS have to parse these paths first before it can operate on them. The overhead from these individual parsing operations might be insignificant, but combined they can really have an impact on the performance of PyUnionFS as the benchmark results have shown.

Lastly, as to the reason why SPAFS2's performance is on par with, and even in some cases surpasses that of DummyFS, we do not believe that the current benchmark results provide enough data to answer this question. More in-dept profiling of both DummyFS and SPAFS2 is needed before we can understand why the performance of SPAFS2, with all of its features and modular architecture is on par with that of DummyFS, a bare-bone loopback file system.

5.3.2 Benchmark Results and Analysis of SPAFS2's SSH Network IO Module Performance

From Table 5.2, the results clearly show that in general SPAFS2 performs worse than both SSHFS and PySSHFS.

Table 5.2: Miro-benchmark results of SPAFS2's SSH Network IO Module and other distributed file systems

		Create Files	Random Reads	Random Writes	Delete Files
SSHFS	ops/sec	223.6	56.061	316.3	232.5
	latency (ms)	202.2	17.5	2.8	67.4
PySSHFS	ops/sec	25.7	14.3	33.3	79.4
	latency (ms)	1860.0	69.5	29.7	199.7
SPAFS2	ops/sec	17.8	13.3	28.0	39.5
	latency (ms)	2692.4	74.6	35.4	402.6

This is as we have expected, because unlike SSHFS or PySSHFS, SPAFS2 must loop over a list of objects inside a dictionary and checking which branch the given file or folder is in before it can performs operation on the said file or folder. While the overhead of each check is quite small, they eventually added up and affect the performance nonetheless.

5.4 Summary

In this chapter, we present the benchmark results of SPAFS2 under various different kinds of micro-benchmark workloads. The benchmark results clearly show that SPAFS2 performance is acceptable, especially when we consider the complexity of SPAFS2's code base as well as the fact that SPAFS2 have modular architecture and more features than most of the other file systems that are being compared against it.

However, as the reader might have already noticed, the benchmark workloads used in the experiment are only micro-benchmark workloads, which only show the overhead of various file system operations and not the indication of overall file system performance.

The reason for this omission is that while there are many general-purpose benchmarks to choose from, and even Filebench itself can functions as one if we use the appropriate workload personality, the problem is that most general-purpose benchmarks does not generate workload that accurately reflexes how people would use their personal computer or portable device. Therefore, we decided to omit using a general-purpose benchmark to evaluate SPAFS2 until we can find a general-purpose benchmark that can generates workload that accurately reflexes how layperson uses their machine.

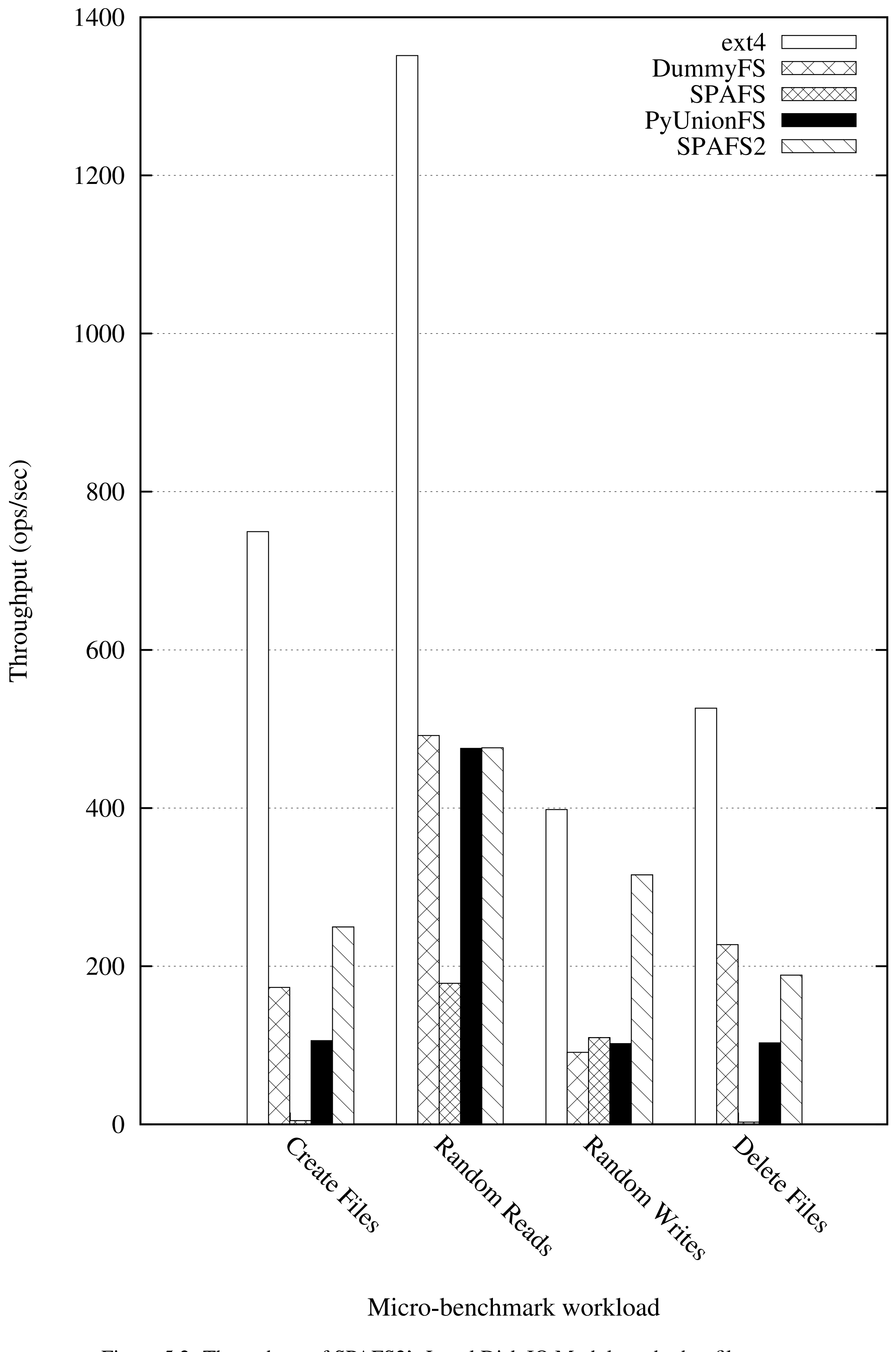

Figure 5.2: Throughput of SPAFS2's Local Disk IO Module and other file systems

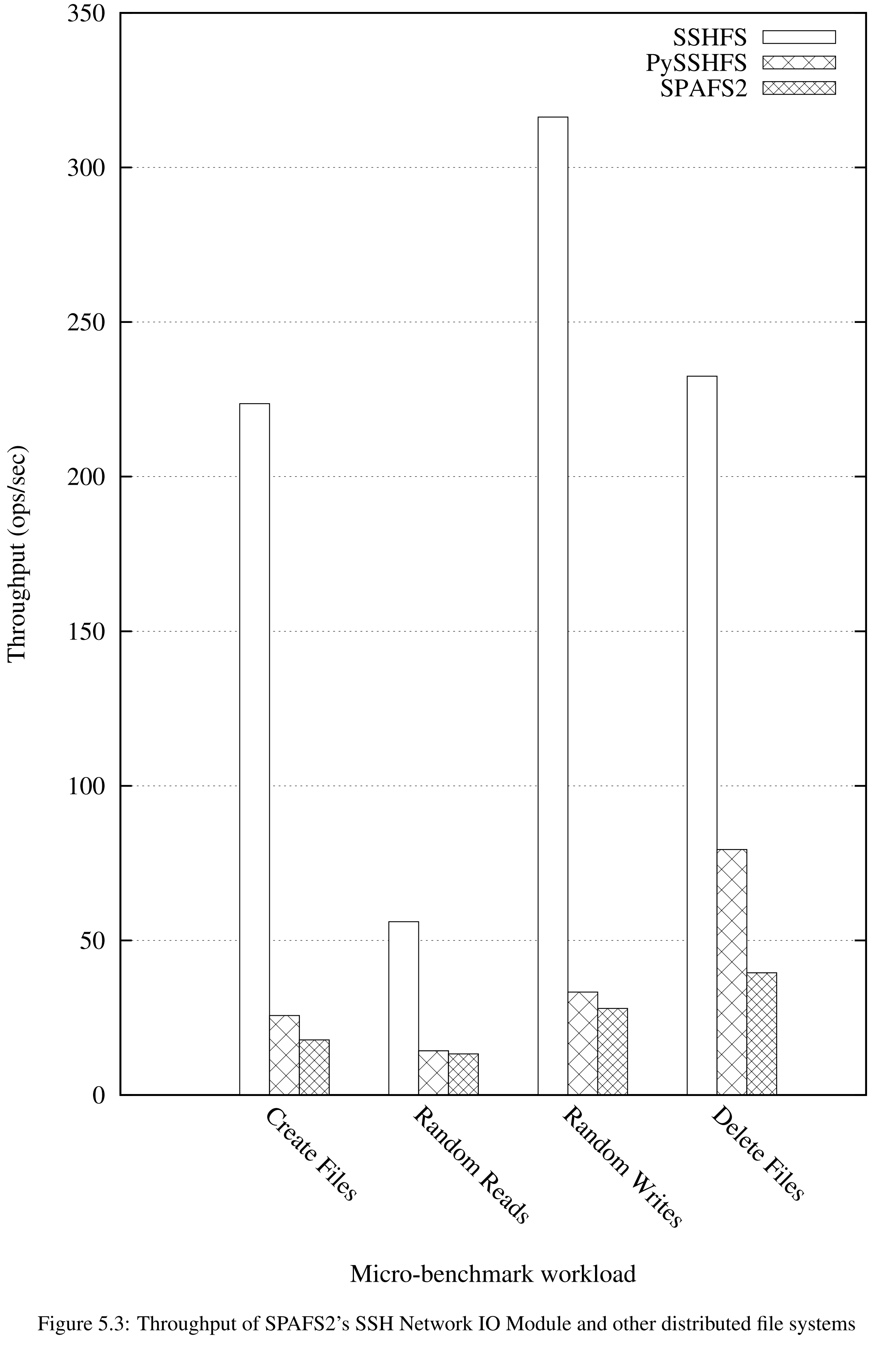

Figure 5.3: Throughput of SPAFS2's SSH Network IO Module and other distributed file systems

CHAPTER VI

CONCLUSION

In this dissertation, we present the design of Personal Cloud File System Framework, a modular userspace file system framework for accessing and manipulating files and folders on multiple personal computers and portable devices. We also implement Simple Protocol Agnostic File System 2 (SPAFS2), the prototype of our framework.

Since Personal Cloud File System Framework combines ideas and concepts from distributed file systems and unification file systems, our framework shared many similarities with both distributed file systems and unification file systems.

Our framework is similar to SSHFS-MUX [4], which is a distributed file system that also capables of merging multiple remote directories into one unified view. Unlike SSHFS-MUX, Personal Cloud File System Framework is capable of merging both remote and local directories while also supports other features such as branch's permission, Whiteout, and Copy Up. Personal Cloud File System Framework is also extensible through IO Module, unlike SSHFS-MUX which only supports SSH protocol.

Another distributed file system that is extensible and also supports multiple network protocols is YaFS [8]. Like our framework, YaFS also does not have its own servers. Unlike our framework, YaFS stripes data before storing them on the servers. While this method of storing data have many benefits, such as bandwidth efficiency, it also means that the user cannot access his data on the servers without using YaFS. The user also cannot use YaFS to access his existing data on the remote machine.

The design of Personal Cloud File System Framework's Cache Management component and how the framework handles offline operation have been inspired by Coda [15].

Just like Unionfs [25], Personal Cloud File System Framework supports all fundamental concepts of unification file systems: branch's priority and permission, Whiteout, Copy Up, and recursive unification. Personal Cloud File System Framework also supports a mixture of Read-Write and Read-Only branches on the condition that the top branch's permission must always be Read-Write. Unlike Unionfs, Personal Cloud File System Framework only creates a shadow directory when performing a copy up operation. Personal Cloud File System Framework also does not support dynamic insertion or removal of branches. Also, just like Plan 9's Union Directory [12] and 4.4BSD-Lite's Union Mount [11], our framework does not preserve ownership of each file that have been copied up.

Thanks to the fact that Personal Cloud File System Framework is implemented as a userspace file system, it can run on all Linux distributions as well as other UNIX-like operating systems that support Python binding of FUSE [20] as well as other libraries required by the framework.

We have measured the performance of SPAFS2 against other file systems that have similar functionality using micro-benchmark workloads of Filebench [21]. The results show that the performance of SPAFS2 is on par with and in some workloads even exceeds other file systems that are being compared against it.

6.1 Contributions

With SPAFS2, the prototype of Personal Cloud File System Framework, we have not only created a working prototype of our framework, but also shown that a modular userspace file system that combines the idea of unification file systems and distributed file systems is feasible and that the performance of such file system is acceptable with minimum overhead. The prototype that we have created can also be used as a building-block to create more sophisticated file systems in the future.

We also abstracts and summaries the code and algorithms needed to create a functional unification file systems and shows that these algorithms are general enough to work on both local and remote files and folders.

We present the design and implementation of Branch Tag, an extension of fundamental concepts of unification file systems that enables the user to directly specify which file in which branch to operate on as well as show files and folders that have been hidden by the framework. By implementation Branch Tag, We also show that our prototype is capable of supporting alternative conflict resolution methods.

Also, as by-products of us trying to implement the working prototype of Personal Cloud File System Framework, several fully functional proof-of-concept userspace file systems have been created.

We have also gained valuable knowledge on how to implement a userspace file system using FUSE's Python binding. We have also described, in great detail, our experiences in trying to implement the working prototype of our framework.

6.2 Future Works

While the core functionalities of Personal Cloud File System Framework is in place, specifically the ability to unify multiple remote and local directories into one unified view and the modular architecture that supports extending the framework via IO module, there is still a lot of work left to do.

We still have yet to implement and test Cache Management, one of the major component in the design of Personal Cloud File System Framework.

