INF3981
MASTER'S THESIS IN
COMPUTER SCIENCE

Integrating libpesto with subversion

Oleg Jakobsen

June 15, 2007

Faculty of Science
Department of Computer Science
University of Tromsø
N-9037 Tromsø, Norway

INF-3981
MASTER’S THESIS IN COMPUTER SCIENCE

Integrating libpesto with subversion

Oleg Jakobsen
<oleg.jakobsen@gmail.com>

June 15, 2007

Abstract

Subversion, an open-source centralized version control system, developed by CoallabNet, is currently the second most popular version control system, after the ever popular CVS. Like CVS, Subversion uses a client-server architecture, but has a cleaner, modular architecture. One set of subversion modules, are the filesystem backends modules of subversion. Two “official” backends are currently supplied with subversion, a berkleyDB based backend(bdb), and a custom filebased filesystem implementation (fsfs). At least another un-official backend module using an SQL-database exists.

Pesto is a secure, decentralized, distributed peer-to-peer storage system, implemented both as a NetBSD filesystem, and as middelware, a portable C-library (libpesto). Currently two applications using libpesto have been written, a C#.Net windows client, and a highly scalable serverfarm for Pesto, written in Java.

In this project we integrate subversion with pesto, by creating a new filesystem backend for subversion using libpesto. The result is a version-control system, that works like a centralized version-control system, but has decentralized storage. We show that this system can be used for backup, mirroring of repositories, and as a decentralized version-control system.

Acknowledgments

First of all I would like to thank my supervisors, Feike Dillema and Anders Andersen, for supervising me, and helping me put the finishing touches on this document. I would also thank Feike for his help in debugging pfs, helping me getting started with pesto, and for implementing new functionality in libpesto when it was required for pfs. I wish you the best of luck in your future work on pesto. I would also like to thank Weihai Yu, for his insights on implementing transactions, which I unfortunately had to cut from pfs, after I decided to base the implementation of fsfs.

Thanks to the students and staff at the Department of Computer Science at the University of Tromsø for creating a great environment for studying computer science. These five years have passed faster than what I though possible.

Last, but not least, I would like to thank my family for supporting me during my studies, and encouraging me to pursue a higher education. I would not have reached this far without your help.

Contents

1 Introduction **3**

1.1 Background . . . 3

1.2 Problem statement . . . 4

1.2.1 Elaboration . . . 5

1.3 Methodology . . . 5

1.4 Outline . . . 5

2 Version control software **7**

2.1 Introduction . . . 7

2.1.1 Delta compression . . . 7

2.1.2 Storage model . . . 8

2.1.3 Terminology . . . 10

2.2 SCCS and RCS . . . 10

2.3 CVS . . . 11

2.4 Subversion . . . 12

2.5 Darcs . . . 12

2.6 GIT . . . 13

3 Pesto **15**

3.1 Usage scenarios for pesto . . . 15

3.2 Architecture . . . 16

3.2.1 Files . . . 16

3.2.2 Policies . . . 18

3.2.3 Trusted bases . . . 18

3.2.4 p2p . . . 19

4 Subversion **21**

4.1 Features . . . 21
4.2 Layers . . . 23
4.3 Modules . . . 25

5 Requirements **27**

5.1 Usage scenarios . . . 27
5.1.1 Subversion server . . . 27
5.1.2 Replication . . . 27
5.1.3 Collaboration . . . 28
5.1.4 Policies . . . 29
5.1.5 Mirroring . . . 29
5.1.6 Decentralized source control . . . 31
5.1.7 Concurrency control . . . 31
5.2 Functional requirements . . . 32
5.2.1 commit . . . 33
5.2.2 checkout . . . 33
5.2.3 update . . . 33
5.2.4 propset . . . 33
5.2.5 propget . . . 33
5.2.6 add . . . 33
5.2.7 delete . . . 34
5.2.8 move . . . 34
5.2.9 copy . . . 34
5.2.10 lock . . . 34
5.2.11 diff . . . 34
5.2.12 log . . . 34
5.3 Non-functional requirements . . . 35
5.4 Target platform . . . 35
5.5 Alternatives . . . 35

6 The fsfs backend **37**
6.1 Nodes and node-revisions 37
6.2 Skip-deltas and data storage 38
6.3 Transactions 39
6.4 Revisions 40
6.5 fsfs layers 40
6.5.1 The file-layer 41
6.5.2 The dag-layer 41
6.5.3 The tree-layer 41

7 Design **43**
7.1 Nodes 44
7.1.1 Noderevision meta-object 45
7.1.2 node-properties 45
7.1.3 node-data 45
7.2 Revisions 47
7.3 Transactions 50
7.4 pesto abstraction 50
7.4.1 Committing a transaction 53
7.4.2 Checking out a revision 53

8 Implementation **55**
8.1 Identifiers 55
8.2 `pfs_vault` 55
8.2.1 Transactions 56
8.2.2 Revisions 57
8.2.3 Node-revisions 58
8.2.4 Representations 60
8.2.5 Caching 61
8.3 Working with transactions 61
8.3.1 Locking 61
8.3.2 Committing a transaction 62
8.3.3 Aborting a transaction 62
8.4 Interfacing with pesto 63

8.4.1 Initializing the pesto backing directory . . . 63
8.4.2 Writing an object . . . 64
8.4.3 Accessing an object with a GUID . . . 64
8.4.4 Accessing a version of an object . . . 64
8.4.5 Finding revision roots . . . 64
8.5 Functionality . . . 66

9 Testing **69**
9.1 Demonstration . . . 69
9.1.1 Demo configuration . . . 69
9.1.2 Normal subversion operations . . . 69
9.1.3 Collaboration . . . 72
9.1.4 Mirroring . . . 73
9.1.5 Decentralized subversion . . . 73
9.2 Testing . . . 74

10 Discussion **79**

11 Future work **81**
11.1 Performance improvements . . . 81
11.1.1 Caching . . . 81
11.1.2 Ramfs . . . 82
11.2 User-friendly features . . . 82
11.2.1 Error handling . . . 82
11.2.2 Installation . . . 82
11.2.3 Repository creation . . . 83
11.2.4 Policy interface . . . 83
11.2.5 Revision dependencies . . . 83
11.3 Distributed/decentralized subversion . . . 83
11.3.1 Consistency control . . . 84
11.3.2 Branching à-la git . . . 85

12 Conclusion **87**

A The CDROM **1**

B Installation instructions **3**

B.1 Building libpesto . . . 3

B.1.1 Building cwebx . . . 3

B.1.2 Building libpesto . . . 3

B.2 Building subversion . . . 3

B.2.1 Required dependencies . . . 3

B.2.2 Building subversion with pfs . . . 4

C Usage **5**

C.1 Creating a repository . . . 5

C.1.1 Creating a repository . . . 5

C.1.2 Creating a mirror repository . . . 5

C.2 Normal use . . . 6

C.2.1 Checking out a repository . . . 6

C.2.2 Updating a repository . . . 6

C.2.3 Committing changes to the repository . . . 6

C.2.4 Getting the log history of a file . . . 6

C.2.5 Get changes between two revisions for a file . . . 6

C.3 Get the GUID of a file stored in an pfs repository . . . 7

List of Figures

2.1 The file modifications F_a, F_b and F_c are merged back with the original file. 9

3.1 Conceptual view of a pesto node. 17
3.2 The update tree for a file 17
3.3 The pesto-node 'A' sends a message to 'B' 19

4.1 Subversion layers 24
4.2 The component parts of subversion 25

6.1 fsfs overview 40

7.1 Timeline for a node 44
7.2 Translation of a pfs node-revision into pesto and subversion objects 46
7.3 The noderevision meta-object 46
7.4 The node properties-object 47
7.5 The node data-object 47
7.6 pfs revision 48
7.7 The revision-anchor object 49
7.8 Translation of a pfs revision-anchor into pesto and subversion objects 49
7.9 pfs transaction 51
7.10 The transaction-anchor object 52
7.11 Translation of a pfs transaction-anchor into files and subversion objects. 52
7.12 pfs overview 54

8.1 The `vault_txn_t` structure. 56
8.2 The `vault_data_t` structure. 57
8.3 The `vault_rev_t` structure. 57
8.4 The `vault_noderev_t` structure used by `pfs_vault`. 58

8.5 The `node_revision_t` structure used by subversion. 59
8.6 The `vault_representation_t` structure used by `pfs_vault`. 59
8.7 The `representation_t` structure used by subversion. 60
8.8 The lookup-table with revsion anchors and the root node-revision. 65

List of Tables

7.1 Vault operations 53

9.1 Configuration of *hydra* 69

9.2 Configuration of *pastaws0* 70

9.3 Statistics for the pesto cvs repository, as reported by `cvs2svn`. 75

9.4 Results from the tests 76

9.5 Results from using pfs over mfs and writing to disk 77

9.6 The most used functions when committing and checking out from a pfs repository as shown by gprof. 77

Chapter 1

Introduction

1.1 Background

Pesto has been created as part of the PENNE project, which itself is part of the PASTA project at the University of Tromsø. The motivation of the project has been to provide secure resources for mobile devices with limited resources.

Pesto is a system that provides mobile, and stationary devices with storage resources. The guiding principle of pesto is that its users will have varying levels of trust to other users of pesto. Depending on the level of trust between users, they can negotiate contracts with each other (outside of pesto), where they offer each other services on their machines, mainly access to storage resources. Pesto supports disconnected operation; files can be shared, and new updates created by users that are offline. Pesto users can also operate in ad-hoc networks between a few users, that are disconnected from the rest of the world. Pesto has, so far, been implemented as a NETBSD filesystem, using user-level tools(e-mail) to exchange pesto messages, and manage files stored in pesto, and more recently, as a user-level C-library, libpesto.

Although there have been written a few applications that use libpesto, they have dealt directly with pesto, not having real-world use outside of pesto. We want to see how feasible it is to have a real-world application use pesto. First of all, because it shows that pesto is a good idea, and secondly to be able to find bugs that have crept into pesto.

Version control software seem to be a fine candidate for a real-world application that can benefit from using pesto. Version control systems usually retain the history of objects that are added to it. Pesto never deletes a file, and file updates in pesto are distributed as new versions of the file. Modern version control software is written to be distributed, having several repositories that share the same source. These systems are hard to use[7, 10] compared to the old type of centralized version control systems. By using pesto as the backend for a centralized version control system it can get many of the properties of a distributed version control system, while still being as easy to use as a centralized system.