Our current design and implementation is specific to Linux and other UNIX-like operating system and is not really cross-platform. The current implementation and algorithms used is heavily ties to how FUSE, especially how Python binding of FUSE, works. Future work that explores how to adapt our design and implementation to other platforms that are not Linux or UNIX-like might be worthwhile.

Also, an in-dept performance analysis of SPAFS2 is also a worthwhile pursuit.

References

[1] Jerome Vouillon Benjamin C. Pierce. 2004. Whats in unison? a formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, University of Pennsylvania.

[2] Canonical Ltd. 2009. Release Notes. [Online]. Available from: https://wiki.ubuntu.com/JauntyJackalope/ReleaseNotes . [2011, April 5].

[3] Irmen de Jong. 2010. Pyro - Python Remote Objects. [Online]. Available from: http://irmen.home.xs4all.nl/pyro3/ . [2011, March 10].

[4] Nan Dun, K. Taura, and A. Yonezawa. 2008. Gmount: Build your grid file system on the fly. In Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing, GRID '08, pp. 328–333, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-2578-5. doi: http://dx.doi.org/10.1109/GRID.2008.4662817. [Online]. Available from: http://dx.doi.org/10.1109/GRID.2008.4662817 .

[5] D. Richard Hipp. 2011. SQLite. [Online]. Available from: http://www.sqlite.org/ . [2011,April 17].

[6] Drew Houston and Arash Ferdowsi. 2011. Dropbox. [Online]. Available from: https://www.dropbox.com/ . [2011, April 4].

[7] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. ACM Trans. Comput. Syst. 6:51–81. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/35037.35059. [Online]. Available from: http://doi.acm.org/10.1145/35037.35059 .

[8] Yutong Lu, Huajian Mao, and Jie Shen. 2009. A distributed filesystem framework for transparent accessing heterogeneous storage services. In IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&amp;Distributed Processing, pp. 1–8, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: http://dx.doi.org/10.1109/IPDPS.2009.5161180.

[9] James G. Mitchell and Jeremy Dion. 1982. A comparison of two network-based file servers. Commun. ACM 25:233–245. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358468.358475. [Online]. Available from: http://doi.acm.org/10.1145/358468.358475 .

[10] Paramiko Team. 2011. Paramiko. [Online]. Available from: http://www.lag.net/paramiko/ . [2011, January 2].

[11] Jan-Simon Pendry and Marshall Kirk McKusick. 1995. Union mounts in 4.4bsd-lite. In Proceedings of the USENIX 1995 Technical Conference Proceedings, TCON'95, pp. 3–3, Berkeley, CA, USA. USENIX Association. [Online]. Available from: http://portal.acm.org/citation.cfm?id=1267411.1267414 .

[12] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. 1990. Plan 9 from bell labs. In In Proceedings of the Summer 1990 UKUUG Conference, pp. 1–9.

[13] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. 1981. Locus a network transparent, high reliability distributed system. SIGOPS Oper. Syst. Rev. 15:169–177. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1067627.806605. [Online]. Available from: http://doi.acm.org/10.1145/1067627.806605 .

[14] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. 1988. Innovations in internetworking. chapter Design and implementation of the Sun network filesystem, pp. 379–390. Artech House, Inc., Norwood, MA, USA. ISBN 0-89006-337-0. [Online]. Available from: http://dl.acm.org/citation.cfm?id=59309.59338 .

[15] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. 1990. Coda: A highly available file system for a distributed workstation environment. IEEE Trans. Comput. 39:447–459. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.54838. [Online]. Available from: http://dx.doi.org/10.1109/12.54838 .

[16] Liba Svobodova. 1981. A reliable object-oriented data repository for a distributed computer system. SIGOPS Oper. Syst. Rev. 15:47–58. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1067627.806591. [Online]. Available from: http://doi.acm.org/10.1145/1067627.806591 .

[17] Liba Svobodova. 1984. File servers for network-based distributed systems. ACM Comput. Surv. 16:353–398. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/3872.3873. [Online]. Available from: http://doi.acm.org/10.1145/3872.3873 .

[18] Daniel Swinehart, Gene McDaniel, and David Boggs. 1979. Wfs a simple shared file system for a distributed environment. In Proceedings of the seventh ACM symposium on Operating systems principles, SOSP '79, pp. 9–17, New York, NY, USA. ACM. ISBN 0-89791-009-5. doi: http://doi.acm.org/10.1145/800215.806564. [Online]. Available from: http://doi.acm.org/10.1145/800215.806564 .

[19] Miklos Szeredi. 2011. SSH Filesystem. [Online]. Available from: http://fuse.sourceforge.net/sshfs.html . [2011, March 9].

[20] Miklos Szeredi. 2011. FUSE: Filesystem in Userspace. [Online]. Available from: http://fuse.sourceforge.net/ . [2011, January 2].

[21] Vasily Tarasov. 2011. FileBench. [Online]. Available from: http://filebench.sourceforge.net/ . [2011, September 13].

[22] The Beat Writers. 2009. Release Notes. [Online]. Available from: http://docs.fedoraproject.org/en-US/Fedora/11/html/Release_Notes . [2011, April 5].

[23] Guido van Rossum. 2011. Python programming language. [Online]. Available from: http://www.python.org/ . [2011, June 3].

[24] Kaushik Veeraraghavan, Jason Flinn, Edmund B. Nightingale, and Brian Noble. 2010. qufiles: the right file at the right time. In Proceedings of the 8th USENIX conference on File and storage technologies, FAST'10, pp. 1–1, Berkeley, CA, USA. USENIX Association. [Online]. Available from: http://dl.acm.org/citation.cfm?id=1855511.1855512 .

[25] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and unix semantics in namespace unification. Trans. Storage 2:74–105. ISSN 1553-3077. doi: http://doi.acm.org/10.1145/1138041.1138045. [Online]. Available from: http://doi.acm.org/10.1145/1138041.1138045 .

APPENDICES

APPENDIX A

EVOLUTION OF PERSONAL CLOUD FILE SYSTEM FRAMEWORK'S PROTOTYPES

In this chapter, we describe the evolution of Personal Cloud File System Framework's prototypes.

We begin by describing DummyFS, a simple userspace file system that we implemented in order to learn FUSE's Python binding, in Section A.1. Next, we describe SPAFS, the framework's first prototype, in Section A.2, then proceed to talk about PyUnionFS, the proof-of-concept file system that implements early version of the algorithms that were later used in SPAFS2, in Section A.3. PySSHFS, the proof-of-concept for SPAFS2's SSH Network IO Module, is discussed next in Section A.4 and we conclude with a brief summary in Section A.5.

A.1 DummyFS

Dummy File System (DummyFS) is a simple userspace file system implemented using Python [23] and Python binding of FUSE [20] that provides access to existing files or folders from its mount point.

It is the first userspace file system that we have created in order to learn how to use FUSE's Python binding. Despite this, DummyFS supports all of the basic file system operations and is actually usable as a makeshift loopback file system.

All of userspace file systems that we later created are derived from DummyFS.

A.2 SPAFS

Simple Protocol Agnostic File System (SPAFS) is the first prototype of Personal Cloud File System Framework that successfully merges contents of two or more directories by using a database to store branch's data and using these data inside the database to conduct files lookup.

SPAFS only implement 2 out of 3 major components of Personal Cloud File System Framework: Interface and List Management. Consequently, SPAFS does not support caching of file for offline usage.

During the development of SPAFS, we try to adherent as closely as possible to the initial design of Personal Cloud File System Framework, which, as shown in Figure A.1, have List Management as a separate component from Interface component.

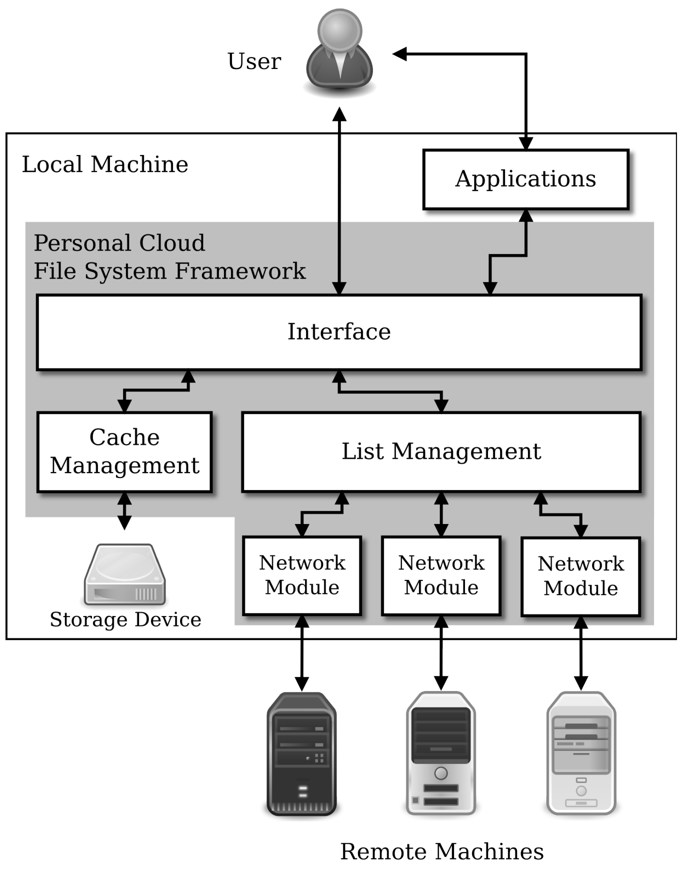

Figure A.1: The initial design of Personal Cloud File System Framework.

This results in the first version of SPAFS implements List Management as a separate daemon process, communicates with Interface component through RPC using Python's Pyro library [3] and stores branch's directory trees inside a SQLite [5] database.

We implement Interface component of SPAFS as userspace file system using FUSE's Python binding.

While this implementation technique works well enough as long as the objects that are passed between Interface and List Management component are pickable, we quickly grew frustrated with this constraint as it limits what we can do with List Management. For example, We cannot pass file objects between Interface and List Management component, meaning that we cannot have List Management handles any file system operations whether directly or through other modules. We are also concerned about performance overhead that Pyro might causes.

For these reasons, we abandon the idea of implementing List Management as a separate process and, in the second version of SPAFS, implement it as a Python module instead.

Aside from having List Management as a Python module that is directly imported and used by Interface component, the second version of SPAFS is pretty much identical to the first one. We still use Python FUSE to implement Interface component and List Management still stores branch's data in the SQLite database.

The decision to implement List Management as a Python module turns out to be a correct one as it simplify how Interface component communicates with List Management component, as well as giving us greater flexibility and freedom on how to implement List Management's features.

During file system's initialization period, SPAFS walks each branch's directory tree recursively and adds all file and folder from each branch as well as other "metadata" information about each file or folder, such as path to the said file or folder, hash of file's name, etc., to the database.

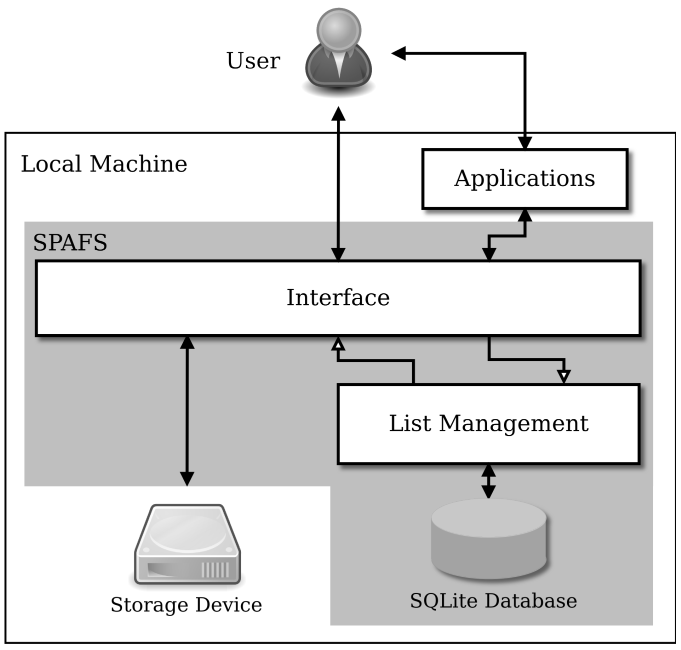

Figure A.2: How SPAFS handles file system calls from the user and applications.

SPAFS operates by passing file or folder path that Interface component received from the user or applications onto List Management component. List Management then used the supplied path to query the database, looking for the matching file and folder name and, once the match is found, return the file or folder "real" path to Interface component to operate on. Figure A.2 shows how SPAFS handles file system calls from the user.

SPAFS is where we first implement the algorithm that hides duplicate directory entry when the user lists SPAFS's mount point contents. A modified version of this algorithm were later used by both PyUnionFS and SPAFS2.

Algorithm 7 Algorithm used by SPAFS to hide duplicate directory entry

```
$dir\_entries\_list$.append(".", "..")
if $path$ = "/" then
    for $branch\_path$ in $branch\_paths\_list$ do
        $branch\_dir\_entries\_list \leftarrow$ os.listdir($branch\_path$)
        for $entry$ in $branch\_dir\_entries\_list$ do
            if $entry$ not in $dir\_entries\_list$ then
                $dir\_entries\_list$.append($entry$)
                return $dir\_entries\_list$
            end if
        end for
    end for
else
    $queried\_path \leftarrow$ list_management.request_path($path$)
    $branch\_dir\_entries\_list \leftarrow$ os.listdir($queried\_path$)
    for $entry$ in $branch\_dir\_entries\_list$ do
        $dir\_entries\_list$.append($entry$)
        return $dir\_entries\_list$
    end for
end if
```

As shown in Algorithm 7, the algorithm works by first check if the path supplied is the root path or not. If the path supplied is the root path, then the algorithm loop over a list of branch's paths, listing and adding contents of every branch's root path to $dir\_entries\_list$, which in our implementation is a Python's list. The exceptions are, of course, files and folders that are already in $dir\_entries\_list$.