Such a system has many benefits. It can be used to provide automatic backups to without any user intervention or backup-scripts. As new files are added to pesto, they can be replicated to backup storage. Similarly it can be used to create mirrors of a repository automatically. Popular open-source projects are often distributed in source form. Often as source-tarballs.

However, some projects can become popular while still being developed so fast that users like to be on the bleeding edge of development[1], and prefer to download the current source as it is. Usually the ratio of users contra developers is quite high. To handle the load of many users trying to get the latest sources, mirror repositories have to be set up. Mirroring is usually done by mirror repositories that pull the central repository for the latest versions several times a day, which is ineffective, since new updates can be added between updates, and mirrors also poll the repository when there are no updates. Pesto instead pushes out new versions of files as soon as they are added. This means that the mirror repositories would be updated almost instantly. Another useful feature is to use a libpesto backed version control system for mobile users. A user that is disconnected while for instance traveling on a plane, might still want to work with the data stored in repository. Traditionally the user would have to get the latest sources before leaving, make the desired changes, and then when connectivity is reestablished, commit the changes to the repository. A pesto backed repository could allow the user to commit and view the repository history while disconnected. And no explicit step to fetch the latest sources before departure would be necessary, since the up to date files would already have been pushed to the users laptop. Finally, there is no reason as to why objects in the repository only should be accessed through the version control client. libpesto provides other means of access to data stored in pesto. Professions that usually do not work with version control systems, like for instance graphic designers, could benefit from it in a collaboration project between programmers and graphic designers, for instance when creating a computer game. Or the situation could be turned around, and pesto used as a delivery/deployment mechanism for the software project in the version control system. The compiled binaries could be put in version control with the source, and a pesto client, based on libpesto could be used to obtain the latest version automatically, when such is available.

This project is being based on subversion. Subversion is a version control system, built to be a modern replacement for the Concurrent Version System (CVS). Like CVS it is a centralized system. It is one of the most popular version control systems. It is written in C, which makes it ideal for integration with libpesto, which is also written in C, since no wrappers/compatibility layers need to be created. Furthermore, subversion has a clean, modular layered-design, which allows us to easily replace components in subversion, without affecting the correctness of the rest of the code.

Basing this project on an already existing distributed version control system might seem like a good idea, but one must keep in mind that these projects have their own idea of how the versioned data should be distributed, and are based around that idea, and their own security features that partially overlap with the functionality of libpesto. Adapting these projects to work with libpesto could as well result in more work than distributing a centralized system.

Finally, a distributed subversion already exists, svk[11, 12], which shows that it is possible to create a distributed version controls system, based on subversion.

1.2 Problem statement

In this thesis we investigate how pesto can be used to implement a versioning storage system, and design and implement such a system using the already existing centralized

[1] The NetBSD-project uses CVS as its primary form of distribution. Users upgrade their systems by downloading and sources from CVS and rebuilding the whole system.

source control system, Subversion.

1.2.1 Elaboration

We design and implement a new filesystem-backend for subversion that uses libpesto to store its data, and perform tests to evaluate its performance and functionality in comparison to the original filesystem backend. We design the system so that it can in the future be extended to work as a peer-to-peer decentralized version-control system, and discuss some aspects of that. However, we do not implement that part of the system, because of limited time, and the not yet finished peer-to-peer part of libpesto.

1.3 Methodology

The main work in this project consists of integrating two already existing software projects. This is different from the traditional software development process, in that most of the time is spent analyzing the already existing codebase, to be able to better understand how to solve the problem. This in turn, means that the solution is bound to be dependent on the implementation of subversion, and small implementation details of subversion can cause large changes in the design of the new system. A initial design phase where the system is carefully designed is therefore impractical. Instead of using a traditional software development framework, we use what we would call *exploratory programming*. Instead of writing code after a given specification, we first try out different ideas, and generally "play" with the existing code of subversion and pesto, to get an impression of what the existing codebase accomplishes. Then starting the implementation with a vague idea of what actually has to be changed, implementing new functionality as needed.

1.4 Outline

The rest of this thesis is organized as follows:

Chapter 2 - A short description of the various version control terminology and techniques, and a short survey of some of the most significant version control systems.

Chapter 3 - Takes a look at the pesto middelware, its functionality and limitations.

Chapter 4 - A closer look at subversion. Here we familiarize ourselves with subversion and how it works. We take a look at the most commonly used components and how they interact.

Chapter 5 - Requirements. Here we analyze the problem and specify the requirements for solving it.

Chapter 6 - Here we take a closer look at the `fsfs` filesystem backend used by subversion. This is necessary, since our own filesystem backend, `pfs` is based on `fsfs`.

Chapter 7 - Design. Gives a conceptual view of how `pfs` should work.

Chapter 8 - Implementation. Describes how pfs is implemented, and details the implementation details for complicated cases.

Chapter 9 - Testing. Here we demonstrate the functionality of pfs, and perform some simple tests to evaluate its performance.

Chapter 10 - Summarizes the achievements we have made.

Chapter 11 - Future work, lists what has yet to be done, and gives recommendations and ideas to how it should be done.

Chapter 12 - Conclusion.

Appendix A - A description of the contents of the attached CD.

Appendix B - Installation instructions.

Appendix C - Usage. A short users guide to subversion, and its most commonly used commands, and instructions on how to set up a subversion/pfs repository.

Chapter 2

Version control software

2.1 Introduction

Version control software is software designed to track changes in a document, or several documents through the different states of a software development project. The version-control software enables its users to roll-back to any previous states of the projects, in case a engineering dead-end is reached, and also enables its users to get the changes between two revisions.

Version control software is mostly used in software engineering to store source-code, but is sometimes used in other projects to store text and sometimes also binary data. When working in software development it is common, that the developers are working on several versions of the code simultaneously. A team can be assigned to develop new functionality, while another team can be assigned the task of finding and fixing bugs in older versions. The version control system must therefore be able to efficiently work with several *branches* of the same project, and be able to *merge* changes from one branch into another.

The simplest form of version control, which no doubt, has been discovered by all beginning programmers, is to keep separate backups of the source for the different versions of the project, under different names. What the beginning programmer, no doubt also discovers, is that this method is both cumbersome to use, and error-prone. Several systems to automate this task have been created, in this chapter we look at some of them, from the very humble beginnings of SCCS and RCS to modern decentralized systems like darcs and git.

2.1.1 Delta compression

Since changes to a project stored in a version-control system between revisions, are usually quite small, compared to the size of the size of the project itself. Delta-compression is used to only store the change from one revision to another. Files then either start out as empty files, where each change is then subsequently stored as the delta against the previous changes. Or the deltas are done against the next-version of the file. In other words, storing the final revision as a delta against an empty file, then changing the next to final revision to be a delta against the final, and so on. The first method makes storage of new versions efficient, while the second makes retrieval of the latest version efficient.

Traditionally revision control systems have only stored text-data, and deltas have only been used for text-files, since the algorithm for delta compression of binary-files is more complicated and requires more resources. However, some revision control systems employ a binary delta algorithm for binary files.

2.1.2 Storage model

In most software development projects, multiple developers work on the same project. If two developers try to change the same file at the same time, without any mechanism to manage concurrent access to the data, developers can easily end up overwriting each others work. Version control systems solve this problem on one of three different models, file locking, version merging and decentralized version control.

File locking

The simplest method of preventing problems due to concurrent access, is for each developer to simply send a request to a central arbitrator, requesting to lock a file, before changing it. If the arbitrator sees that no other users have locks on the file, the request is granted, and a lock on the file registered on that user. When the developer commits his/hers changes, the revision control system checks whether the developer has the necessary locks to commit. If not the commit is aborted. Or alternatively, if the system uses advisory locks, the user can be warned about possible aftereffects of committing without the necessary locks, but still allowed to commit. Files in the repository are usually readable by all developers[1] , and file-locking usually only grants exclusive access to modify a file.

The argument against using locks, is that it is cumbersome to use, since developers are not always aware of what files they want to change before they start working on some changes. Locks also slow down the development process, since they serialize access to files. In the worst case it can prevent the whole development team of getting work done, if they all have to wait for a file to be released before they can commit their changes.

Version merging

Most version control systems allow multiple developers to edit the same files concurrently. When the developers commit their changes to the central repository, the repository merges changes between the different versions of the same file submitted by the developers. If the files cannot be merged, ie, the same line changes differently in each developers version, then the file has to be merged manually, usually by the developer that checks in the conflicting file last.

We illustrate (in figure 2.1): if three developers *a*,*b*,*c* modify the same file, *F* with revision *R*. Then *a* commits F_a, creating a new file F' with revision *R+1*, then *b* commits F_b. F_b and F' are merged to create F'' with revision *R+2*, although they both are based on *R*. Later *c* commits. The changes made by *c*, F_c are merged with F'' producing F''' with revision *R+3*.

[1] Subversion and possibly other version control systems, can be set up to deny access to some parts of the repository to a remote user, if the user doesn't have the necessary privileges.