If the path supplied is not the root path, then the algorithm pass the path onto List Management. List Management then query the database to find the matching folder name and, once the match is found, return the "real" path to folder back to the algorithm, which then proceed to add contents of the folder to $dir\_entries\_list$. Note that in this case we do not need to check for duplicate files or folders in subfolder before adding its contents to $dir\_entries\_list$ because every subfolder's namespace is already consistence.

After the algorithm have finished populating $dir\_entries\_list$, the list is then used by FUSE to satisfy user's request to list directory.

Another important breakthrough we had with SPAFS is the finding that Python FUSE use getattr method to check for the existence of a file or folder before creating it and that Python FUSE will only create new file or folder if the getattr method return error value -errno.ENOENT (-2).

This means that for Interface component to work correctly, we must make sure that getattr method return error value -2 for every files and folders that does not exist.

We had planned to implement Cache Management component for SPAFS, as well as creating plugin interface for List Management. However, after preliminary benchmark results shown that SPAFS have unacceptable performance, we decided to abandon the idea of using a database to store branch information.

A.3 PyUnionFS

After the abysmal benchmark results of SPAFS show us that relying on a database for most of file system operations is probably not a good idea, at least performance wise, we decided to start over from the beginning and try to implement file system that can unify contents of multiple directories without having to rely on a database or any other form of persistence storage.

Python Unification File System (PyUnionFS), a proof-of-concept unification userspace file system implemented entirely in Python using Python FUSE, is the result of this rewrite.

The first iteration of PyUnionFS tries to emulate how SPAFS works, but instead of storing branch's data inside a database, PyUnionFS stores branch's data in the memory (in a Python's list, to be precise). While preliminary evaluation of PyUnionFS shows that it performs better than SPAFS, we still need to walk each branch's directory tree and store the data inside PyUnionFS's memory during file system's initialization process, which can take considerable amount of time if the branch's directory tree contains large amount of files and folders.

Later, having been inspired by the simplicity of SSHFS-MUX [4] way of handling files and folders lookup, which basically boils down to looping through each branch while trying to perform operations on the file or folder and stop once the operations complete successfully, we implement a new lookup algorithm that uses similar concept.

The new lookup algorithm works by simply looping over a list of branch's paths, using the user supplied path to check for the existence of file for folder that the user wish to operate on. Once the file or folder is found, we operate on it.

This new lookup algorithm enables us to implement code necessary for merging the contents of multiple directories without having to first store data of each branch.

Another nice benefit of this new lookup method is that since we use the structure of directory tree itself for files and folders lookup, any changes in any of the branches will also shows up in our file system, while in SPAFS if any changes are made directly to the branch without going through SPAFS's mount point it will causes an error during lookup operations because now the branch's directory tree is different than the one SPAFS had stored in its database.

With PyUnionFS, we are able to not only implement directory unification, but other features of unification file system, such as branch's permission, whiteout, and copy up, have also been successfully implemented. Most of the algorithms used to implemented these features in PyUnionFS will later be used in Simple Protocol Agnostic File System 2 (SPAFS2), the current prototype of Personal Cloud File System Framework.

Performance evaluation of PyUnionFS shows that the new file system performs significantly better than SPAFS, thanks in large part to the new lookup algorithm that does not require storing branch's data in a database or any other storages at all.

A.4 PySSHFS

Python Secure SHell File System (PySSHFS) is an userspace file system that let the user operating on files on a remote machine using SFTP subsystem of SSH. It is implemented as a proof-of-concept for SPAFS2's SSH Network IO Module.

Like its name implies, PySSHFS is implemented in Python using Python FUSE just like SPAFS and PyUnionFS. SSH and SFTP support for PySSHFS were implemented using Paramiko library [10].

While PySSHFS supports most basic file operations, it does not have any caching mechanism. Couple this with the fact that Paramiko library is written entirely in Python means that while PySSHFS is functional, its performance is less than desirable.

A.5 Summary

In this chapter, we describe early prototypes of Personal Cloud File System Framework. The lessons learned from each of these prototypes are used in the design and implement of both Personal Cloud File System Framework itself and SPAFS2, the current prototype of the framework at the time of this writing.

APPENDIX B

DUMMYFS'S SOURCE CODE

This chapter presents the full source code of Dummy File System (DummyFS).

B.1 DummyFS's Main Program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#    DummyFS - FUSE based Dummy Filesystem
#
#    Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import stat
import errno
import logging

import fuse

fuse.fuse_python_api = (0, 2)

class DummyStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 2
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 4096
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0

class DummyFS(fuse.Fuse):
    def __init__(self, *args, **kw):
        """
        Initialize file system class
        """
        fuse.Fuse.__init__(self, *args, **kw)

        self.root = os.path.expanduser('~')

        self.logging = False

        self.log_dir = os.path.expanduser('/tmp/Dummyfs')
        self.log_file = os.path.join(self.log_dir, 'Dummylog')

    def fsinit(self):
        """
        Initialize file system.
        """
        os.chdir(self.root)

        if self.logging:
            if os.path.exists(self.log_dir):
                if os.path.exists(self.log_file):
                    os.rename(self.log_file, os.path.join(self.
      log_dir,
                              'Dummylog.old'))
            else:
                os.makedirs(self.log_dir)

            logging.basicConfig(filename=self.log_file,
                                level=logging.DEBUG,)

            logger = logging.getLogger('DummyFS.fsinit')
            logger.debug('method_called')
            logger.debug('current_root_path_is_{0}'.format(self.root)
  )

def getattr(self, path):
    """
    Get file/folder attributes.
    """
    if self.logging:
        logger = logging.getLogger('DummyFS.getattr')
        logger.debug('path_is_{0}'.format(path))

    stat = DummyStat()

    os_st = os.lstat("." + path)

    stat.st_mode = os_st.st_mode
    stat.st_ino = os_st.st_ino
    stat.st_dev = os_st.st_dev
    stat.st_nlink = os_st.st_nlink
    stat.st_uid = os_st.st_uid
    stat.st_gid = os_st.st_gid
    stat.st_size = os_st.st_size
    stat.st_atime = os_st.st_atime
    stat.st_mtime = os_st.st_mtime
    stat.st_ctime = os_st.st_ctime

    return stat

def readdir(self, path, offset):
    """
    Read directory.
    """
    if self.logging:
        logger = logging.getLogger('DummyFS.readdir')
        logger.debug('path_is_{0},_offset_is_{1}'.format(path,
  offset))

    ld = os.listdir("." + path)

    for r in (".", ".."):
        ld.append(r)

    for i in ld:
        yield fuse.Direntry(i)

def mkdir(self, path, mode):
    """
    Create a directory.
    """
    if self.logging:
        logger = logging.getLogger('DummyFS.mkdir')
        logger.debug('path_is_{0},_mode_is_{1}'.format(path, mode
  ))

    os.mkdir("." + path, mode)

def rmdir(self, path):
    """
    Remove a directory.
    """
    if self.logging:
        logger = logging.getLogger('DummyFS.rmdir')
        logger.debug('path_is_{0}'.format(path))
        os.rmdir("." + path)

    def read(self, path, size, offset):
        """
        Read data from an open file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.read')
            logger.debug('path␣is␣{0},␣size␣is␣{1},␣offset␣is␣{2}'.
      format(path,
                          size, offset))

        with open("." + path, 'r') as file:
            file.seek(offset)
            return file.read(size)

    def mknod(self, path, mode, dev):
        """
        Create a file node.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.mknod')
            logger.debug('path␣is␣{0},␣mode␣is␣{1},␣dev␣is␣{2}'.
      format(path,
                              mode, dev))

        os.mknod("." + path, mode, dev)

    def write(self, path, buf, offset):
        """
        Write data to an open file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.write')
            logger.debug('path␣is␣{0},␣buf␣is␣{1!r},␣offset␣is␣{2}'.
      format(
                              path, buf, offset))

        with open("." + path, 'r+') as file:
            file.seek(offset)
            file.write(buf)
            return len(buf)

    def rename(self, path, path1):
        """
        Rename a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.rename')
            logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path,
      path1))

        os.rename("." + path, "." + path1)

    def truncate(self, path, size):
        """
        Change the size of a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.truncate')
            logger.debug('path␣is␣{0},␣size␣is␣{1}'.format(path, size
      ))

        with open("." + path, "a") as file:
            file.truncate(size)

    def unlink(self, path):
        """
        Remove a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.unlink')
            logger.debug('path␣is␣{0}'.format(path))

        os.unlink("." + path)

    def readlink(self, path):
        """
        Read the target of a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.readlink')
            logger.debug('path␣is␣{0}'.format(path))

        return os.readlink("." + path)

    def symlink(self, path, path1):
        """
        Create a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.symlink')
            logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path,
      path1))

        os.symlink(path, "." + path1)

    def link(self, path, path1):
        """
        Create a hard link to a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.link')
            logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path,
      path1))

        os.link("." + path, "." + path1)

    def chmod(self, path, mode):
        """
        Change the permission bits of a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.chmod')
            logger.debug('path␣is␣{0},␣mode␣is␣{1}'.format(path, mode
      ))

        os.chmod("." + path, mode)

    def chown(self, path, user, group):
        """
        Change the owner and group of a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.chown')
            logger.debug('path␣is␣{0},␣user␣is␣{1},␣group␣is␣{2}'.
      format(path,
                              user, group))

        os.chown("." + path, user, group)

    def utime(self, path, times):
        """
        Change the access and/or modification times of a file.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.utime')
            logger.debug('path␣is␣{0},␣times␣is␣{1}'.format(path,
      times))

        os.utime("." + path, times)

    def statfs(self):
        """
        Get file system statistics.
        """
        if self.logging:
            logger = logging.getLogger('DummyFS.statfs')
            logger.debug('method␣called')

        os.statvfs(".")

    def fsdestroy(self):
        if self.logging:
            logger = logging.getLogger('DummyFS.fsdestroy')
            logger.debug('method called')

    def main(self, *args, **kw):
        """
        Filesystem main method
        """
        self.root = os.path.realpath(self.root)

        return fuse.Fuse.main(self, *args, **kw)

def main():
    usage=fuse.Fuse.fusage +\
"\nDummyFS: A toy filesystem that I create in order to learn Python-
      FUSE."

    server = DummyFS(version="%prog " + fuse.__version__,
                     usage=usage,
                     dash_s_do='setsingle')
    server.parser.add_option(mountopt='root',
                             metavar="PATH",
                             default=server.root,
                             help=\
"mirror filesystem from under PATH (default: %default)")
    server.parser.add_option('-l', '--logging',
                             action="store_true",
                             dest='logging',
                             default=server.logging,
                             help="enable logging to a file")
    server.parse(values=server, errex=1)

    if server.logging:
        print 'Start logging to file: {0}\n'.format(server.log_file)

    server.main()

if __name__ == '__main__':
    main()
```

APPENDIX C

SPAFS'S SOURCE CODE

This chapter presents the full source code of Simple Protocol Agnostic File System (SPAFS) and its List Management.

C.1 SPAFS's Main Program

```
#!/usr/bin/env python

#   SPAFS - Single Point Access Filesystem
#
#   Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import stat
import errno
import logging
import ConfigParser

import fuse
import pynotify
import list_management

fuse.fuse_python_api = (0, 2)

class SPAStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 2
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 4096
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0

class SPAFS(fuse.Fuse):
    def __init__(self, *args, **kw):
        """
        Initialize file system class
        """
        fuse.Fuse.__init__(self, *args, **kw)

        self.root = "."

        self.logging = False
        self.log_file = '/tmp/SPALog'
        self.config_file = '/tmp/spa.conf'

        self.branch = None

    def local_walk(self, path):
        """
        Reimplementation of Python's os.walk using stat module
        """
        dirs, nondirs = [], []
        names = os.listdir(path)

        for name in names:
            joined_path = os.path.join(path, name)

            mode = os.lstat(joined_path).st_mode
            if stat.S_ISDIR(mode):
                dirs.append(name)
            else:
                nondirs.append(name)

        yield path, dirs, nondirs

        for name in dirs:
            new_path = os.path.join(path, name)
            for i in self.local_walk(new_path):
                yield i

    def fsinit(self):
        """
        Initialize file system.
        """
        if self.logging:
            if os.path.exists(self.log_file):
                os.rename(self.log_file, '/tmp/SPALog.old')
            logging.basicConfig(filename=self.log_file, level=logging
.DEBUG,)
        logger = logging.getLogger('SPAFS.fsinit')
        logger.debug('method_called')
        logger.debug('current_root_path_is_{0}'.format(self.root))
        for i in range(len(self.splited_branches)):
            logger.debug('branch_{0}_is_{1}'.format(i + 1,
                                                    self.
splited_branches[i]))

        os.chdir(self.root)

        pynotify.init("SPA_Filesystem")

        list_management.create_db()

        n = pynotify.Notification("SPA_Filesystem",
            "Populate_database_with_brach's_directory_structure",
            "gtk-harddisk")
        n.set_timeout(0)
        n.show()

        num_list = iter(range(100))

        for branch_path in self.splited_branches:
            logger.debug('walk_into_{0}'.format(branch_path))
            branch_name = \
            str(num_list.next()) + '_' + os.path.basename(branch_path
)
            for path, dirs, files in self.local_walk(branch_path):
                list_management.pop_db(path, dirs, files, branch_name
)

        n.update("SPA_Filesystem", "Database_populated", "gtk-
harddisk")
        n.set_timeout(2000)
        n.show()

    def local_getattr(self, path):
        """
        Query database for path to file/folder and return its
attributes
        """
        if path == "/":
            return os.lstat(self.splited_branches[0])
    if list_management.request_path_lstat(self.splited_branches,
   path) == \
        errno.ENOENT:
        return errno.ENOENT
    else:
        qpath = \
        list_management.request_path_lstat(self.splited_branches,
    path)
        return os.lstat(qpath)

def getattr(self, path):
    """
    Get file/folder attributes.
    """
    logger = logging.getLogger('SPAFS.getattr')
    logger.debug('path_is_{0}'.format(path))

    if self.local_getattr(path) == errno.ENOENT:
        return -errno.ENOENT
    else:
        entry_st = self.local_getattr(path)

    stat = SPAStat()

    stat.st_mode = entry_st.st_mode
    stat.st_ino = entry_st.st_ino
    stat.st_dev = entry_st.st_dev
    stat.st_nlink = entry_st.st_nlink
    stat.st_uid = entry_st.st_uid
    stat.st_gid = entry_st.st_gid
    stat.st_size = entry_st.st_size
    stat.st_atime = entry_st.st_atime
    stat.st_mtime = entry_st.st_mtime
    stat.st_ctime = entry_st.st_ctime

    return stat

def readdir(self, path, offset):
    """
    Read directory.
    """
    logger = logging.getLogger('SPAFS.readdir')
    logger.debug('path_is_{0},_offset_is_{1}'.format(path, offset
   ))

    ld = []

    if path == '/':
        for branch_path in self.splited_branches:
            ldd = os.listdir(branch_path)
            for i in ldd:
                if i not in ld:
                    ld.append(i)
    else:
        qpath = \
        list_management.request_path_listdir(self.
   splited_branches, path)
        ldd = os.listdir(qpath)
        for i in ldd:
            ld.append(str(i))

    for r in (".", ".."):
        ld.append(r)

    for i in ld:
        yield fuse.Direntry(i)

def mkdir(self, path, mode):
    """
    Create a directory.
    """
    logger = logging.getLogger('SPAFS.mkdir')
    logger.debug('path_is_{0},_mode_is_{1}'.format(path, mode))

    qpath = \
    list_management.request_path_create(self.splited_branches,
                                        path, 'True')
    os.mkdir(qpath, mode)
```