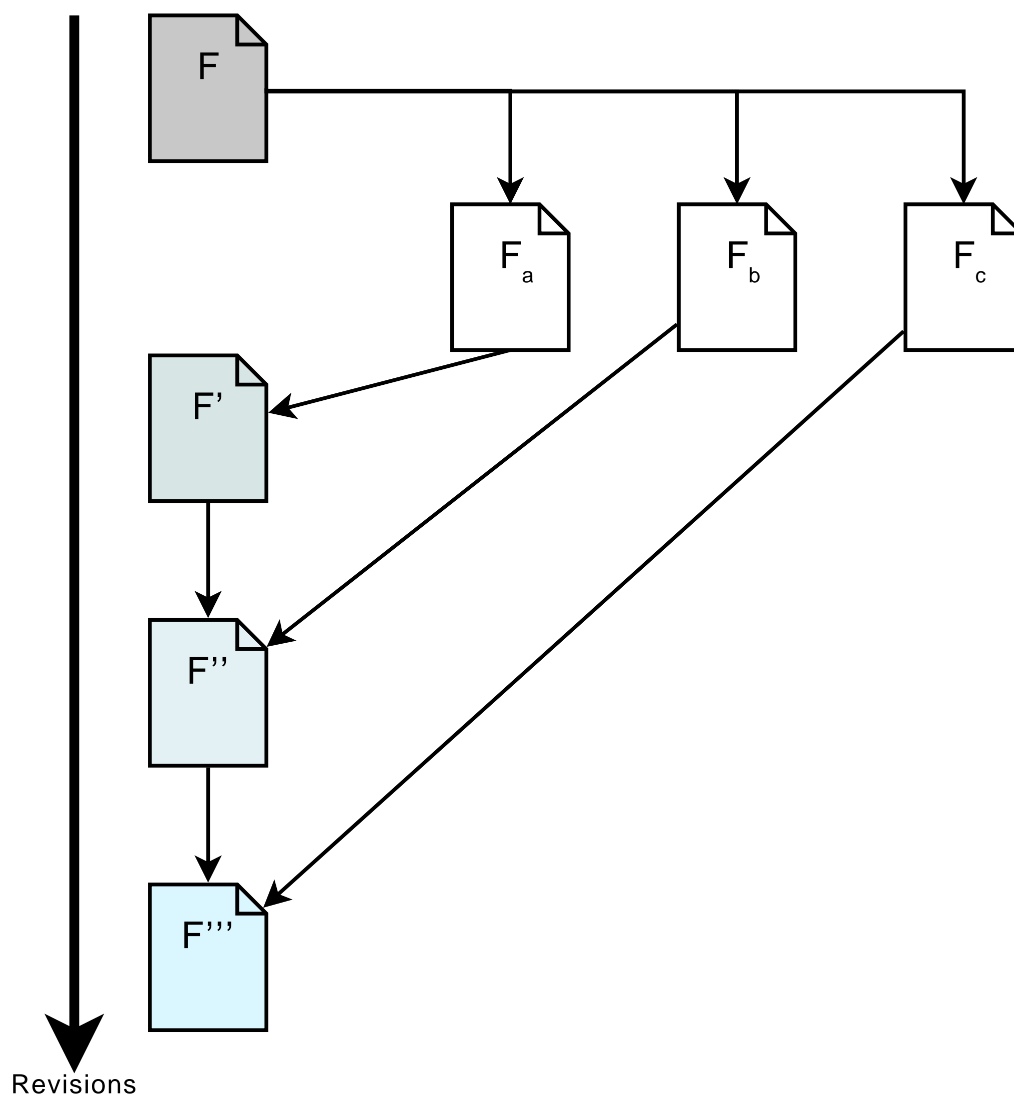


Figure 2.1: The file modifications F_a, F_b and F_c are merged back with the original file.

Most version control systems combine locking with version merging, allowing a file to be locked for exclusive access to one developer, although version-merging is supported by the version control system.

Decentralized revision control

In a decentralized version control system, there is no central repository, instead each user operates his/hers own repository, which is an equal peer with the other repositories. Repositories are organized in peer-to-peer fashion. Since there is no central repository, there is no "official" version of the code. Instead of checking out and committing code to the central repository, users instead pull updates from the other repositories. Usually a great deal of user control is involved in this process, allowing the user to decide which changes to import and which to leave.

Popular implementation of decentralized revision control systems are: BitKeeper, Darcs, Git, Mercurial and Monotone.

2.1.3 Terminology

In the rest of this document we use the following cvs/subversion-centric terminology. The set of related files stored in a version control system is called the *repository*. The files stored by the user are called the *working copy*. To *checkout* a file or a set of files, is to fetch the requested version of the files from the repository, and create a working copy for the files. To *commit* a file or a set of files, is to update the repository so that the newest version of the files stored in the repository corresponds to the working copy. To *update* is to acquire the latest changes from the repository and applying them to the working copy.

2.2 SCCS and RCS

Source Code Control System (SCCS) was the first version control system. Written at Bell Labs in 1972 by Marc J. Rochkind. While today considered obsolete, SCCS' delta storage technique is now considered by many systems to be key to advanced merging techniques.

RCS [9], short for Revision Control System is a program written by Walter F. Tichy in 1985, it is used for revision control of text-files: configuration information, source-code and documentation. It can handle binary data, but the delta algorithm used by RCS, does not do so efficiently.

RCS keeps revisions of individual files and stores the revisions in a RCS-file. The contents of the files are stored using delta compression. The first revision of the file is a delta against an empty file, and each subsequent revision is stored as a delta against the previous. RCS provides functionality for storing a new revision of a file, creating branches, ie several different versions of the file, all based on the same version, reading any version of a file, and merging said branches to form a new, combined version. The RCS files are stored in the working directory with the suffix "*,v*". The commands `ci` and `co` are used to respectively check in and check out revisions. RCS does not employ a client server paradigm, and is only usable by a single user, on

a single computer, in other words, it is a small improvement on keeping backup copies of the revisions of a file.

Because of its limitations, RCS has been replaced by other version control systems.

2.3 CVS

The Concurrent Versions System (CVS)[8], is an open-source revision control system created by Dick Grune. CVS based on RCS, and like RCS keeps track of all changes in a set of files, and uses the same file-format as RCS. Unlike RCS, CVS is designed to allow several users to work with the same repository. A later add-on to CVS allows a CVS repository to be accessed remotely. Each cvs-user has a working copy, where the users changes are stored. The user connects to the repository and fetches the changes between the requested version and the working copy. When the user has finished editing the files, he/she commits the changes back to the repository, which makes them accessible to other users. Clients are usually connected to the server over a LAN network or over the internet, but a "local" cvs-server, running on the same machine as the client is often used as a "personal" single-user repository.

Several clients can access the repository simultaneously, and CVS allows several developers, potentially widely separated, to collaborate on a project. To prevent users from canceling each others changes, CVS does only permit a user to commit a file, if the file in the users working copy is based on the most recent revision of that file in the repository. If that is not the case, the user is prompted to update the file from the repository. Any differences between the users file, and the one stored in the repository are attempted to be automatically merged into the users file. If a conflict is detected, ie the same area is changed in both the repository version and the users copy, the user has to resolve the conflict manually.

When a file is committed, the version number of the file is incremented, and the authors name is added to the log file, usually followed by a log message, describing the changes.

Individual branches can be created, allowing several developers to work on their own branches, committing changes to their individual branches, and later merging the branches back into a single version. Or the branches can be used to separate maintenance projects from development of new features, by having a "development" and "stable" branch. "Tags" can also be used, to tag the files at a particular time, to be able to easily refer to the state of the repository at a particular time.

CVS only revisions individual files. This makes a commit non-atomic, since some of the files form the users working copy can be committed, while others are not committed if the commit should be aborted or fails for some reason. File renames are also not versioned. If a file is renamed at a point in the software development cycle, all history information regarding the file is lost. Files can however be renamed by editing the cvs-repository manually, but it is more of a hack, and is impractical to use. Symbolic links are not versioned in cvs. Which means that any project that uses symbolic links, has to use scripts that can recreate the symbolic links, and instead version the scripts.

Despite its weaknesses, CVS is the most popular open source version control system in use today; having found use in academia, most open source projects and many commercial software firms.

2.4 Subversion

The subversion [3] team at CollabNet set out with the goal of creating a modern replacement for CVS. Subversion is almost command compatible with CVS, meaning that the most used commands of CVS have counterparts in subversion, which can be used by substituting *"svn"* for *"cvs"*. Utilities exist which can transform an existing CVS-repository into a subversion-repository, preserving the change history of the cvs-repository.

Despite this, subversion is quite different from CVS "under the hood". Instead of revisioning individual files, subversion revisions the whole repository. When a file is changed, no matter how small the change is, a new snapshot of the repository is created. This means that the repository is always in consistent state. Ie, if developers always build the project, before committing, any revision in the repository is guaranteed to compile. Unlike CVS, when the repository can be in an inconsistent state, having some files from one users working copy, and some files from another users. To store the snapshots efficiently, subversion employs copy-on-write semantics, meaning that files are only stored when they are changed. Delta compression is also used to store the changes more efficiently.

Like CVS subversion also uses a client-server architecture, and allows several users to collaborate on the same project. Conflicts are attempted automatically resolved during a commit, if they cannot be resolved, the user is prompted to resolve the conflict manually. Since subversion revisions the complete repository, it also detects conflicts in the namespace, if a user for instance deletes a directory, which another user edits a file in, subversion will detect the conflict, and prompt the user to fix it. In addition to merging, subversion supports locking, which allows a user to have exclusive write access to a set of files. Read access to the locked files are allowed. File-lock in subversion are advisory, they can be overridden by a user when committing. But give the user a warning that somebody is working on a file/directory, and does not want the file/directory to be overwritten.

Branches and tags are not directly supported by subversion, instead subversions copy-on-write semantics are used to create cheap copies (in the resource saving meaning) of the project in a *"tags"* or *"branches"* directory. This means however that users have to be disciplined not to write to an existing tag, since subversion has no mechanism for distinguishing a tag from a normal directory copy.

Unlike CVS, commits in subversion are atomic. Subversion starts a new transaction each time a commit is started, changes are applied to the transaction, before exclusive access to the repository is granted, and the transaction either is added to the repository or aborted. Depending on which storage backend is used for the repository, several techniques are used to avoid a situation where a system crash leaves a transaction partially committed.

Subversion also supports symbolic links, on platforms that support those, and creates copies of the affected files, on platforms that do not.

2.5 Darcs

CVS and subversion are centralized version control systems. Systems like this are fine for development projects that have a central authority, like an organization. However in projects that

are developed across organizational borders, or by a loosely knit group of people, like many open-source projects, a centralized approach is not as well suited. Instead a decentralized version control system is better to use.

Darcs[10] is a decentralized version control system created by David Roundy. In contrast to subversion and CVS, it does not have an official repository, or an official branch. Instead each users working copy acts as a repository on its own, which allows several different versions to be maintained at different locations. Patches are global to all repositories, and are exchanged and ordered based on a "theory of patches". To use darcs, a user first gets a copy of any repository, then does changes to the data, pulls changes from other repositories, which is analogous to update in a centralized system, and finally pushes the changes to the other repositories, which is analogous to a commit. Darcs commands are interactive, allowing the user to select which patches that should be incorporated in the users working copy.

2.6 GIT

GIT [7] is another open-source decentralized version control system, written by Linus Torvalds to mange the source tree for the Linux kernel. GIT is written with the specific goal of being useful for a loosely knit team, working on an open-source project. It is designed to be fast, and to be able to merge branches easily [2].

In GIT each developers working copy is, like in darcs, a private repository on its own. Creating and merging branches in git is made easy, by tracking the complete merge history of a file, which is done by tracking all parents of a file, and not only the previous version. Branches in GIT, are only visible to the developers committing to the branch, except for the *master* branch. Therefore it is usual for each developer to have their own branch, or even several branches, which can be used to test out some feature, or to work on some experimental fetures. The GIT development process lets developers commit their changes to their own branch, then the developer notifies potential interested parties of the update. Other developers can then integrate the changes into their own branch as they see fit. Typically the project is organized in a hiearchical fashion, topped by the *master branch*, which is the "official" version of the project. An integrator controls the master branch, and receives patches from only a few trusted "lieutenants", and integrates the changes into the master branch.

[2] A talk held by Linus Torvalds, the creator of git (amongst other things) about git, can be watched at http://youtube.com/watch?v=4XpnKHJAok8

Chapter 3

Pesto

The pesto storage system is a distributed, secure, p2p storage system, that supports disconnected operations [1]. Pesto was developed by F.W Dillema at the University of Tromsø for his Ph.d dissertation. Pesto is designed to allow users to share data with other users, *and themselves* in a distributed environment, where the level of trust between users, and connectivity may vary. Each user of pesto owns one or more pesto-nodes, and is in position to negotiate an agreement with other users of pesto. Depending on the level of trust between the users, they may agree on performing different tasks for each other. A user could for instance have an agreement with another user that he could store his encrypted data on her node. Another user that is more trusted, could be allowed to access the plaintext data, or even be allowed to create new versions of the data.

The nodes communicate with each other in a peer to peer fashion over an asynchronous request-response protocol. Nodes can be part of an ad-hoc network, or even exchange data over other channels than the p2p network.

Symmetric-key cryptography is used by pesto to encrypt its data. Access to data stored in pesto is therefore reduced to access to the encryption keys to the files, which in turn is controlled by encrypting the keys with asymmetrical cryptography[4]. Access to storage space then means fetching and storing encrypted data at a node. While access to content means access to the encryption key, which is needed to decrypt the data. Pesto allows nodes to share storage contents independently from actual content.

Pesto is currently implemented as a filesystem for NETBSD, pfs [1]. And libpesto, a portable library written in C.

3.1 Usage scenarios for pesto

Pesto's goals is to design a storage system that can be useful for several different applications. Here are a few possible applications for which pesto can be useful.

[1]Which should not be confused with pfs, the libpesto based filesystem-backend for subversion, that we implement for this thesis.

Content distribution network
Pesto could be used as the backend of a content distribution network. It provides functionality for pushing updates to mirror-nodes, as soon as the content is available.

DoS-resistant WWW server
Pesto could be used to implement a DoS-resistant web-server. When the server becomes too loaded, because of a DoS-attack, or simply overloaded because of an overwhelming number of valid requests, aka the slashdot effect, it could select other nodes in the pesto-network to use as mirrors. These nodes could be possibly previously untrusted nodes, but due to the severe condition, it might be a lesser risk to use these nodes as backup, than to risk stopping operations. A pesto-node can quickly grant the new server read access to its data, given that it has replicated its data to the new server, by giving it the decryption key for the data.

Mobile office
Pesto can enable a user to truly use a mobile office. The idea is to let the user roam freely between different working environments, while continuing to work on the same data. This means that the user should be able to go from machine to machine, not only on his workplace, but also to use his home-computer to access the data. In other words, this is the traditional file-sharing system with a twist, instead of sharing data between several users, the user instead shares data with himself on different locations, without fetching the data from the same server. Since data is stored locally, a user can edit the data, even when disconnected. This however means that the out-of-date data can be edited. The user is in principle responsible for resolving these conflicts, but an application-level consistency control can be implemented on top of pesto.

Clinical information system
Pesto can be used to implement a clinical information system. Since it is secure, and allows ad-hoc, quick sharing of data, when needed.

3.2 Architecture

Pesto itself is divided into two layers, the storage layer, and the distribution layer. The storage layer is responsible for storing file-updates, policies, and messages to local storage, while the distribution layer is responsible for communication with other nodes, and distribution of data to other nodes.

3.2.1 Files

The pesto storage system provides distributed storage of files to its users. A file in pesto starts its life as an empty file, when the contents of the file are updated, the update is added to pesto, but it is stored as a separate entity. Each file-update is identified by its GUID, a 128bit random number. Each update also stores the GUID of the update it was based on, its *parent update*. Updates can thus be organized in a tree, called the *version tree* or *vtree* for short. Several nodes in the vtree can have the same parent. This could happen if two nodes try to create an update of a file without knowing that a more up-to-date version of the file exists. pesto itself does

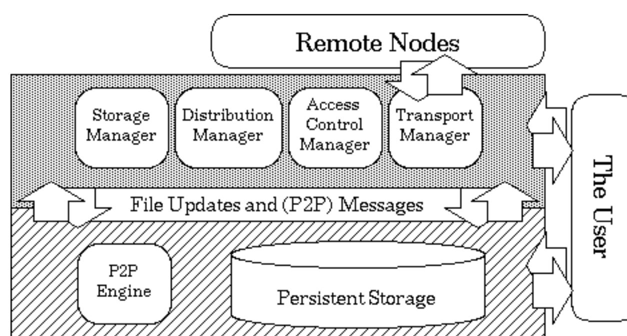


Figure 3.1: Conceptual view of a pesto node.

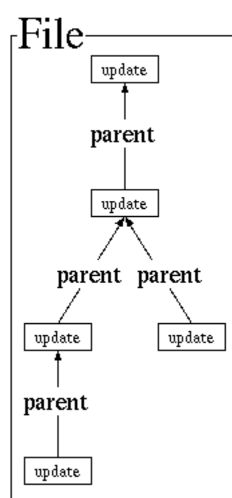


Figure 3.2: The update tree for a file

not have any consistency-control mechanisms. Instead pesto exposes the branches created by concurrent updates to the application, leaving the task of consistency control to the application. The semantics of a file update is also left to the application. It could either store deltas between each version in updates, or it could store the complete file-version in each update.

The storage model for file updates is Write-Once Read-Many-Times (WORM). Files are stored as a series of updates, and pesto provides no functionality to delete an update once it has been created. The update can be in two different states, *created* and *stored*. When an update is created, its metadata is stored and distributed. When the update is stored, its data is stored and distributed. Once an update has been created, it can only be stored. Once it has been stored it cannot be removed. But a stored update can be freed from a nodes local storage. Freeing an update, does not, however forcibly remove the update from pesto-nodes that have already received the update, but the node is then free to delete the update.

To create a file, pesto first creates and stores the first, empty update, of the file. After the update has been stored it cannot be removed. Meaning that when a file first has been created, it cannot be removed, only updated.

Nodes themselves are also represented by a file. This node contains administrative information for the node, the GUID of this file identifies the node, and is called the *node-id*.

Each time an update is created, the node-id of the node that created it, is called of the node it was created on, the *creator id*, is associated with it. The creator of the root update for the file, is special, since it is the *owner* of the file.

3.2.2 Policies

Each file has a *replication policy* and an *access control policy* associated with it. These policies are themselves also stored as files, and identified by their GUIDs. Policies are specified when creating the file, and the policy governing a file cannot later be changed.

The replication policy specifies the set of nodes where the user wants to store the data of the file, and also specifies the set of nodes responsible for distributing the replicas to these nodes. Replicas are distributed according to the replication policy, anytime when a node can establish a connection with any node that should store the replica.

The access control policy specifies who is allowed access to the contents of a file. I.e the nodes that should have read access to a files updates. The access policy is implemented by encrypting the file-updates. Each update is encrypted with its own *member key*. Read-access for a file-update is granted by giving out the member-key for that update. All member-keys are encrypted with a file-key, which is generated when the file is created. Access to all current and future updates is granted by giving out the file-key. Update access to a file is guarded by the same mechanism as read access. When a request to update a file is received in the form of a new file update, it is considered authorized if the update is encrypted with the fresh member key, and that key is found encrypted with the file key for that file. A key is considered "fresh" if there is no local file update encrypted with that key. To grant somebody update rights, a user that knows the file-key, has to generate a new member key and encrypt it with the file-key. Nodes encrypt updates with the received member-key. Any node that knows the file-key can check the encryption of an update and verify the freshness of the member-key. This means that any node that knows the file-key can grant another node a one time update right, without giving read access to previous or future versions of the file.

An update authorization that has been granted to a node, can later be revoked. This might be needed if a node hoards update authorizations, without using them. To revoke the authorization the owner simply creates an empty file-update with the member key used for the grant. If a new update is created with the same member-key, the member-key is no longer fresh, and the update is not accepted. There is no mechanism to revoke read access to a update. The closest thing to doing it is to copy the file data to a new file, and stop replicating the old.

3.2.3 Trusted bases

Pesto separates the different mechanisms for distribution, storage and access control. The nodes responsible for handling the different mechanisms can be viewed as a collection of bases.

The *trusted storage base* (TSB) implements the storage policy for a file. The nodes in the TSB are responsible for keeping the file stored and available. This is done by having the node store a copy of the (encrypted) file on local storage.

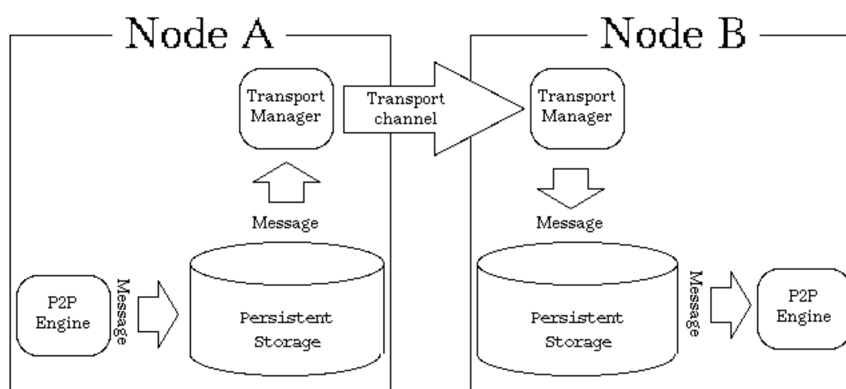


Figure 3.3: The pesto-node 'A' sends a message to 'B'

The *trusted access base* (TAB) implements the access control policy for a file. The nodes in the TAB hand out encryption keys that allow reading the contents of a file, and hand out authorizations for file updates. Multiple TABs can be defined for a file, one for each type of access.