```
def rmdir(self, path):
    """
    Remove a directory.
    """
    logger = logging.getLogger('SPAFS.rmdir')
    logger.debug('path_is_{0}'.format(path))

    qpath = list_management.request_path_rmdir(self.
  splited_branches, path)
    os.rmdir(qpath)

def read(self, path, size, offset):
    """
    Read data from an open file.
    """
    logger = logging.getLogger('SPAFS.read')
    logger.debug('path_is_{0},_size_is_{1},_offset_is_{2}'.format
  (path,
              size, offset))

    qpath = list_management.request_path_read(self.
  splited_branches, path)
    with open(qpath, "r") as file:
        file.seek(offset)
        return file.read(size)

def mknod(self, path, mode, dev):
    """
    Create a file node.
    """
    logger = logging.getLogger('SPAFS.mknod')
    logger.debug('path_is_{0},_mode_is_{1},_dev_is_{2}'.format(
  path,
              mode, dev))

    qpath = \
    list_management.request_path_create(self.splited_branches,
  path)
    os.mknod(qpath, mode)

def write(self, path, buf, offset):
    """
    Write data to an open file.
    """
    logger = logging.getLogger('SPAFS.write')
    logger.debug('path_is_{0},_offset_is_{1}'.format(path, offset
  ))

    qpath = list_management.request_path_read(self.
  splited_branches, path)
    with open(qpath, "r+") as file:
        file.seek(offset)
        file.write(buf)
        return len(buf)

def rename(self, path, dest_path):
    """
    Rename a file.
    """
    logger = logging.getLogger('SPAFS.rename')
    logger.debug('path_is_{0},_dest_path_is_{1}'.format(path,
  dest_path))

    return -errno.EPERM

def truncate(self, path, size):
    """
    Change the size of a file.
    """
    logger = logging.getLogger('SPAFS.truncate')
    logger.debug('path_is_{0},_size_is_{1}'.format(path, size))

    qpath = list_management.request_path_read(self.
  splited_branches, path)
    with open(qpath, "a") as file:
        file.truncate(size)
    def unlink(self, path):
        """
        Remove a file.
        """
        logger = logging.getLogger('SPAFS.unlink')
        logger.debug('path␣is␣{0}'.format(path))

        qpath = \
        list_management.request_path_unlink(self.splited_branches,
      path)
        os.unlink(qpath)

    def readlink(self, path):
        """
        Read the target of a symbolic link.
        """
        logger = logging.getLogger('SPAFS.readlink')
        logger.debug('path␣is␣{0}'.format(path))

        return -errno.EPERM

    def symlink(self, path, path1):
        """
        Create a symbolic link.
        """
        logger = logging.getLogger('SPAFS.symlink')
        logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path, path1))

        return -errno.EPERM

    def link(self, path, path1):
        """
        Create a hard link to a file.
        """
        logger = logging.getLogger('SPAFS.link')
        logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path, path1))

        return -errno.EPERM

    def chmod(self, path, mode):
        """
        Change the permission bits of a file.
        """
        logger = logging.getLogger('SPAFS.chmod')
        logger.debug('path␣is␣{0},␣mode␣is␣{1}'.format(path, mode))

        return -errno.EPERM

    def chown(self, path, user, group):
        """
        Change the owner and group of a file.
        """
        logger = logging.getLogger('SPAFS.chown')
        logger.debug('path␣is␣{0},␣user␣is␣{1},␣group␣is␣{2}'.format(
      path,
                    user, group))

        return -errno.EPERM

    def utime(self, path, times):
        """
        Change the access and/or modification times of a file.
        """
        logger = logging.getLogger('SPAFS.utime')
        logger.debug('path␣is␣{0},␣times␣is␣{1}'.format(path, times))

        qpath = list_management.request_path_read(self.
      splited_branches, path)
        os.utime(qpath, times)

    def statfs(self):
        """
        Get file system statistics.
        """
        logger = logging.getLogger('SPAFS.statfs')
        logger.debug('method␣called')

        os.statvfs(self.splited_branches[0])

    def fsdestroy(self):
        """
        Called when filesystem is unmount
        """
        logger = logging.getLogger('SPAFS.fsdestroy')
        logger.debug('method␣called')

        n = pynotify.Notification("SPA␣Filesystem", "Filesystem␣
      unmount",
                                  "gtk-harddisk")
        n.set_timeout(pynotify.EXPIRES_DEFAULT)
        n.show()

    def main(self, *a, **kw):
        """
        Filesystem main method
        """
        self.root = os.path.realpath(self.root)

        config = ConfigParser.RawConfigParser()

        branch_section = 'List␣of␣branch␣to␣unified'

        if os.path.isfile(self.config_file):
            config.read(self.config_file)

            self.branch = config.get(branch_section, 'branch')

        else:
            config.add_section(branch_section)
            config.set(branch_section, 'branch', 'None')

            with open(self.config_file, 'wb') as configfile:
                config.write(configfile)

        if self.branch != 'None':
            self.splited_branches = self.branch.split(":")
        for i in range(len(self.splited_branches)):
            self.splited_branches[i] = \
            os.path.realpath(self.splited_branches[i])

        return fuse.Fuse.main(self, *a, **kw)

def main():
    usage="""
SPAFS: Single Point Access Filesystem
    """ + fuse.Fuse.fusage

    server = SPAFS(version="%prog␣" + fuse.__version__,
                   usage=usage,
                   dash_s_do='setsingle')
    server.parser.add_option(mountopt = 'branch',
                             metavar = "PATH1:PATH2:...",
                             default = server.branch,
                             help = "List␣of␣folder's␣PATH␣to␣merge␣
      separated␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣colon")
    server.parser.add_option('-l', '--logging',
                             action="store_true",
                             dest='logging',
                             default=server.logging,
                             help="enable␣logging␣to␣a␣file")
    server.parse(values=server, errex=1)

    if server.logging:
        # Tell user where we put the log file.
        print 'Start␣logging␣to␣file:␣{0}\n'.format(server.log_file)

    server.main()

if __name__ == '__main__':
    main()
```

C.2 SPAFS's List Management Module

```
"""
SPAFS's List Management module.
"""

import os
import errno
import sqlite3
import hashlib

db_file = '/tmp/spalist.db'

def create_db():
    """
    Create database.
    """
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS fs_namespace
              (id INTEGER PRIMARY KEY, path TEXT, name TEXT,
              is_dir TEXT, path_hash TEXT UNIQUE, branch_name TEXT,
              logical_name TEXT, logical_path_hash TEXT)''')
    conn.commit()
    c.close()

def hash_path(path):
    """
    Find hash of path to file/folder.
    """
    return hashlib.md5(path).hexdigest()

def add_entry(path, name, is_dir, path_hash, branch_name,
              logical_name, logical_path_hash):
    """
    Add file/folder into database.
    """
    path = path.decode('utf_8')
    name = name.decode('utf_8')
    branch_name = branch_name.decode('utf_8')
    logical_name = logical_name.decode('utf_8')

    t = (path, name, is_dir, path_hash, branch_name,
         logical_name, logical_path_hash)

    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''INSERT OR IGNORE INTO fs_namespace
              (path, name, is_dir, path_hash, branch_name,
              logical_name, logical_path_hash)
              VALUES (?, ?, ?, ?, ?, ?, ?)''', t)
    conn.commit()
    c.close()

def remove_entry(path_hash):
    """
    Remove file/folder from database.
    """
    t = path_hash,

    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('DELETE FROM fs_namespace WHERE path_hash LIKE ?', t)
    conn.commit()
    c.close()

def retrieve_db_entries(is_dir=None):
    """
    Retrieve data from database.
    """
    if is_dir == None:
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('SELECT * FROM fs_namespace ORDER BY branch_name')
        unified_namespace = c.fetchall()
        c.close()
        return unified_namespace
    else:
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        t = is_dir,
        c.execute('''SELECT * FROM fs_namespace WHERE is_dir LIKE ?
                  ORDER BY branch_name''', t)
        f_namespace = c.fetchall()
        c.close()

        return f_namespace

def pop_db(path, dirs, files, branch_name):
    """
    Populate the database.

    Populate the database with directory tree like the one obtained
      from
    Python's os.walk method.
    """
    conn = sqlite3.connect(db_file)
    c = conn.cursor()

    for file in files:
        file_namespace = retrieve_db_entries('False')
        joined_path = os.path.join(path, file)
        path_hash = hash_path(joined_path)
        # Convert string into unicode
        path = path.decode('utf_8')
        file = file.decode('utf_8')
        branch_name = branch_name.decode('utf_8')

        for i in file_namespace:
            if file == i[2]:
                logical_file_name = file + '_[' + branch_name + ']'
                joined_logical_path = os.path.join(path,
   logical_file_name)
                logical_path_hash = hash_path(joined_logical_path)

                t = (path, file, 'False', path_hash, branch_name,
                    logical_file_name, logical_path_hash)
                c.execute('''INSERT OR IGNORE INTO fs_namespace
                          (path, name, is_dir, path_hash, branch_name
   ,
                          logical_name, logical_path_hash)
                          VALUES (?, ?, ?, ?, ?, ?, ?)''', t)

        t = path, file, 'False', path_hash, branch_name, file,
   path_hash
        c.execute('''INSERT OR IGNORE INTO fs_namespace
                  (path, name, is_dir, path_hash, branch_name,
                  logical_name, logical_path_hash)
                  VALUES (?, ?, ?, ?, ?, ?, ?)''', t)

    for dir in dirs:
        folder_namespace = retrieve_db_entries('True')
        joined_path = os.path.join(path, dir)
        path_hash = hash_path(joined_path)
        # Convert string into unicode
        path = path.decode('utf_8')
        dir = dir.decode('utf_8')
        branch_name = branch_name.decode('utf_8')

        for i in folder_namespace:
            if dir == i[2]:
                logical_dir_name = dir + '_[' + \
                                   branch_name + ']'
                joined_logical_path = \
                os.path.join(path, logical_dir_name)
                logical_path_hash = hash_path(joined_logical_path)

                t = (path, dir, 'True', path_hash, branch_name,
                    logical_dir_name, logical_path_hash)
                c.execute('''INSERT OR IGNORE INTO fs_namespace
                        (path, name, is_dir, path_hash, branch_name
    ,
                        logical_name, logical_path_hash)
                        VALUES (?, ?, ?, ?, ?, ?, ?)''', t)

        t = path, dir, 'True', path_hash, branch_name, dir, path_hash
        c.execute('''INSERT OR IGNORE INTO fs_namespace
                    (path, name, is_dir, path_hash, branch_name,
                    logical_name, logical_path_hash)
                    VALUES (?, ?, ?, ?, ?, ?, ?)''', t)

    conn.commit()
    c.close()

def request_path_listdir(branch_list, path):
    """
    Return a path to directroy.
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)

    folder_namespace = retrieve_db_entries('True')

    if dir_name == '/':
        for i in folder_namespace:
            for branch_path in branch_list:
                if i[1] == branch_path:
                    if i[2] == name:
                        joined_path = os.path.join(i[1], i[2])
                        return joined_path

    for i in folder_namespace:
        if dir_name in i[1]:
            if i[2] == name:
                joined_path = os.path.join(i[1], i[2])
                return joined_path

def request_path_lstat(branch_list, path):
    """
    Return path to file/folder (lstat).
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)

    if dir_name == '/':
        t = name,
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('''SELECT * FROM fs_namespace WHERE name = ?
                    ORDER BY branch_name''', t)
        f_info = c.fetchone()
        c.close()

        if f_info:
            joined_path = os.path.join(f_info[1], f_info[2])
            return joined_path

    t = (name, '%' + dir_name)
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''SELECT * FROM fs_namespace WHERE name = ? and path
      like ?
            ORDER BY branch_name''', t)
    f_info = c.fetchone()
    c.close()

    if f_info:
        joined_path = os.path.join(f_info[1], f_info[2])
        return joined_path

    return errno.ENOENT

def request_path_rmdir(branch_list, path):
    """
    Return path to folder (rmdir).
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)

    if dir_name == '/':
        t = (name, 'True')
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('''SELECT * FROM fs_namespace WHERE name = ? and
      is_dir = ?
                    ORDER BY branch_name''', t)
        f_info = c.fetchone()

        if f_info:
            joined_path = os.path.join(f_info[1], f_info[2])
            path_hash = hash_path(joined_path)

            t = path_hash,

            c.execute('DELETE FROM fs_namespace WHERE path_hash LIKE 
      ?', t)
            conn.commit()
            c.close()
            return joined_path