3.2.4 p2p

Pesto is loosely organized as a p2p network. Nodes are not part of pesto because they participate in a global p2p network. They are pesto nodes simply because they can act as pesto nodes and understand the p2p messages. The p2p protocol is fairly simple. There are in total four commands, and the protocol is divided into two parts. The first part has the two commands, *store* and *fetch*. The store and fetch messages are one to many, and symmetrical. I.e a node can send a store or fetch request to many nodes, and will always get a store response to a fetch request, and a fetch response to a store request. Both store and fetch commands take a list of GUIDs as its arguments, which is the identifiers of the files to be stored or fetched. A fetch-request message informs its recipients that the sender wants to store the file-updates specified by its argument. A store-request message informs its recipients that the file updates identified by the GUIDs are available for download. By responding to a fetch-request, with a store-response, the sender acknowledges that it has the file identified by the GUID, and can send it to the requester. By responding to a store-request with a fetch-response, the sender says that it is ready to accept the file-update.

The second part of the protocol consists of the *read* and *update* commands. The read-command has a list of GUIDs as its argument, and requests the encryption keys for its argument file-updates. The update command requests a *fresh* encryption key to be used for making a new valid authorized child update of the GUID in its argument.

The p2p system is implemented by the *transport manager*, which is a separate process. The transport manager looks in the pesto directory for new file-updates and sends it to its destinations as given by the files storage policy.

Chapter 4

Subversion

Subversion tries to be "command-compatible" with CVS, and appears to have a lot in common with CVS. But under the hood subversion is quite a different beast. CVS keeps revisions of files, and tracks changes to these files. Subversion on the other hand, keeps revisions of the whole repository.

Users of subversion have each their own *working copy*, which is stored on the users machine. The working-copy is a mirror of the repository. The user edits the files in the working copy, and does a *commit*, when finished. Subversion then initiates a *transaction*, and a new revision of the repository is then created in which the files are either updated to the contents of the users working copy, or no revision is created at all. A working copy that contains only the files and sub-directories of some directory in the repository can also be created. This working copy is handled like any other working copy, and under a commit, the files not in the working copy, are simply assumed to not have been changed.

Subversion itself is implemented as a collection of shared libraries. The subversion client, server and management interface link with these libraries, but it is simple to create other programs that can be used instead of the official subversion programs.

The subversion server is also special, since there is no single master server-process. Instead multiple server-processes, usually one per transaction are started. There is no inter-process communication, and the server-processes all work on the same files.

4.1 Features

Subversion has some features that differentiate it from CVS, and make it a better versioning system:

Revisioned metadata

Subversion does not only version changes done to files, but the whole working copy. This lets the user easily roll-back to a consistent view of the repository, with all files in the state they had at a certain revision. It also makes it possible to track subtle, non-data changes to the repository.

Copying or moving a file with a subversion-client records the move in the repository, which means that the history of the file is saved. Subversion also supports unix symbolic-links, and recreates these on platforms that support symbolic links.

Cheap branching/tagging

Copying in subversion is cheap, the file/directory that is copied, is not really copied in the repository. Instead a new entry for the file/directory is created in the parent directory. Both the new and the old entries point to the same file, much like unix hard-links are implemented. But in contrast to hard-links, when either the original or the copy is changed, a copy of the file/directory is created in the repository, giving the subversion file-system copy-on-write semantics. Copying a filesystem tree is similarly effective. When changes are made, only the parent-directories of the changed file are copied. All other directory-entries are left still pointing to the original files.

This makes subversion-copy cheap, O(1) to be precise, since copying a file/directory does not depend on the file size, or size of the directory sub-tree, the copy is done in constant time, and constant space (the space of an extra directory entry).

Tags and branches are therefore implemented as copies in subversion. A subversion-repository customary has a "*tags*" and a "*branches*" directory, where, respectively, "tags" and "branches" are kept. To create a "tag" the root directory for the project is simply copied to the *tags*-directory, and to create a branch, the project directory is copied to the *branches*-directory.

Atomic commits

Commits in subversion are atomic. The repository is either updated to the same state as the working copy, or it is not touched at all. Subversion uses transactions to guarantee atomic commits. Each time a user starts a commit a new transaction is started. All changes done to the working copy are applied to the transaction. If no errors occur, and the transaction can be committed, the transaction is added as the next revision of the repository. If an error occurs, the transaction is discarded.

Properties

A feature of the subversion filesystem is its support for properties. A file/directory in subversion can have properties associated with it. Properties are simple name, value pairs of text, which subversion versions like other file-data. Subversion itself uses properties internally, these properties are prefixed with "svn", and contain information on the type of file, whether the file is executable, etc. Users can set and edit their own custom properties via the svn-client interface.

File-sharing

Suppose that two users, A and B are working on the same file, "*foo.c*". A finishes first her work firsts, and commits her changes. B finishes his work after A and tries to commit. B's copy of the edited file are then outdated, and the changes cannot be committed.

If a situation like this occurs, subversion will abort B's commit. B can then try to update his working copy to get the newest version. Since B's "*foo.c*" has been modified from the initial version, subversion cannot update B's "*foo.c*" to A's version, but B, still needs to see A's changes. Subversion then tries to merge A and B's changes into one file, and adds A's version, B's version and the merged version to B's working copy. B will then have to manually resolve this conflict. Subversion is more tolerant of namespace conflicts. If A creates the file "*/foo/bar*" and B creates "*/foo/baz*, subversion will automatically produce a revision in which both "*/foo/bar*" and "*/foo/baz*" exits.

Subversion also has locks. Locks allow a user to reserve the right to edit a file, or all subfiles/directories in a directory. Locks are held until they are freed by the user, or broken by the administrator. However, locks are not absolute. A user can still commit changes to a locked file, but the commit has to be forced, by giving the subversion client the `''--force''` switch.

4.2 Layers

Subversion uses a layered design, where each layer interfaces only with the layers directly above and below it. The layers are implemented as shared libraries, and any can be replaced with other modules that implement the same functionality.

The uppermost layer, is the *client-layer*. It consist of the *libsvn_client* and *libsvn_wc* libraries. The client library is responsible for communicating with the server, and the wc-library is responsible for working with the working-copy.

Under the client-layer, lies the *repository-access layer* or *WebDAV* layer, depending on how the repository is accessed. The WebDav is used by an apache-module, which allows the apache2 web-server to act as a subversion server, and also allows browsing of the repository as a ftp server/dav server. The standard repository-access layer (RA) is used to access the repository through the svnserve-server, or locally, through the *file* protocol, if the repository is stored locally.

Beneath the RA-layer lies the *repository-layer*. It is made up of the *libsvn_repos* and *libsvn_fs* libraries. The repos-library is responsible for firing of hooks-scripts, custom scripts which users can add to the repository, and which should be executed after certain subversion operations. And to access the repository. The fs-library is an abstraction layer above the different filesystem-backends for subversion. Together they implement the filesystem-interface of subversion.

The filesystem-backend provides a versioned filesystem. Instead of accessing files with a path, files are accessed with a {path,revision number} tuple, and returns the contents of the file, how it was in that particular revision.

There are two official filesystem-backends used by subversion now. The original base filesystem-backend which is implemented as a berkley-db [13] database, bdb and the file-based backend, fsfs. In addition to that there is a non-official sql filesystem-backend, which is based on the bdb-backend, but uses an sql-database. It is possible to chose which filesystem is used when a repository is created by giving `svnadamin` the `--fs-type` switch.

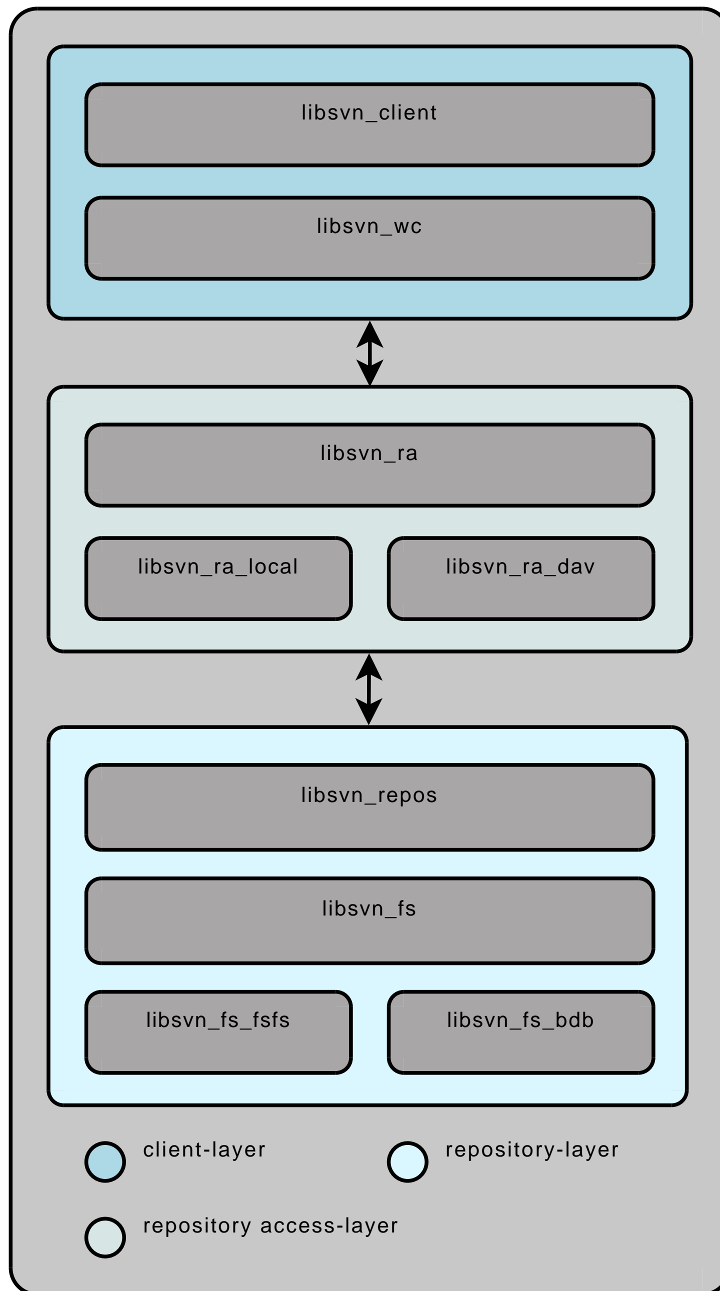


Figure 4.1: Subversion layers

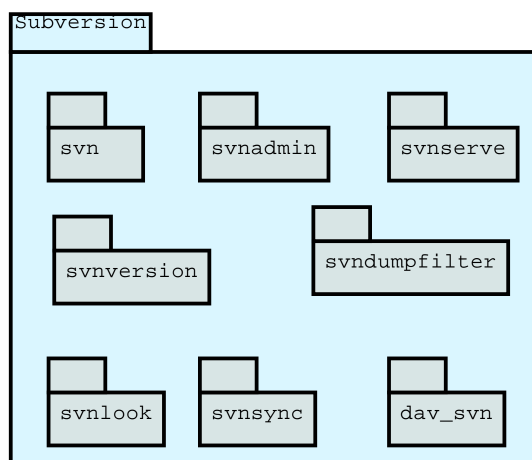


Figure 4.2: The component parts of subversion

bdb

The bdb filesystem backend uses a berkely-db database as its storage. Directories and files are stored as table-entries, which have property and data representations. The representations have a many-to-one relationship with the files/directories table, so a file/dir can have many representations, one for each time the file/directory was modified.

When a transaction is committed, bdb stores the last version of the file-data as a plaintext, and updates the previous version so that it is a delta against the newest version. This makes the common-case of HEAD-checkouts faster than checkouts of previous versions.

fsfs

The fsfs filesystem is a more recent reimplementation of the filesystem backend for subversion. We will describe fsfs in more detail later, but for now it is sufficient to say that fsfs uses plaintext-files for storage. Each revision is stored in a file, which contains the **changes** for that revision. In fsfs each file starts as an initially empty file. When the file is changed the delta from the previous version to the next is to be reconstructed from its previous versions; and revision-files are never changed once they have been created.

4.3 Modules

Subversion is implemented as a server and a client module, in addition to several management and administration modules. The standard client shipped with subversion is the command-line only client `svn`. Several other clients exits, most of which either integrate with the file-manager of the operating system, or are a part of an integrated development enviroment.

Clients all link with the subversion client-library, *libsvn_client*. The library implements access to the repository over various protocols. The repository can be accessed locally, over the *file-protocol*, or it can be accessed over the *svn-protocol*, or it can be accessed over the *http-protocol*.

When a client accesses the repository over the file-protocol, it simply manipulates the files in the repository directly. When it uses the http-protocol, a apache-module acts like a subversion-sever, and communicates with the client over the WebDAV protocol.

When the client uses the svn-protocol, a special server-program, `svnserve`, is started on the machine hosting the repository. `svnserve` can be started either as a daemon that listens for new connections, or a new process can be spawned for each new connection by the *inetd-daemon* `svnserve` speaks with the client and manipulates the repository-files on behalf of the client. A special-case is when the *svn+ssh* protocol is used. *ssh* is used to tunnel the communication between the client and `svnserve`, and a svnserve-process is started automatically on the host, with the permissions of the user. To use svn+ssh the user needs to have a login on the server.

While the different protocols that subversion uses is of little consequence to this project, we should note a few things. One, the repository has to be writable by all local-users that need to access it. It should also be writable by the inetd-user, and the apache user. Two, multiple, heterogeneous processes can access the repository concurrently. The filesystem-backend has to be designed in a way that permits this, ie files that are written to, should be protected by file-locks, and commits to the repository have to be serialized by some means.

Chapter 5

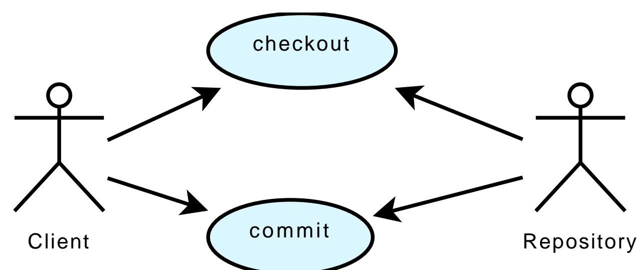
Requirements

5.1 Usage scenarios

The goal of this project is to modify subversion, so that it uses pesto for storage, instead of storing data to local repositories. Before identifying the requirements for the system, we shall look at some possible usage-scenarios for a *pestified* subversion.

5.1.1 Subversion server

This usage-scenario is quite straightforward. Users use a pesto-enabled subversion repository like they would any other repository, perhaps not even knowing or caring abouth how subversion stores their data. The only major difference from normal subversion operation, is that the data is encrypted, before being written to local storage.

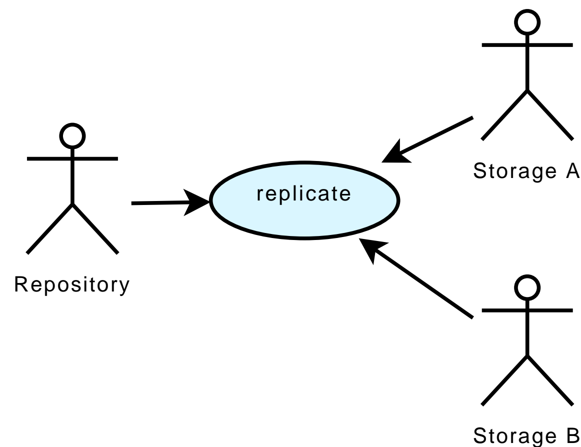


Use-case:	Standard subversion repository
Actors: Description:	Repository, client. The client interacts with the subversion-repository. (Either checking out files, or committing changes.)

5.1.2 Replication

In this usage-scenario, a pesto-enabled subversion repository sends its data to other pesto-nodes that are trusted by it to store its encrypted data. The repository could be running on a computer with limited storage resources, for instance an laptop. The storage nodes could be

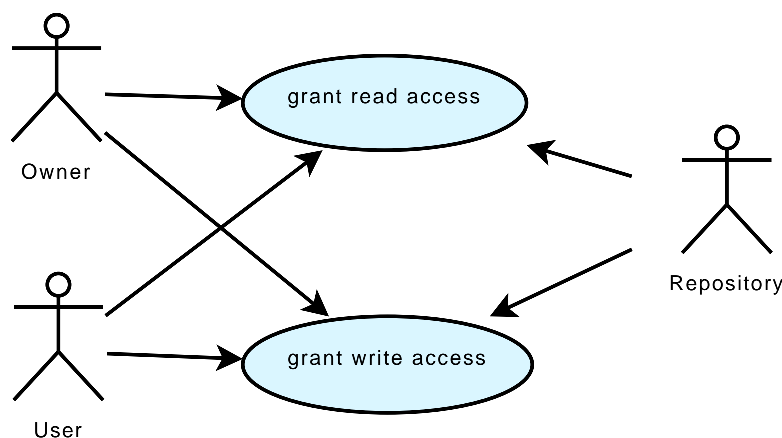
servers with lot of storage space. The repository could even be programmed so that it removes non-recent data from its disk, so that only the most recent changes are kept locally, and the earlier changes are kept on the storage servers.



Use-case:	Replication of data
Actors:	Repository, storage node A, storage node B.
Description:	The repository replicates its data to the storage nodes A and B.

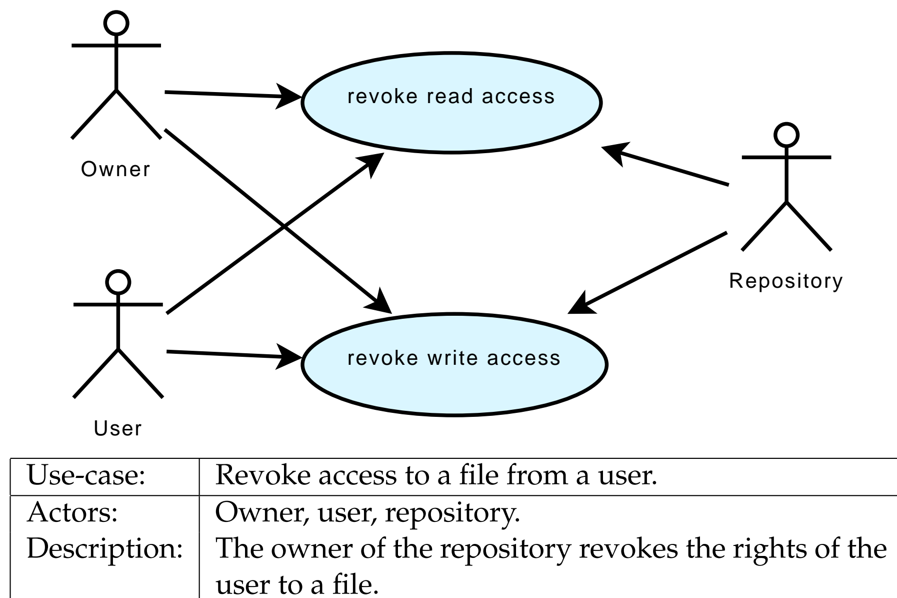
5.1.3 Collaboration

In this usage-scenario, the owner of a pesto-node, that runs a subversion repository, gives another user (a pesto-node) rights to read a version/create a new version of a file stored in the repository.



Use-case:	Grant access to a file to another user.
Actors:	Owner, user, repository.
Description:	The owner of the repository grants the user some rights to a file.

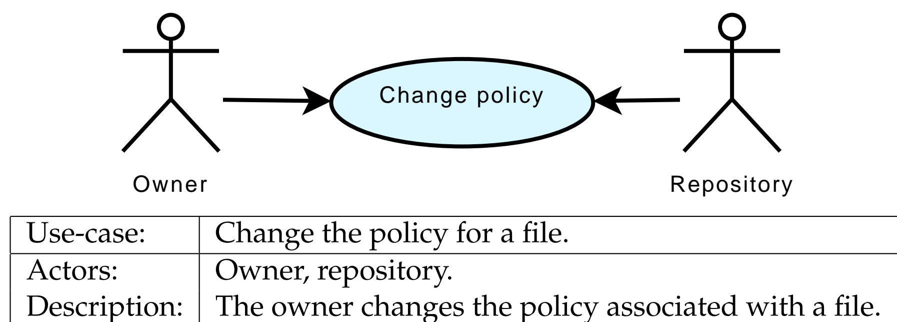
Similarly, a user can revoke the rights to modify a file from a user.