    t = (name, '%' + dir_name, 'True')
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''SELECT * FROM fs_namespace WHERE name = ? and path
      like ?
            and is_dir = ? ORDER BY branch_name''', t)
    f_info = c.fetchone()

    if f_info:
        joined_path = os.path.join(f_info[1], f_info[2])
        path_hash = hash_path(joined_path)

        t = path_hash,

        c.execute('DELETE FROM fs_namespace WHERE path_hash LIKE ?',
      t)
        conn.commit()
        c.close()
        return joined_path

def request_path_unlink(branch_list, path):
    """
    Return path to file (unlink).
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)

    if dir_name == '/':
        t = name,
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('''SELECT * FROM fs_namespace WHERE name = ?
                    ORDER BY branch_name''', t)
        f_info = c.fetchone()

        if f_info:
            joined_path = os.path.join(f_info[1], f_info[2])
            # Remove file/folder data from database.
            path_hash = hash_path(joined_path)

            t = path_hash,

            c.execute('DELETE FROM fs_namespace WHERE path_hash LIKE 
      ?', t)
            conn.commit()
            c.close()
            return joined_path

    t = (name, '%' + dir_name)
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''SELECT * FROM fs_namespace WHERE name = ? and path
      like ?
            ORDER BY branch_name''', t)
    f_info = c.fetchone()
    if f_info:
        joined_path = os.path.join(f_info[1], f_info[2])
        # Remove file/folder data from database.
        path_hash = hash_path(joined_path)

        t = path_hash,

        c.execute('DELETE FROM fs_namespace WHERE path_hash LIKE ?',
      t)
        conn.commit()
        c.close()
        return joined_path

def request_path_read(branch_list, path):
    """
    Return path to file/folder (read).
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)

    if dir_name == '/':
        t = name,
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('''SELECT * FROM fs_namespace WHERE name = ?
                    ORDER BY branch_name''', t)
        f_info = c.fetchone()
        c.close()

        if f_info:
            joined_path = os.path.join(f_info[1], f_info[2])
            return joined_path

    t = (name, '%' + dir_name)
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''SELECT * FROM fs_namespace WHERE name = ? and path
      like ?
                ORDER BY branch_name''', t)
    f_info = c.fetchone()
    c.close()

    if f_info:
        joined_path = os.path.join(f_info[1], f_info[2])
        return joined_path

def request_path_create(branch_list, path, is_dir='False'):
    """
    Return path to new file/folder that are to be created (write).
    """
    name = os.path.basename(path)
    dir_name = os.path.dirname(path)
    folder_name = os.path.basename(dir_name)
    folder_path = os.path.dirname(dir_name)

    if dir_name == '/':
        joined_path = os.path.join(branch_list[0], name)
        path_hash = hash_path(joined_path)
        branch_name = '0_' + os.path.basename(branch_list[0])
        add_entry(branch_list[0], name, is_dir, path_hash,
      branch_name,
                  name, path_hash)
        return joined_path

    elif folder_path == '/':
        t = (folder_name, 'True')
        conn = sqlite3.connect(db_file)
        c = conn.cursor()
        c.execute('''SELECT * FROM fs_namespace WHERE name = ? and
      is_dir = ?
                    ORDER BY branch_name''', t)
        f_info = c.fetchone()

        if f_info:
            path_to_folder = os.path.join(f_info[1], f_info[2])
            joined_path = os.path.join(path_to_folder, name)
            path_hash = hash_path(joined_path)

            path_to_folder = path_to_folder.decode('utf_8')
            name = name.decode('utf_8')
            branch_name = f_info[5].decode('utf_8')
            logical_name = name.decode('utf_8')
            logical_path_hash = path_hash

            t = (path_to_folder, name, is_dir, path_hash, branch_name
      ,
                 logical_name, logical_path_hash)

            c.execute('''INSERT OR IGNORE INTO fs_namespace
                        (path, name, is_dir, path_hash, branch_name,
                        logical_name, logical_path_hash)
                        VALUES (?, ?, ?, ?, ?, ?, ?)''', t)
            conn.commit()
            c.close()
            return joined_path

    t = (folder_name, '%' + folder_path, 'True')
    conn = sqlite3.connect(db_file)
    c = conn.cursor()
    c.execute('''SELECT * FROM fs_namespace WHERE name = ? and path
      like ?
                and is_dir = ? ORDER BY branch_name''', t)
    f_info = c.fetchone()

    if f_info:
        path_to_folder = os.path.join(f_info[1], f_info[2])
        joined_path = os.path.join(path_to_folder, name)
        path_hash = hash_path(joined_path)

        path_to_folder = path_to_folder.decode('utf_8')
        name = name.decode('utf_8')
        branch_name = f_info[5].decode('utf_8')
        logical_name = name.decode('utf_8')
        logical_path_hash = path_hash

        t = (path_to_folder, name, is_dir, path_hash, branch_name,
             logical_name, logical_path_hash)

        c.execute('''INSERT OR IGNORE INTO fs_namespace
                    (path, name, is_dir, path_hash, branch_name,
                    logical_name, logical_path_hash)
                    VALUES (?, ?, ?, ?, ?, ?, ?)''', t)
        conn.commit()
        c.close()
        return joined_path
```

APPENDIX D

PYUNIONFS'S SOURCE CODE

This chapter presents the full source code of Python Unification File System (PyUnionFS).

D.1 PyUnionFS's Main Program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#    PyUnionFS - FUSE based Union File System
#
#    Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import stat
import errno
import collections
import shutil
import logging

import fuse

fuse.fuse_python_api = (0, 2)

class PyUnionStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 2
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 4096
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0

class PyUnionFS(fuse.Fuse):
    def __init__(self, *args, **kw):
        """
        Initialize file system class
        """
        fuse.Fuse.__init__(self, *args, **kw)

        self.branch = os.path.realpath('~')

        self.logging = False

        self.log_dir = os.path.expanduser('/tmp/PyUnionfs')
        self.log_file = os.path.join(self.log_dir, 'PyUnionlog')

    def resolve_path(self, path, branch_list, create=False,
                     return_paths=False):
        """
        Return a tuple containing absolute path to file or folder as
       well as
        permission of the branch that contain the said file or folder
       .

        Alternatively, return a list of one or more absolute path to
       folder
        (needed by readdir method to unify contents of subfolders).
        """
        entry_name = os.path.basename(path)
        entry_path = os.path.dirname(path)

        branch_path_list = self.branch_list.keys()

        if return_paths:
            list_of_paths = []

        if entry_path == '/':
            if create:
                return (os.path.join(branch_path_list[0], path[1:]),
 'rw')
            else:
                for path_to_branch, branch_perm in branch_list.
 iteritems():
                    abs_path = os.path.join(path_to_branch, path[1:])

                    if os.path.exists(abs_path):
                        if return_paths:
                            list_of_paths.append(abs_path)
                        else:
                            return (abs_path, branch_perm)
        else:
            for path_to_branch, branch_perm in branch_list.iteritems
 ():
                if create:
                    path_to_entry = os.path.join(path_to_branch,
                                                 entry_path[1:])

                    if os.path.exists(path_to_entry):
                        return (os.path.join(path_to_branch, path
 [1:]),
                                branch_perm)
                else:
                    abs_path = os.path.join(path_to_branch, path[1:])

                    if os.path.exists(abs_path):
                        if return_paths:
                            list_of_paths.append(abs_path)
                        else:
                            return (abs_path, branch_perm)

        if return_paths:
            return list_of_paths

        return (errno.ENOENT, )

    def return_cow_path(self, path, copy=False, source=None, dest=
  None):
        """
        Create and return a path to copy-up file or folder.
        """
        abs_entry_path = os.path.join(self.branch_path_list[0],
                                      os.path.dirname(path)[1:])

        if os.path.isdir(abs_entry_path):
            if copy:
                shutil.copyfile(source, dest)
            return os.path.join(self.branch_path_list[0], path[1:])
        else:
            os.makedirs(abs_entry_path)
            if copy:
                shutil.copyfile(source, dest)
            return os.path.join(self.branch_path_list[0], path[1:])

    def create_white_out(self, path):
        """
        Create whiteout of a given file or folder.
        """
        entry_name = os.path.basename(path)
        entry_path = os.path.dirname(path)
        white_out_file = '.' + entry_name + '.white_out'

        if entry_path == '/':
            for path_to_branch, branch_perm in self.branch_list.
       iteritems():
                path_to_entry = os.path.join(path_to_branch,
       entry_name)

                if os.path.exists(path_to_entry):
                    os.mknod(os.path.join(path_to_branch,
       white_out_file))
        else:
            for path_to_branch, branch_perm in self.branch_list.
       iteritems():
                path_to_entry = os.path.join(path_to_branch,
       entry_path[1:])

                if os.path.exists(path_to_entry):
                    os.mknod(os.path.join(path_to_entry,
       white_out_file))

    def fsinit(self):
        """
        Initialize file system.
        """
        os.chdir(self.branch_path_list[0])

        if self.logging:
            if os.path.exists(self.log_dir):
                if os.path.exists(self.log_file):
                    os.rename(self.log_file, os.path.join(self.
       log_dir,
                              'PyUnionlog.old'))
            else:
                os.makedirs(self.log_dir)

            logging.basicConfig(filename=self.log_file,
                                level=logging.DEBUG,)

        logger = logging.getLogger('PyUnionFS.fsinit')
        logger.debug('method␣called')
        logger.debug('current␣root␣path␣is␣{0}'.format(self.root))

    def getattr(self, path):
        """
        Get file/folder attributes.
        """
        logger = logging.getLogger('PyUnionFS.getattr')
        logger.debug('path␣is␣{0}'.format(path))

        real_path = self.resolve_path(path, self.branch_list)

        stat = PyUnionStat()

        if path == '/':
            os_st = os.lstat(self.branch_path_list[0])

            stat.st_mode = os_st.st_mode
            stat.st_ino = os_st.st_ino
            stat.st_dev = os_st.st_dev
            stat.st_nlink = os_st.st_nlink
            stat.st_uid = os_st.st_uid
            stat.st_gid = os_st.st_gid
            stat.st_size = os_st.st_size
            stat.st_atime = os_st.st_atime
            stat.st_mtime = os_st.st_mtime
            stat.st_ctime = os_st.st_ctime

            return stat
        elif real_path[0] == errno.ENOENT:
            return -errno.ENOENT
        else:
            os_st = os.lstat(real_path[0])

            stat.st_mode = os_st.st_mode
            stat.st_ino = os_st.st_ino
            stat.st_dev = os_st.st_dev
            stat.st_nlink = os_st.st_nlink
            stat.st_uid = os_st.st_uid
            stat.st_gid = os_st.st_gid
            stat.st_size = os_st.st_size
            stat.st_atime = os_st.st_atime
            stat.st_mtime = os_st.st_mtime
            stat.st_ctime = os_st.st_ctime

            return stat

    def readdir(self, path, offset):
        """
        Read directory.
        """
        logger = logging.getLogger('PyUnionFS.readdir')
        logger.debug('path␣is␣{0},␣offset␣is␣{1}'.format(path, offset
      ))

        dir_entries = ['.', '..']

        if path == '/':
            for branch_path in self.branch_path_list:
                bdir_entries = os.listdir(branch_path)
                for entry in bdir_entries:
                    if entry not in dir_entries:
                        if '.' + entry + '.white_out' not in
      dir_entries:
                            dir_entries.append(entry)
        else:
            list_of_paths = self.resolve_path(path, self.branch_list,
                                              return_paths=True)

            for path_to_dir in list_of_paths:
                bdir_entries = os.listdir(path_to_dir)
                for entry in bdir_entries:
                    if entry not in dir_entries:
                        if '.' + entry + '.white_out' not in
      dir_entries:
                            dir_entries.append(entry)

        for entry in dir_entries:
            yield fuse.Direntry(entry)

    def mkdir(self, path, mode):
        """
        Create a directory.
        """
        logger = logging.getLogger('PyUnionFS.mkdir')
        logger.debug('path␣is␣{0},␣mode␣is␣{1}'.format(path, mode))

        real_path = self.resolve_path(path, self.branch_list, create=
      True)

        if real_path[1] == 'rw':
            os.mkdir(real_path[0], mode)
        else:
            cow_path = self.return_cow_path(path)

            os.mkdir(cow_path, mode)

    def rmdir(self, path):
        """
        Remove a directory.
        """
        logger = logging.getLogger('PyUnionFS.rmdir')
        logger.debug('path␣is␣{0}'.format(path))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            os.rmdir(real_path[0])
        else:
            self.create_white_out(path)

    def read(self, path, size, offset):
        """
        Read data from an open file.
        """
        logger = logging.getLogger('PyUnionFS.read')
        logger.debug('path_is_{0},_size_is_{1},_offset_is_{2}'.format
      (path,
                    size, offset))

        real_path = self.resolve_path(path, self.branch_list)

        with open(real_path[0], 'r') as file:
            file.seek(offset)
            return file.read(size)

    def mknod(self, path, mode, dev):
        """
        Create a file node.
        """
        logger = logging.getLogger('PyUnionFS.mknod')
        logger.debug('path_is_{0},_mode_is_{1},_dev_is_{2}'.format(
      path, mode,
                    dev))

        real_path = self.resolve_path(path, self.branch_list, create=
      True)

        if real_path[1] == 'rw':
            os.mknod(real_path[0], mode, dev)
        else:
            cow_path = self.return_cow_path(path)

            os.mknod(cow_path, mode, dev)

    def write(self, path, buf, offset):
        """
        Write data to an open file.
        """
        logger = logging.getLogger('PyUnionFS.write')
        logger.debug('path_is_{0},_buf_is_{1!r},_offset_is_{2}'.
      format(path,
                    buf, offset))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            with open(real_path[0], 'r+') as file:
                file.seek(offset)
                file.write(buf)
                return len(buf)
        else:
            cow_path = self.return_cow_path(path,
                                            copy=True,
                                            source=real_path[0],
                                            dest=os.path.join(
                                            self.branch_path_list[0],
                                            path[1:]))

            with open(cow_path, 'r+') as file:
                file.seek(offset)
                file.write(buf)
                return len(buf)

    def rename(self, path, path1):
        """
        Rename a file.
        """
        logger = logging.getLogger('PyUnionFS.rename')
        logger.debug('path_is_{0},_path1_is_{1}'.format(path, path1))

        real_path1 = self.resolve_path(path, self.branch_list)
        real_path2 = self.resolve_path(path1, self.branch_list,
                                       create=True)

        if real_path2[1] == 'rw':
            os.rename(real_path1[0], real_path2[0])
        else:
            cow_path = self.return_cow_path(path1)

            os.rename(real_path1[0], cow_path)

    def truncate(self, path, size):
        """
        Change the size of a file.
        """
        logger = logging.getLogger('PyUnionFS.truncate')
        logger.debug('path_is_{0},_size_is_{1}'.format(path, size))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            with open(real_path[0], "a") as file:
                file.truncate(size)
        else:
            cow_path = self.return_cow_path(path,
                                            copy=True,
                                            source=real_path[0],
                                            dest=os.path.join(
                                            self.branch_path_list[0],
                                            path[1:]))

            with open(cow_path, "a") as file:
                file.truncate(size)

    def unlink(self, path):
        """
        Remove a file.
        """
        logger = logging.getLogger('PyUnionFS.unlink')
        logger.debug('path_is_{0}'.format(path))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            os.unlink(real_path[0])
        else:
            self.create_white_out(path)

    def readlink(self, path):
        """
        Read the target of a symbolic link.
        """
        logger = logging.getLogger('PyUnionFS.readlink')
        logger.debug('path_is_{0}'.format(path))

        real_path = self.resolve_path(path, self.branch_list)

        return os.readlink(real_path[0])

    def symlink(self, path, path1):
        """
        Create a symbolic link.
        """
        logger = logging.getLogger('PyUnionFS.symlink')
        logger.debug('path_is_{0},_path1_is_{1}'.format(path, path1))

        real_path1 = self.resolve_path(os.path.normpath(os.path.join(
      '/',
                                       path)),
                                       self.branch_list)
        real_path2 = self.resolve_path(path1, self.branch_list,
                                       create=True)

        if real_path2[1] == 'rw':
            os.symlink(real_path1[0], real_path2[0])
        else:
            cow_path = self.return_cow_path(path1)

            os.symlink(real_path1[0], cow_path)

    def link(self, path, path1):
        """
        Create a hard link to a file.
        """
        logger = logging.getLogger('PyUnionFS.link')
        logger.debug('path_is_{0},_path1_is_{1}'.format(path, path1))

        real_path1 = self.resolve_path(path, self.branch_list)
        real_path2 = self.resolve_path(path1, self.branch_list,
                                       create=True)

        if real_path2[1] == 'rw':
            os.link(real_path1[0], real_path2[0])
        else:
            cow_path = self.return_cow_path(path1)

            os.link(real_path1[0], cow_path)

    def chmod(self, path, mode):
        """
        Change the permission bits of a file.
        """
        logger = logging.getLogger('PyUnionFS.chmod')
        logger.debug('path␣is␣{0},␣mode␣is␣{1}'.format(path, mode))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            os.chmod(real_path[0], mode)
        else:
            cow_path = self.return_cow_path(path,
                                            copy=True,
                                            source=real_path[0],
                                            dest=os.path.join(
                                            self.branch_path_list[0],
                                            path[1:]))

            os.chmod(cow_path, mode)

    def chown(self, path, user, group):
        """
        Change the owner and group of a file.
        """
        logger = logging.getLogger('PyUnionFS.chown')
        logger.debug('path␣is␣{0},␣user␣is␣{1},␣group␣is␣{2}'.format(
      path,
                     user, group))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            os.chown(real_path[0], user, group)
        else:
            cow_path = self.return_cow_path(path,
                                            copy=True,
                                            source=real_path[0],
                                            dest=os.path.join(
                                            self.branch_path_list[0],
                                            path[1:]))

            os.chown(cow_path, user, group)

    def utime(self, path, times):
        """
        Change the access and/or modification times of a file.
        """
        logger = logging.getLogger('PyUnionFS.utime')
        logger.debug('path␣is␣{0},␣times␣is␣{1}'.format(path, times))

        real_path = self.resolve_path(path, self.branch_list)

        if real_path[1] == 'rw':
            os.utime(real_path[0], times)
        else:
            cow_path = self.return_cow_path(path,
                                            copy=True,
                                            source=real_path[0],
                                            dest=os.path.join(
                                            self.branch_path_list[0],
                                            path[1:]))

            os.utime(cow_path, times)

    def statfs(self):
        """
        Get file system statistics.
        """
        logger = logging.getLogger('PyUnionFS.statfs')
        logger.debug('method␣called')

        os.statvfs(self.branch_path_list[0])

    def fsdestroy(self):
        logger = logging.getLogger('PyUnionFS.fsdestroy')
        logger.debug('method␣called')

    def main(self, *args, **kw):
        """
        Filesystem main method
        """
        raw_branch_list = self.branch.split(':')

        for index, path in enumerate(raw_branch_list):
            if path == '':
                del raw_branch_list[index]

        self.branch_list = []

        for item in raw_branch_list:
            splited_item = item.split('#')
            if len(splited_item) == 1:
                self.branch_list.append((os.path.realpath(
       splited_item[0]),
                                        'rw'))
            else:
                self.branch_list.append((os.path.realpath(
       splited_item[0]),
                                        splited_item[1]))

        self.branch_list = collections.OrderedDict(self.branch_list)

        self.branch_path_list = self.branch_list.keys()

        return fuse.Fuse.main(self, *args, **kw)

def main():
    usage="""
PyUnionFS: A simple unification file system.
    """ + fuse.Fuse.fusage

    server = PyUnionFS(version="%prog␣" + fuse.__version__,
                       usage=usage,
                       dash_s_do='setsingle')
    server.parser.add_option(mountopt='branch',
                             metavar=\
"PATH1[#ro/rw]:PATH2[#ro/rw]:PATH3[#ro/rw]:...",
                             default=server.branch,
                             help=\
"list␣of␣paths␣to␣directory␣to␣unify␣separated␣by␣colon")
    server.parser.add_option('-l', '--logging',
                             action="store_true",
                             dest='logging',
                             default=server.logging,
                             help="enable␣logging␣to␣a␣file")
    server.parse(values=server, errex=1)

    if server.logging:
        print 'Start␣logging␣to␣file:␣{0}\n'.format(server.log_file)

    server.main()

if __name__ == '__main__':
    main()
```

APPENDIX E

PYSSHFS'S SOURCE CODE

This chapter presents the full source code of Python Secure SHell File System (PySSHFS).

E.1 PySSHFS's Main Program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#   PySSHFS - FUSE based SFTP Filesystem
#
#   Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import sys
import stat
import errno
import logging
import getpass
import socket

import fuse
import paramiko

fuse.fuse_python_api = (0, 2)

class PySSHStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 2
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 4096
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0

class PySSHFS(fuse.Fuse):
    def __init__(self, *args, **kw):
        """
        Initialize file system class
        """
        fuse.Fuse.__init__(self, *args, **kw)

        self.root = os.path.expanduser('~')

        self.logging = False

        self.log_dir = os.path.expanduser('/tmp/PySSHfs')
        self.log_file = os.path.join(self.log_dir, 'PySSHlog')

        self.host = ''
        self.port = 22
        self.key = None
        self.password = None
        self.auth = 'key'

    def fsinit(self):
        """
        Initialize file system.
        """
        os.chdir(self.root)

        if self.logging:
            if os.path.exists(self.log_dir):
                if os.path.exists(self.log_file):
                    os.rename(self.log_file, os.path.join(self.
log_dir,
                                'PySSHlog.old'))
            else:
                os.makedirs(self.log_dir)

            logging.basicConfig(filename=self.log_file,
                                level=logging.DEBUG,)

            logger = logging.getLogger('PySSHFS.fsinit')
            logger.debug('method called')
            logger.debug('current root path is {0}'.format(self.root)
)

        self.client = paramiko.SSHClient()
        self.client.load_system_host_keys()
        self.client.set_missing_host_key_policy(paramiko.
AutoAddPolicy())
        self.client.connect(hostname=self.hostname, port=int(self.
port),
                            username=self.username, pkey=self.key,
                            password=self.password)

        self.sftp = self.client.open_sftp()

        self.sftp.chdir(self.remote_root)

    def getattr(self, path):
        """
        Get file/folder attributes.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.getattr')
            logger.debug('path is {0}'.format(path))

        stat = PySSHStat()

        os_st = self.sftp.lstat("." + path)

        stat.st_mode = os_st.st_mode
        stat.st_uid = os_st.st_uid
        stat.st_gid = os_st.st_gid
        stat.st_size = os_st.st_size
        stat.st_atime = os_st.st_atime
        stat.st_mtime = os_st.st_mtime

        return stat

    def readdir(self, path, offset):
        """
        Read directory.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.readdir')
            logger.debug('path is {0}, offset is {1}'.format(path,
offset))

        ld = self.sftp.listdir("." + path)

        for r in (".", ".."):
            ld.append(r)

        for i in ld:
            yield fuse.Direntry(i.encode('utf-8'))

    def mkdir(self, path, mode):
        """
        Create a directory.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.mkdir')
            logger.debug('path_is_{0},_mode_is_{1}'.format(path, mode
))

        self.sftp.mkdir("." + path, mode)

    def rmdir(self, path):
        """
        Remove a directory.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.rmdir')
            logger.debug('path_is_{0}'.format(path))

        self.sftp.rmdir("." + path)

    def read(self, path, size, offset):
        """
        Read data from an open file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.read')
            logger.debug('path_is_{0},_size_is_{1},_offset_is_{2}'.
format(path,
                    size, offset))

        file = self.sftp.open("." + path, 'r')
        file.seek(offset)
        buf = file.read(size)
        file.close()

        return buf

    def mknod(self, path, mode, dev):
        """
        Create a file node.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.mknod')
            logger.debug('path_is_{0},_mode_is_{1},_dev_is_{2}'.
format(path,
                        mode, dev))

        file = self.sftp.open("." + path, 'w')
        file.close()

    def write(self, path, buf, offset):
        """
        Write data to an open file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.write')
            logger.debug('path_is_{0},_buf_is_{1!r},_offset_is_{2}'.
format(
                        path, buf, offset))

        file = self.sftp.open("." + path, 'r+')
        file.seek(offset)
        file.write(buf)
        file.close()

        return len(buf)

    def rename(self, path, path1):
        """
        Rename a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.rename')
            logger.debug('path_is_{0},_path1_is_{1}'.format(path,
path1))

        self.sftp.rename("." + path, "." + path1)

    def truncate(self, path, size):
        """
        Change the size of a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.truncate')
            logger.debug('path_is_{0},_size_is_{1}'.format(path, size
))

        file = self.sftp.open("." + path, "a")
        file.truncate(size)

    def unlink(self, path):
        """
        Remove a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.unlink')
            logger.debug('path_is_{0}'.format(path))

        self.sftp.unlink("." + path)

    def readlink(self, path):
        """
        Read the target of a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.readlink')
            logger.debug('path_is_{0}'.format(path))

        return self.sftp.readlink("." + path)

    def symlink(self, path, path1):
        """
        Create a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.symlink')
            logger.debug('path_is_{0},_path1_is_{1}'.format(path,
path1))

        self.sftp.symlink(path, "." + path1)

    def link(self, path, path1):
        """
        Create a hard link to a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.link')
            logger.debug('path_is_{0},_path1_is_{1}'.format(path,
path1))

        return -errno.EPERM

    def chmod(self, path, mode):
        """
        Change the permission bits of a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.chmod')
            logger.debug('path_is_{0},_mode_is_{1}'.format(path, mode
))

        self.sftp.chmod("." + path, mode)

    def chown(self, path, user, group):
        """
        Change the owner and group of a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.chown')
            logger.debug('path_is_{0},_user_is_{1},_group_is_{2}'.
format(path,
                        user, group))
        self.sftp.chown("." + path, user, group)

    def utime(self, path, times):
        """
        Change the access and/or modification times of a file.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.utime')
            logger.debug('path␣is␣{0},␣times␣is␣{1}'.format(path,
   times))

        self.sftp.utime("." + path, times)

    def statfs(self):
        """
        Get file system statistics.
        """
        if self.logging:
            logger = logging.getLogger('PySSHFS.statfs')
            logger.debug('method␣called')

        return -errno.EPERM

    def fsdestroy(self):
        if self.logging:
            logger = logging.getLogger('PySSHFS.fsdestroy')
            logger.debug('method␣called')

        self.sftp.close()
        self.client.close()

    def main(self, *args, **kw):
        """
        Filesystem main method
        """
        host_splited = self.host.split('@')

        if len(host_splited) == 2:
            self.username = host_splited[0]
            host_and_dir = host_splited[1].split(':')
            if len(host_and_dir) == 2:
                self.hostname, self.remote_root = host_and_dir
            else:
                self.hostname = host_and_dir[0]
                self.remote_root = "."
        else:
            self.username = os.environ['USER']
            host_and_dir = host_splited[0].split(':')
            if len(host_and_dir) == 2:
                self.hostname, self.remote_root = host_and_dir
            else:
                self.hostname = host_and_dir[0]
                self.remote_root = "."

        if sys.argv[1] != '-h' and sys.argv[1] != '--help':
            if self.auth == 'key':
                path_rsa = os.path.join(os.environ['HOME'], '.ssh', '
   id_rsa')
                path_dss = os.path.join(os.environ['HOME'], '.ssh', '
   id_dsa')
                if os.path.exists(path_rsa):
                    try:
                        self.key = paramiko.RSAKey.\
                                from_private_key_file(path_rsa)
                    except paramiko.PasswordRequiredException:
                        password = getpass.getpass('RSA␣key␣password:
   ␣')
                        self.key = paramiko.RSAKey.\
                                from_private_key_file(path_rsa,
   password)
                elif os.path.exists(path_dss):
                    try:
                        self.key = paramiko.DSSKey.\
                                from_private_key_file(path_dss)
                    except paramiko.PasswordRequiredException:
                        password = getpass.getpass('RSA␣key␣password:
   ␣')
                        self.key = paramiko.DSSKey.\
                                from_private_key_file(path_dss,
   password)
                else:
                    self.password = getpass.getpass()
            elif self.auth == 'plain':
                self.password = getpass.getpass()

        return fuse.Fuse.main(self, *args, **kw)

def main():
    usage=fuse.Fuse.fusage + "\n" + \
"\nPySSHFS:␣A␣simple␣SFTP␣file␣system."