Use-case:	Revoke access to a file from a user.
Actors:	Owner, user, repository.
Description:	The owner of the repository revokes the rights of the user to a file.

5.1.4 Policies

In this usage-scenario the owner of the pesto-node hosting the subversion repository changes the policies under which the files are stored.



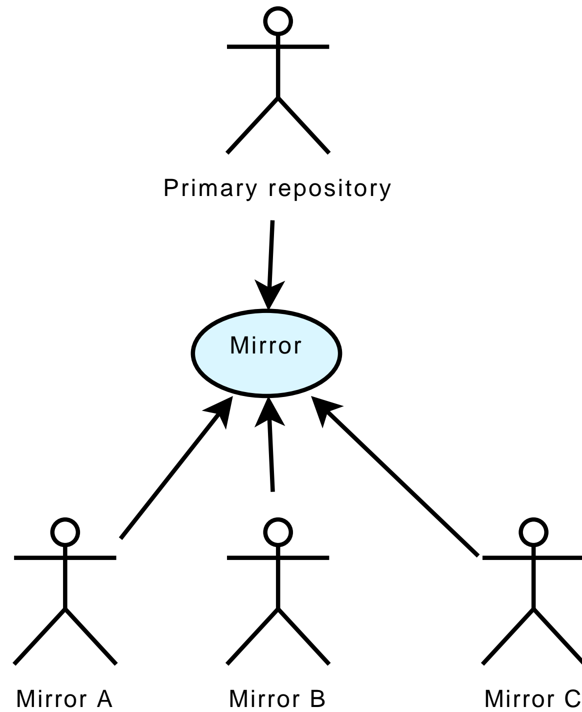
Use-case:	Change the policy for a file.
Actors:	Owner, repository.
Description:	The owner changes the policy associated with a file.

5.1.5 Mirroring

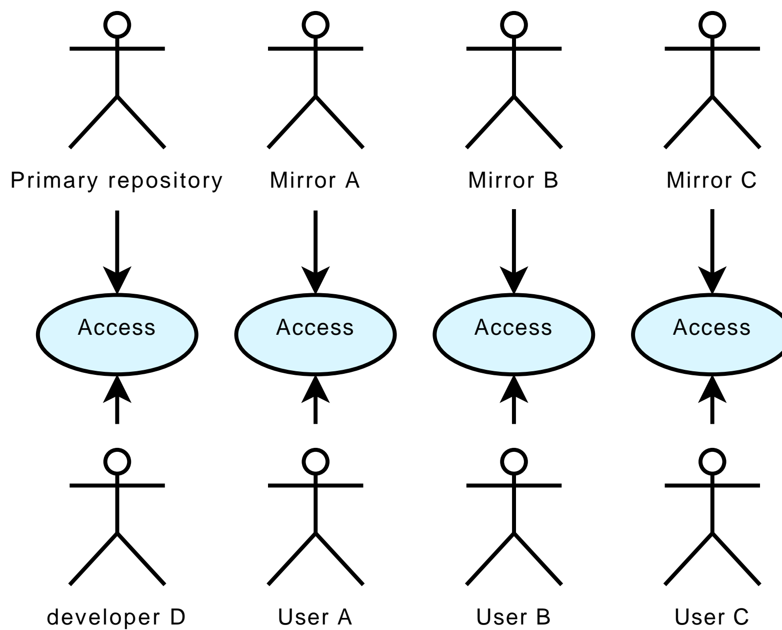
Several subversion-repositories can be created that share the same data. These repositories can be used as mirrors to a single repository. This can be useful for a project with a relatively small number of committers, and a large number of users that access the projects data, so that the central repository to which developers commit is offloaded, while others can download the sources from the mirror repositories.

This is common for a lot of opensource projects where the primary mode of getting the source code for the project is checking it out of source control. Still, there is a small controlled group that commits to the repository.

Note that if the mirror repository has not been granted rights to create new versions of the files in the repository, it cannot commit to the repository.



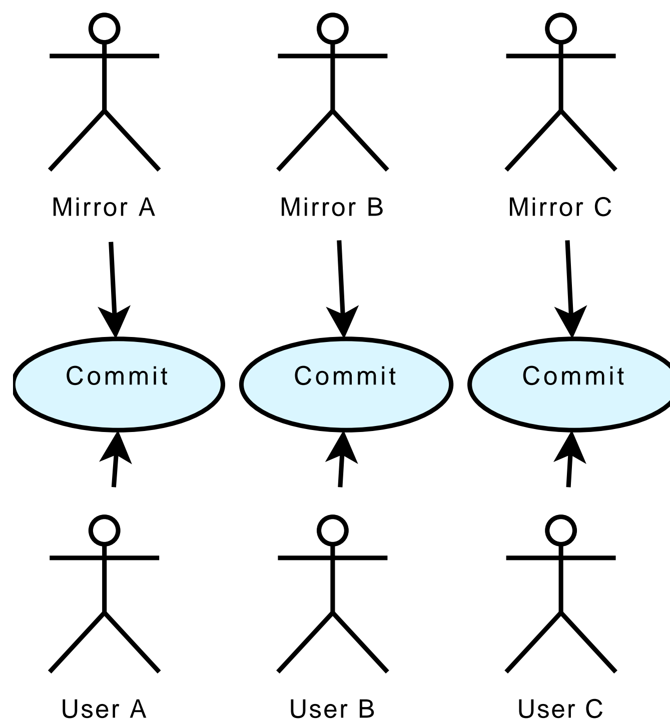
Use-case:	Mirroring of a repository.
Actors: Description:	Primary repository, mirror repositories A, B and C. The central repository offloads its data to the mirror-repositories.



Use-case:	Mirroring of a repository.
Actors:	Primary repository, mirror repositories A, B and C, users a, b and c, developer d.
Description:	The users access the repository closest to them.

5.1.6 Decentralized source control

This usage scenario is somewhat similar to the mirroring scenario, but here the mirror-nodes have been given write access to the files. This means that commits can be done from any of the nodes. This creates a decentralized source control system.



Use-case:	Decentralized source control.
Actors:	Repositories A, B and C, users a, b and c.
Description:	The users commit to the closest repository.

5.1.7 Concurrency control

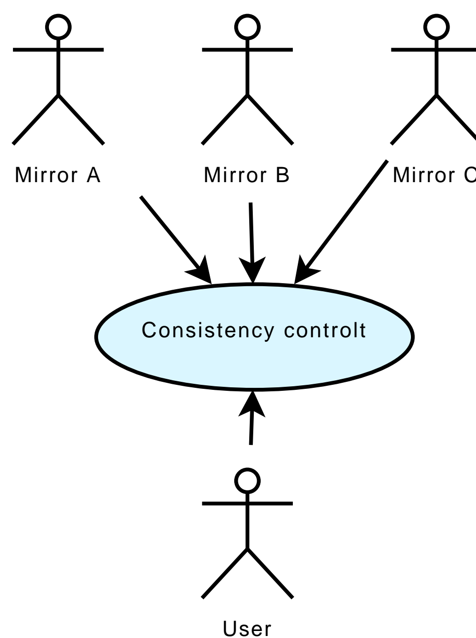
In a system that supports concurrent access to its replicated data, several nodes, some form of consistency control has to be used. There are basically two forms of consistency control, optimistic and pessimistic, and one always has to chose between degrees of availability, and data consistency [6].

In a distributed system there is bound to be some delay from when a user commits the changes at one repository, and the other repository sees the changes. With a pessimistic approach, a commit to any repository would have to be delayed until either all repositories are up, or at

least until a central, master repository is up and running. This is not desirable for this system, since a user then would not be able to use the system while offline.

If an optimistic consistency control approach is taken, users are allowed to commit at any time, and the conflict is resolved at a later time. An optimistic concurrency control system usually carries a log of the changes done at each node, and tries to reorder the changes so that they can be serialized, undoing any changes that are in conflict [6]. A version-control system is different from normal consistency-control scenarios, in that it allows the user to correct inconsistencies manually should they occur, which is typically referred to as merging in a version-control system. The difference between typical merging and optimistic concurrency control, is that the merging is done before or during a commit, and optimistic consistency control has to be done after a commit.

Consistency control should therefore be done automatically if possible, should there be a conflict that cannot be resolved without user intervention, then the user has to be prompted to resolve the conflict manually during a check-out of the inconsistent revision.



Use-case:	Decentralized source control.
Actors:	Repositories A, B and C.
Description:	The repositories do optimistic consistency control, with the help of the user.

5.2 Functional requirements

The *pestified* repositories should be accessible by standard subversion clients. To do this, the pesto-enabled subversion has to implement the core functionality of subversion. Here we identify and describe the core-functions of subversion.

5.2.1 commit

Use-case:	Commit
Actors: Description:	Repository, user, working copy. The users commits the changes in his/hers working copy to the repository.

5.2.2 checkout

Use-case:	Checkout
Actors: Description:	Repository, user, working copy. The users checks out a revision from the repository, and makes it his/hers working copy.

5.2.3 update

Use-case:	Update
Actors: Description:	Repository, user, working copy. The users updates hers/his working copy to the contents of a revision in the repository.

5.2.4 propset

Use-case:	Propset
Actors: Description:	user, file, working copy. The users sets the properties of a file in the working copy.

The property is set in the repository after the working copy is committed.

5.2.5 propget

Use-case:	Propget
Actors: Description:	user, file, working copy. The users gets a property for a file in the working copy.

Use-case:	Propget
Actors: Description:	user, file, repository. The users gets a property for a file in the repository.

5.2.6 add

Use-case:	Add
Actors: Description:	user, file, working copy. The users adds a file to the working copy.

The file is permanently added after the working copy.

5.2.7 delete

Use-case:	Delete
Actors: Description:	user, file, working copy. The users deletes a file from the working copy.