    server = PySSHFS(version="%prog␣" + fuse.__version__,
                     usage=usage,
                     dash_s_do='setsingle')
    server.multithreaded = False
    server.parser.add_option(mountopt='host',
                             metavar="[USER@]HOST[:DIR]",
                             default=server.host,
                             help="host␣to␣connect␣to")
    server.parser.add_option(mountopt='port',
                             metavar="PORT",
                             default=server.port,
                             help="port␣to␣connect␣to␣(default:␣%
   default)")
    server.parser.add_option(mountopt='auth',
                             metavar="key/plain",
                             default=server.auth,
                             help=\
"which␣authentication␣method␣to␣use␣(default:␣%default)")
    server.parser.add_option('-l', '--logging',
                             action="store_true",
                             dest='logging',
                             default=server.logging,
                             help="enable␣logging␣to␣a␣file")
    server.parse(values=server, errex=1)

    if server.logging:
        print 'Start␣logging␣to␣file:␣{0}\n'.format(server.log_file)

    server.main()

if __name__ == '__main__':
    main()
```

APPENDIX F

SPAFS2'S SOURCE CODE

This chapter presents the full source code of Simple Protocol Agnostic File System 2 (SPAFS2) as well as its IO Modules.

F.1 SPAFS2's Main Program

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

#   SPAFS2 - Simple Protocol Agnostic File System 2
#
#   Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import stat
import errno
import collections
import shutil
import logging
import getpass
import re

import fuse

import local_disk_module
import ssh_network_module

fuse.fuse_python_api = (0, 2)

class SPAStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 2
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 4096
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0

class SPAFS2(fuse.Fuse):
    def __init__(self, *args, **kw):
        """
        Initialize file system class
        """
        fuse.Fuse.__init__(self, *args, **kw)

        self.logging = False

        self.log_dir = os.path.expanduser('/tmp/SPAfs')
        self.log_file = os.path.join(self.log_dir, 'SPAlog')

        self.branch = 'None:None'
        self.cow = False
        self.show_branch = False
        self.raw_pattern_matching = re.compile( '(/\+raw|\+raw)' )
        self.branch_pattern_matching = re.compile( '(\+branch_\d*)' )

    def fsinit(self):
        """
        Initialize file system.
        """
        if self.logging:
            if os.path.exists(self.log_dir):
                if os.path.exists(self.log_file):
                    os.rename(self.log_file, os.path.join(self.
    log_dir,
                              'SPAlog.old'))
            else:
                os.makedirs(self.log_dir)

            logging.basicConfig(filename=self.log_file,
                                level=logging.DEBUG,)

            logger = logging.getLogger('SPAFS2.fsinit')
            logger.debug('method_called')

        for index, items in self.io_modules.iteritems():
            items[0].connect()

        self.fb_io_module = self.io_modules.items()[0][1][0]

    def getattr(self, path):
        """
        Get file/folder attributes.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.getattr')
            logger.debug('path_is_{0}'.format(path))

        stat = SPAStat()

        if self.show_branch:
            path = os.path.normpath(self.raw_pattern_matching.sub('',
        path))
            path = self.branch_pattern_matching.sub('', path)

            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][0].\
                   path_exists(splited_path[0]):

                    os_st = self.io_modules[branch_name][0].\
                            lstat(splited_path[0])

                    stat.st_mode = os_st.st_mode
                    stat.st_uid = os_st.st_uid
                    stat.st_gid = os_st.st_gid
                    stat.st_size = os_st.st_size
                    stat.st_atime = os_st.st_atime
                    stat.st_mtime = os_st.st_mtime

                    return stat
                elif self.fb_io_module.path_exists(splited_path[0]):
                    os_st = self.fb_io_module.lstat(splited_path[0])

                    stat.st_mode = os_st.st_mode
                    stat.st_uid = os_st.st_uid
                    stat.st_gid = os_st.st_gid
                    stat.st_size = os_st.st_size
                    stat.st_atime = os_st.st_atime
                    stat.st_mtime = os_st.st_mtime
                    return stat

        if path == '/':
            os_st = self.fb_io_module.lstat('.')

            stat.st_mode = os_st.st_mode
            stat.st_uid = os_st.st_uid
            stat.st_gid = os_st.st_gid
            stat.st_size = os_st.st_size
            stat.st_atime = os_st.st_atime
            stat.st_mtime = os_st.st_mtime

            return stat
        else:
            for index, items in self.io_modules.iteritems():
                if items[0].path_exists(path):
                    os_st = items[0].lstat(path)

                    stat.st_mode = os_st.st_mode
                    stat.st_uid = os_st.st_uid
                    stat.st_gid = os_st.st_gid
                    stat.st_size = os_st.st_size
                    stat.st_atime = os_st.st_atime
                    stat.st_mtime = os_st.st_mtime

                    return stat

        return -errno.ENOENT

    def readdir(self, path, offset):
        """
        Read directory.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.readdir')
            logger.debug('path␣is␣{0},␣offset␣is␣{1}'.format(path,
      offset))

        dir_entries = ['.', '..']

        show_branch_name = False

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                if splited_path[-1] == 'raw':
                    show_branch_name = True
                    path = splited_path[0]
                else:
                    branch_name = splited_path[-1]

                    branch_dir_entries = \
                    self.io_modules[branch_name][0].listdir(
      splited_path[0])

                    for entry in branch_dir_entries:
                        if entry not in dir_entries:
                            white_out_file = '/.' + entry + '.
      white_out'
                            if not self.io_modules[branch_name][0].\
                                   path_exists(white_out_file):
                                if not show_branch_name:
                                    dir_entries.append(entry)
                                else:
                                    dir_entries.append(entry + '+' +
      index)

        if path == '/':
            for index, items in self.io_modules.iteritems():
                branch_dir_entries = items[0].listdir('.')

                for entry in branch_dir_entries:
                    if entry not in dir_entries:
                        white_out_file = '/.' + entry + '.white_out'
                        if not items[0].path_exists(white_out_file):
                            if not show_branch_name:
                                dir_entries.append(entry)
                            else:
                                dir_entries.append(entry + '+' +
      index)
        else:
            for index, items in self.io_modules.iteritems():
                if items[0].path_exists(path):
                    branch_dir_entries = items[0].listdir(path)

                    for entry in branch_dir_entries:
                        if entry not in dir_entries:
                            white_out_file = '.' + entry + '.
      white_out'
                            real_white_out_path = os.path.join(path,
                                                              white_out_file
      )
                            if not items[0].path_exists(
      real_white_out_path):
                                if not show_branch_name:
                                    dir_entries.append(entry)
                                else:
                                    dir_entries.append(entry + '+' +
      index)

        for entry in dir_entries:
            yield fuse.Direntry(entry)

    def mkdir(self, path, mode):
        """
        Create a directory.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.mkdir')
            logger.debug('path␣is␣{0},␣mode␣is␣{1}'.format(path, mode
      ))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].mkdir(
      splited_path[0],
                                                         mode)

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    self.fb_io_module.mkdir(splited_path[0], mode)

                    return

        entry_path = os.path.dirname(path)

        if entry_path == '/':
            self.fb_io_module.mkdir(path, mode)
        else:
            for index, items in self.io_modules.iteritems():
                if items[0].path_exists(entry_path):
                    if items[1] == 'rw':
                        items[0].mkdir(path, mode)
                    else:
                        if not self.fb_io_module.path_exists(
      entry_path):
                            self.fb_io_module.makedirs(entry_path)

                        self.fb_io_module.mkdir(path, mode)

    def rmdir(self, path):
        """
        Remove a directory.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.rmdir')
            logger.debug('path_is_{0}'.format(path))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].rmdir(
    splited_path[0])

                    return
                else:
                    self.io_modules[branch_name][0].\
                    create_whiteout(splited_path[0])

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].rmdir(path)
                else:
                    items[0].create_whiteout(path)

    def read(self, path, size, offset):
        """
        Read data from an open file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.read')
            logger.debug('path_is_{0},_size_is_{1},_offset_is_{2}'.
    format(path,
                          size, offset))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                return self.io_modules[branch_name][0].read(
    splited_path[0],
                                                         size,
    offset)

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                return items[0].read(path, size, offset)

    def mknod(self, path, mode, dev):
        """
        Create a file node.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.mknod')
            logger.debug('path_is_{0},_mode_is_{1},_dev_is_{2}'.
    format(path,
                          mode,
                          dev))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].mknod(
    splited_path[0],
                                                       mode,
                                                       dev)

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    self.fb_io_module.mknod(splited_path[0], mode,
    dev)

                    return

        entry_path = os.path.dirname(path)

        if entry_path == '/':
            self.fb_io_module.mknod(path, mode, dev)
        else:
            for index, items in self.io_modules.iteritems():
                if items[0].path_exists(entry_path):
                    if items[1] == 'rw':
                        items[0].mknod(path, mode, dev)
                    else:
                        if not self.fb_io_module.path_exists(
    entry_path):
                            self.fb_io_module.makedirs(entry_path)

                        self.fb_io_module.mknod(path, mode, dev)

    def write(self, path, buf, offset):
        """
        Write data to an open file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.write')
            logger.debug('path_is_{0},_offset_is_{2}'.\
                         format(path, buf, offset))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].write(
    splited_path[0],
                                                         buf, offset
    )

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(splited_path[0], 'r')
                    file_dst = self.fb_io_module.open(splited_path
    [0], 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.write(splited_path[0], buf,
    offset)

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    return items[0].write(path, buf, offset)
                else:
                    entry_path = os.path.dirname(path)

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(path, 'r')
                    file_dst = self.fb_io_module.open(path, 'w')

                    shutil.copyfileobj(file_src, file_dst)
                return self.fb_io_module.write(path, buf, offset)

    def rename(self, src_path, dst_path):
        """
        Rename a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.rename')
            logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(src_path,
                                                             dst_path)
  )

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    dst_entry_path = os.path.dirname(dst_path)

                    if dst_entry_path == '/':
                        self.io_modules[branch_name][0].rename(
  src_path,
                                                                dst_path
  )
                    elif self.io_modules[branch_name][0].\
                         path_exists(dst_entry_path):
                        self.io_modules[branch_name][0].rename(
  src_path,
                                                                dst_path
  )
                    else:
                        return -errno.EPERM
                else:
                    return -errno.EROFS

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(src_path):
                if items[1] == 'rw':
                    dst_entry_path = os.path.dirname(dst_path)

                    if dst_entry_path == '/':
                        items[0].rename(src_path, dst_path)
                    elif items[0].path_exists(dst_entry_path):
                        items[0].rename(src_path, dst_path)
                    else:
                        return -errno.EPERM
                else:
                    return -errno.EROFS

    def truncate(self, path, size):
        """
        Change the size of a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.truncate')
            logger.debug('path␣is␣{0},␣size␣is␣{1}'.format(path, size
  ))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].truncate(
  splited_path[0],
                                                              size)

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(splited_path[0], 'r')
                    file_dst = self.fb_io_module.open(splited_path
  [0], 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.truncate(splited_path[0], size)

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].truncate(path, size)
                else:
                    entry_path = os.path.dirname(path)

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(path, 'r')
                    file_dst = self.fb_io_module.open(path, 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.truncate(path, size)

    def unlink(self, path):
        """
        Remove a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.unlink')
            logger.debug('path␣is␣{0}'.format(path))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].unlink(
  splited_path[0])

                    return
                else:
                    self.io_modules[branch_name][0].\
                    create_whiteout(splited_path[0])

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].unlink(path)
                else:
                    items[0].create_whiteout(path)

    def readlink(self, path):
        """
        Read the target of a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.readlink')
            logger.debug('path␣is␣{0}'.format(path))

        return -errno.ENOSYS

    def symlink(self, path, path1):
        """
        Create a symbolic link.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.symlink')
            logger.debug('path␣is␣{0},␣path1␣is␣{1}'.format(path,
  path1))
```

```
        return -errno.ENOSYS

    def link(self, path, path1):
        """
        Create a hard link to a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.link')
            logger.debug('path_is_{0},_path1_is_{1}'.format(path,
    path1))

        return -errno.ENOSYS

    def chmod(self, path, mode):
        """
        Change the permission bits of a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.chmod')
            logger.debug('path_is_{0},_mode_is_{1}'.format(path, mode
    ))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].chmod(
    splited_path[0],
                                                           mode)

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(splited_path[0], 'r')
                    file_dst = self.fb_io_module.open(splited_path
    [0], 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.chmod(splited_path[0], mode)

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].chmod(path, mode)
                else:
                    entry_path = os.path.dirname(path)

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    # Copy file between branches
                    file_src = items[0].open(path, 'r')
                    file_dst = self.fb_io_module.open(path, 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.chmod(path, mode)

    def chown(self, path, user, group):
        """
        Change the owner and group of a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.chown')
            logger.debug('path_is_{0},_user_is_{1},_group_is_{2}'.
    format(path,
                      user, group))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].chown(
    splited_path[0],
                                                           user, group
    )

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(splited_path[0], 'r')
                    file_dst = self.fb_io_module.open(splited_path
    [0], 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.chown(splited_path[0], user,
    group)

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].chown(path, user, group)
                else:
                    entry_path = os.path.dirname(path)

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(path, 'r')
                    file_dst = self.fb_io_module.open(path, 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.chown(path, user, group)

    def utime(self, path, times):
        """
        Change the access and/or modification times of a file.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.utime')
            logger.debug('path_is_{0},_times_is_{1}'.format(path,
    times))

        if self.show_branch:
            splited_path = path.split('+')

            if len(splited_path) > 1:
                branch_name = splited_path[-1]

                if self.io_modules[branch_name][1] == 'rw':
                    self.io_modules[branch_name][0].utime(
    splited_path[0],
                                                           times)

                    return
                else:
                    entry_path = os.path.dirname(splited_path[0])

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(splited_path[0], 'r')
                    file_dst = self.fb_io_module.open(splited_path
    [0], 'w')
```

```
                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.utime(splited_path[0], times)

                    return

        for index, items in self.io_modules.iteritems():
            if items[0].path_exists(path):
                if items[1] == 'rw':
                    items[0].utime(path, times)
                else:
                    entry_path = os.path.dirname(path)

                    if not self.fb_io_module.path_exists(entry_path):
                        self.fb_io_module.makedirs(entry_path)

                    file_src = items[0].open(path, 'r')
                    file_dst = self.fb_io_module.open(path, 'w')

                    shutil.copyfileobj(file_src, file_dst)

                    self.fb_io_module.utime(path, times)

    def statfs(self):
        """
        Get file system statistics.
        """
        if self.logging:
            logger = logging.getLogger('SPAFS2.statfs')
            logger.debug('method_called')

        self.fb_io_module.statfs()


    def fsdestroy(self):
        if self.logging:
            logger = logging.getLogger('SPAFS2.fsdestroy')
            logger.debug('method_called')

        for index, items in self.io_modules.iteritems():
            items[0].close_connection()

    def main(self, *args, **kw):
        """
        Filesystem main method
        """
        raw_branch_list = self.branch.split('#')

        for index, address in enumerate(raw_branch_list):
            if address == '':
                del raw_branch_list[index]

        address_list = []

        for index, item in enumerate(raw_branch_list):
            splited_items = item.split(':')

            if len(splited_items) > 2:
                if self.cow:
                    address_list.append((index,               # Index
                                         (splited_items[0],   #
      Protocol
                                         splited_items[1],    # Address
                                         splited_items[2])))  #
      Permission
                else:
                    address_list.append((index,
                                         (splited_items[0],
                                         splited_items[1],
                                         'rw')))
            else:
                address_list.append((index,
                                     (splited_items[0],
                                     splited_items[1],
                                     'rw')))

        addresses = collections.OrderedDict(address_list)
```

```
        io_module_list = []

        for index, items in addresses.iteritems():
            # For Local Disk Module
            if items[0] == 'local':
                real_path = os.path.realpath(items[1])

                disk_io_module = local_disk_module.DiskIO(real_path)

                io_module_list.append(('branch_' + str(index), #
      Index
                                     (disk_io_module,          # IO
      module
                                     items[2])))               #
      Permission

            if items[0] == 'ssh':
                pkey = None

                host_splited = items[1].split('@')

                if len(host_splited) == 2:
                    username = host_splited[0]

                    conn_items = host_splited[1].split('%')
                else:
                    username = os.environ['USER']

                    conn_items = host_splited[0].split('%')

                if len(conn_items) > 3:
                    host = conn_items[0]      # Host address
                    auth = conn_items[1]      # Authenticate method
                    port = conn_items[2]      # Port
                    root_dir = conn_items[3] # Remote root path
                else:
                    # If no dir is given, default to user's home.
                    host = conn_items[0]      # Host address
                    auth = conn_items[1]      # Authenticate method
                    port = conn_items[2]      # Port
                    root_dir = '.'            # Remote root path

                if auth == 'key':
                    path_rsa = os.path.join(os.environ['HOME'], '.ssh
      ',
                                            'id_rsa')
                    path_dss = os.path.join(os.environ['HOME'], '.ssh
      ',
                                            'id_dsa')
                    if os.path.exists(path_rsa):
                        try:
                            pkey = ssh_network_module.paramiko.RSAKey
      .\
                                   from_private_key_file(path_rsa)
                        except ssh_network_module.paramiko.\
                               PasswordRequiredException:
                            password = getpass.getpass('RSA_key_
      password:_')
                            pkey = ssh_network_module.paramiko.RSAKey
      .\
                                   from_private_key_file(path_rsa,
                                                         password)
                    elif os.path.exists(path_dss):
                        try:
                            pkey = ssh_network_module.paramiko.DSSKey
      .\
                                   from_private_key_file(path_dss)
                        except ssh_network_module.paramiko.\
                               PasswordRequiredException:
                            password = getpass.getpass('DSS_key_
      password:_')
                            pkey = ssh_network_module.paramiko.DSSKey
      .\
                                   from_private_key_file(path_dss,
                                                         password)
                elif auth == 'plain':
```

```
                password = getpass.getpass('Password for user {0}: '\
                                            .format(username))

                ssh_io_module = ssh_network_module.SSH(host, port, username,
                                                        pkey, password,
                                                        root_dir)

                io_module_list.append(('branch_' + str(index), # Index
                                       (ssh_io_module,          # IO module
                                        items[2])))             # Permission

        self.io_modules = collections.OrderedDict(io_module_list)

        return fuse.Fuse.main(self, *args, **kw)

def main():
    usage=fuse.Fuse.fusage + \
"""
SPAFS2: Simple Protocol Agnostic File System 2."""

    server = SPAFS2(version="%prog " + fuse.__version__,
                    usage=usage,
                    dash_s_do='setsingle')
    server.multithreaded = False
    server.parser.add_option(mountopt='branch',
                             metavar=\
"PROTOCOL:ADDRESS1[:rw/ro]#PROTOCOL:ADDRESS2[:rw/ro]#...",
                             default=server.branch,
                             help=\
"list of branch to unify separated by hash")
    server.parser.add_option('--cow',
                             action="store_true",
                             dest='cow',
                             default=server.cow,
                             help=\
"enable copy-on-write (warning: experimental)")
    server.parser.add_option('--branch-tag',
                             action="store_true",
                             dest='show_branch',
                             default=server.show_branch,
                             help=\
"enable branch tag (warning: experimental)")
    server.parser.add_option('-l', '--logging',
                             action="store_true",
                             dest='logging',
                             default=server.logging,
                             help="enable logging to a file")
    server.parse(values=server, errex=1)

    if server.logging:
        print 'Start logging to file: {0}\n'.format(server.log_file)

    server.main()

if __name__ == '__main__':
    main()
```

F.2 SPAFS2's Local Disk IO Module

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

# Local Disk Module
#
# Part of SPAFS2
#
# Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os

class DiskIO(object):
    def __init__(self, root_path):
        self.root_path = root_path

    def connect(self):
        return self.root_path

    def close_connection(self):
        return True

    def path_exists(self, path, check_path_dirname=False):
        real_path = os.path.join(self.root_path, path[1:])

        if check_path_dirname:
            entry_path = os.path.dirname(path)
            real_entry_path = os.path.join(self.root_path, entry_path
[1:])

            return os.path.exists(real_entry_path)
        else:
            return os.path.exists(real_path)

    def makedirs(self, path, mode=0777):
        path_created = ''
        splited_path = path.split('/')

        for folder in splited_path:
            if folder == '':
                continue

            if path_created == "":
                first_folder = os.path.join(self.root_path, folder)

                try:
                    os.mkdir(first_folder, mode)
                except OSError:
                    pass

                path_created = first_folder
            else:
                path_created = os.path.join(path_created, folder)

                try:
                    os.mkdir(path_created, mode)
                except OSError:
                    pass

    def create_whiteout(self, path):
        entry_name = os.path.basename(path)
        entry_path = os.path.dirname(path)

        real_path = os.path.join(self.root_path, entry_path[1:])

        white_out_file = '.' + entry_name + '.white_out'
        real_white_out_path = os.path.join(real_path, white_out_file)

        os.mknod(real_white_out_path)

    def open(self, path, mode):
        real_path = os.path.join(self.root_path, path[1:])

        return open(real_path, mode)

    def lstat(self, path):
        real_path = os.path.join(self.root_path, path[1:])

        return os.lstat(real_path)
    def listdir(self, path):
        real_path = os.path.join(self.root_path, path[1:])

        return os.listdir(real_path)

    def mkdir(self, path, mode):
        real_path = os.path.join(self.root_path, path[1:])

        os.mkdir(real_path, mode)

    def rmdir(self, path):
        real_path = os.path.join(self.root_path, path[1:])

        os.rmdir(real_path)

    def read(self, path, size, offset):
        real_path = os.path.join(self.root_path, path[1:])

        with open(real_path, 'r') as file:
            file.seek(offset)
            return file.read(size)

    def mknod(self, path, mode, dev):
        real_path = os.path.join(self.root_path, path[1:])

        os.mknod(real_path, mode, dev)

    def write(self, path, buf, offset):
        real_path = os.path.join(self.root_path, path[1:])

        with open(real_path, 'r+') as file:
            file.seek(offset)
            file.write(buf)
            return len(buf)

    def rename(self, current_path, new_path):
        real_current_path = os.path.join(self.root_path, current_path
      [1:])
        real_new_path = os.path.join(self.root_path, new_path[1:])

        os.rename(real_current_path, real_new_path)

    def truncate(self, path, size):
        real_path = os.path.join(self.root_path, path[1:])

        with open(real_path, 'a') as file:
            file.truncate(size)

    def unlink(self, path):
        real_path = os.path.join(self.root_path, path[1:])

        os.unlink(real_path)

    def readlink(self, path):
        real_path = os.path.join(self.root_path, path[1:])

        return os.readlink(real_path)

    def symlink(self, source, link_name):
        real_source = os.path.join(self.root_path, source[1:])
        real_link_name = os.path.join(self.root_path, link_name[1:])

        os.symlink(real_source, real_link_name)

    def link(self, source, link_name):
        real_source = os.path.join(self.root_path, source[1:])
        real_link_name = os.path.join(self.root_path, link_name[1:])

        os.link(real_source, real_link_name)

    def chmod(self, path, mode):
        real_path = os.path.join(self.root_path, path[1:])

        os.chmod(real_path, mode)

    def chown(self, path, user, group):
        real_path = os.path.join(self.root_path, path[1:])

        os.chown(real_path, user, group)

    def utime(self, path, times):
        real_path = os.path.join(self.root_path, path[1:])

        os.utime(real_path, times)

    def statfs(self):
        return os.statvfs(self.root_path)
```

F.3 SPAFS2's SSH Network IO Module

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

# SSH Network Module
#
# Part of SPAFS2
#
# Author: Smith Dhumbumroong <zodmaner@gmail.com>

import os
import paramiko
import errno

class SSH(object):
    def __init__(self, hostname, port, username, pkey, password,
      root_path):
        self.hostname = hostname
        self.port = port
        self.username = username
        self.pkey = pkey
        self.password = password
        self.root_path = root_path

    def connect(self):
        self.client = paramiko.SSHClient()
        self.client.load_system_host_keys()
        self.client.set_missing_host_key_policy(paramiko.
      AutoAddPolicy())
        self.client.connect(hostname=self.hostname, port=int(self.
      port),
                            username=self.username, pkey=self.pkey,
                            password=self.password)

        self.sftp = self.client.open_sftp()

        self.sftp.chdir(self.root_path)

    def close_connection(self):
        self.sftp.close()
        self.client.close()

    def path_exists(self, path):
        try:
            if self.sftp.stat(path[1:]):
                return True
        except IOError:
            return False

    def makedirs(self, path, mode=0777):
        path_created = ''
        splited_path = path.split('/')

        for folder in splited_path:
            if folder == '':
                continue

            if path_created == "":
                first_folder = folder

                try:
                    self.sftp.mkdir(first_folder, mode)
                except OSError:
                    pass

                path_created = first_folder
            else:
                path_created = os.path.join(path_created, folder)

                try:
                    self.sftp.mkdir(path_created, mode)
                except OSError:
                    pass

    def create_whiteout(self, path):
        entry_name = os.path.basename(path)
        entry_path = os.path.dirname(path[1:])

        white_out_file = '.' + entry_name + '.white_out'
        real_white_out_path = os.path.join(entry_path, white_out_file
       )

        self.sftp.mknod(real_white_out_path)

    def open(self, path, mode):
        return self.sftp.open(path[1:], mode)

    def lstat(self, path):
        return self.sftp.lstat(path[1:])

    def listdir(self, path):
        return self.sftp.listdir(path[1:])

    def mkdir(self, path, mode):
        self.sftp.mkdir(path[1:], mode)

    def rmdir(self, path):
        self.sftp.rmdir(path[1:])

    def read(self, path, size, offset):
        file = self.sftp.open(path[1:], 'r')
        file.seek(offset)
        buf = file.read(size)
        file.close()

        return buf

    def mknod(self, path, mode, dev):
        file = self.sftp.open(path[1:], 'w')
        file.close()

    def write(self, path, buf, offset):
        file = self.sftp.open(path[1:], 'r+')
        file.seek(offset)
        file.write(buf)
        file.close()

        return len(buf)

    def rename(self, current_path, new_path):
        self.sftp.rename(current_path[1:], new_path[1:])

    def truncate(self, path, size):
        file = self.sftp.open(path[1:], "a")
        file.truncate(size)
        file.close()

    def unlink(self, path):
        self.sftp.unlink(path[1:])

    def readlink(self, path):
        return self.sftp.readlink(path[1:])

    def symlink(self, source, link_name):
        return errno.ENOSYS

    def link(self, source, link_name):
        return errno.ENOSYS

    def chmod(self, path, mode):
        self.sftp.chmod(path[1:], mode)

    def chown(self, path, user, group):
        self.sftp.chown(path[1:], user, group)

    def utime(self, path, times):
        self.sftp.utime(path[1:], times)

    def statfs(self):
        return errno.ENOSYS
```

APPENDIX G

LIST OF PUBLICATIONS

Parts of this work are published in the following article.

International Conference Proceedings

1. Smith Dhumbumroong and Krerk Piromsopa, “Personal Cloud Filesystem: A distributed unification filesystem for personal computer and portable device”, Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), Thailand, 2011

Biography

Smith Dhumbumroong was born in Bangkok, Thailand, on April, 1985. He received Bachelor of Arts in Economics from Thammasat University, Thailand, on 2007.