The file is removed from all future revisions, once the working copy is committed.

5.2.8 move

Use-case:	Move
Actors: Description:	user, file, working copy. The users moves/renames a file in the working copy.

Subversion implements this as a delete followed by an add. Changes are made permanent after the working copy is committed.

5.2.9 copy

Use-case:	Move
Actors: Description:	user, file, working copy. The users copies the file in the working copy. The history of the file is preserved.

The changes are made permanent, once the working copy is committed.

5.2.10 lock

Use-case:	Lock
Actors: Description:	user, file, repository. The users locks the file in the repository, reserving the rights to change the file for him/herself.

The file is locked, but other users can force edits to the file, in other words, the lock is a warning to other users, not a barrier.

5.2.11 diff

Use-case:	Lock
Actors: Description:	user, file, repository. The users gets the difference between two revisions of a file in the repository.

5.2.12 log

Use-case:	log
Actors: Description:	user, repository. The users gets the update history for a file, directory or repository, between two revisions.

5.3 Non-functional requirements

Performance
The system is to be used in a research setting, and performance is therefore secondary. Still it should give acceptable performance. Also, performance should conform with the users regard of the complexity of the operation. Ie, adding small changes to the repository, should take less time than adding huge changes, updating a recent working copy to the newest version should take less time than updating an out of date copy/checking out the complete repository.

Extendibility
The resulting system shall be used for future research, and the code written should be easy to extend in the future.

Security
The resulting system uses pesto which is a secure system, and the security of any component of the system should not be much weaker than any of the security mechanisms in pesto. It should also not leak any more information, than which is necessary to store files in pesto.

5.4 Target platform

Pesto itself is portable and can be used on several unix implementations, and is mainly developed under NETBSD and MaxOS X. It is does also compile under Linux and is ported to Microsoft Windows NT-family[1].

Subversion is ported to many unix-flavors, linux, MacOS X and Microsoft Windows NT.

While this, in theory should mean that the new system should run under all platforms mentioned above, it should be made clear that this is just a research system, and resources for testing on multiple platforms are not-available. The system should strive to be platform-independent, and isolate any code that is platform dependent, but it should as minimum only run on two platforms, NETBSD and linux running on intel 64 and 32bit cpus (x86 and x86_64).

5.5 Alternatives

The easiest way to store subversion data in pesto would be to take the whole subversion repository after each commit, compress it into a new file, and add that file to pesto. This could even be accomplished by using subversion hook-scripts that would hook onto commits and checkouts. While this approach would be simple to implement it would have a few serious drawbacks.

First of all, since pesto doesn't allow a file to be neither modified, nor deleted, after it has been stored, it is impossible to replace the old repository in pesto with the new one. This means that the data for the repository would be duplicated in each revision. This could be alleviated

[1]The original Windows NT, Windows 2000 and Windows XP

by storing deltas/changes between each revision, like it is done in the fsfs backend. But there is still a problem. The requirements specify that it should be possible to take any version of a file in the repository, and grant another pesto-user access to that file-version. This means, in practice, that each file has to be managed by its own policy, which in turn means that every file-version has to be stored as a separate pesto-file. Deltas of individual files, can also not be used. This is because it would be necessary to have access to all previous versions of the file to be able to reconstruct it. Reverse deltas, can also not be used because of pesto's WORM semantics.

To fulfill the requirements, a new filesystem backend for pesto should be created that uses pesto. We started to implement a new backend from scratch. While this approach had the most chance of leading to a clean design, it proved to be quite difficult, and too time-consuming to be implemented. Therefore we abandoned the project, and decided on a new approach. However, this "experiment" did not prove to be completely fruitless; we had learned a valuable lesson, the fsfs filesystem could be, relatively easily, modified to use pesto. It already has immutable revisions, fundamentally, what had to be done, was to split up the revision-files into separate files, and change the way file data was stored. In the next chapter, we will therefore look closer at the implementation and design of the fsfs filesystem, before moving on to discuss how the fsfs filesystem should be integrated with pesto in the design and implementation chapters.

Chapter 6

The fsfs backend

Since the libpesto backend for subversion implemented for this thesis is based on the fsfs backend used by subversion it is necessary for the reader to familiarize him/herself with some of the implementation details of fsfs before moving on to discuss how the pfs-backend is implemented.

6.1 Nodes and node-revisions

fsfs is a revisioned filesystem implemented on top of the operating systems filesystem. By a revisioned filesystem, we mean a file-system in which any file or directory stored can be restored to the state it had in any of its previous versions.

A file/directory in fsfs is stored in a *fsfs-node*, which is analogous to an inode [19] in unix. A node is identified by its node-id. When a node is changed, a new revisions of the node, called a *node-revision* is created. Each node-revision has a *node-id* which identifies it with the node, but has, in addition a *copy-id* which is incremented every time a new node-revision of the node is created.

A node-revision can be in one of two states, mutable or immutable. A node-revisions is mutable when it is created, and becomes immutable as soon as it is committed. Node-revisions can only be created in a transaction [1], and a mutable-node-revision is therefore always part of a transaction. Node-revisions belonging to a transaction are marked with a transaction id, while immutable node-revisions are marked with the revision number of the revision in which they were committed. Note that node-revisions in transactions use internal counters for copy-ids, and two node-revisions in concurrent transactions can have the same combination of node-id,copy-id, and a node is therefore only uniquely identified by the combination of its node-id,copy-id and transaction-id or revision number.

To illustrate, suppose that a user creates a file, *foo.c* and adds it to the repository, with the path */src/foo.c*. A new node for *foo.c* is created, and the first node-revision for the node is created. The parent directory, */src* must then be updated to add *foo.c* to its directory entries. A new node-revision for */src* is created, and *foo.c* is added to its directory entries. Since a new node-revision for */src* is created, / has to be updated to point to the newest node-revision of */src*. A

[1] With the exception of the zeroth revision, in which the root-node is created outside any transactions.

new node-revision for / is therefore created, and the reference to */src* is updated to point to the newly created node-revision for */src*. To minimize the number of node-revisions created, only the first change to a node in a transaction creates a new node-revision, all remaining changes are done to the node-revision which is already in the transaction.

Each node-revision can have data and properties. Properties are meta-information for the file/directory, including but not limited to, permissions, mime-type, last author, and other properties set by the user. The data of a node is either the contents (plaintext or delta) of file, or the directory entries of a directory. Both data and properties are stored in what fsfs refers to as *representations*. Each node-revision keeps a reference to its current data and properties representations, which can be shared amongst several files. The representations are stored in the node-file for node-revisions that are part of a transaction, or they are stored in the revision file with the rest of the revisions data, if they belong to a committed revision. If a property of a mutable node-revision changes, its properties-representation is updated. However, if the properties for that node-revision are shared with a previous node-revision, the old properties-representation is copied to the transaction as a new properties representation, and the property is set. Data-representations are handled analogous to properties-representations, but are stored differently, depending on their type. File data is never copied to the new representation, but always created as an empty delta against the base-revision.

A similar operation to creating a new node-revision is copying a node. fsfs implements copying by having a copyfrom-field in each node-revision. When a node is copied, its copyfrom is set to the node-id and revision of the original node. The data and properties representations of the new node are left pointing to the same as data and properties as the original. Only when the data changes, either for the original or the copy, is a new copy of the data created. Firstly it makes it easy to track the copy-history of a node, to determine if the node was created in a revision, or if it was at some time “forked” off another node. Two, it makes copying a node a cheap operation, both in terms of speed and disk-space used. The last property comes in handy, since subversion implements “tags” and “branches” as copies.

Subversion, in contrast to CVS, does not really use tagging. Instead it is common practice to copy a “tag” to a separate directory. To illustrate, lets look at the fictious project “foo”, then */foo/dev* could be the unstable, development branch of the foo project, */foo/4.0rc1* could be the release candidate for version 4.0, and */foo/3.0* could be the release for the not so popular, but still supported foo version 3.0. Suppose that a file *bar.h* hasn’t been changed since v3.0. Then */foo/dev/bar.h*, */foo/3.0/bar.h* and */foo/4.0rc1/bar.h* would all point to the same node-revision. The file *bar.c* has been changed between 3.0 and rc1, but has not been changed between from then. */foo/dev/bar.c* and */foo/4.0rc1/bar.c* would point to the same node-revision, while */foo/3.0/bar.c* would point to an earlier node-revision of *bar.c*.

6.2 Skip-deltas and data storage

How the data-representation of a node is stored depends on what type of node it is, and what type of data the node has.

Directory entries are always stored in plaintext, that is, the newest/current version is always written completely to the file.

Binary-files files are always stored in plaintext, since a delta of a binary file is impractical and of little value in itself. Text-files are however stored as deltas. fsfs stores the first revision of the file as a delta against an empty file. Each succeeding revision is then stored as a delta against the previous. This creates a lot of overhead when checking out the latest revision, since all previous revisions have to be read. To avoid this subversion uses skip-deltas. A skip-delta is a delta against a revision other than the directly preceding revision. To determine which revision to base the delta on, subversion "prints" the revision number out in binary, using big-endian notation, flips the rightmost '1' bit, and bases the revision on it. Example, 54 (110110) becomes 52(110100), which means that we base the delta for the 54th revision on the plaintext of revision 52. The 52nd revision is based on the 48th(11000), the 48th is based on the 16th(10000), and the 16th is based on the 0th revision. This reduces the number of revisions needed to reconstruct a file, from O(R) to O(log(R)), where R is the number of previous revisions of a file.

6.3 Transactions

When the user does a commit, he or she starts a new *transaction*. Subversion transactions guarantee atomicity. Either all updates in the transaction are applied to the repository, or none of them are applied. Subversion transactions are persistent objects. A client can stop a transaction, disconnect, and at a later time reopen it. And then either abort or finish it. The transaction is not removed until either the client commits or aborts it, or it is explicitly removed by the administrator or a cleanup-script.

The result of a transaction can be one of three. Either it fails, leaving the repository in the same state as it was before the commit, or it succeeds, updating all the files to the same state as the ones in the users working copy, or it is merged with the results of another transaction.

A transaction can fail for several reasons, the most common being conflicts. If two developers both check out the file */src/foo.c* with revision number 1. Both developers change the file and commit. Subversion will let the first commit, to complete, through. When the second commit starts, subversion will detect that it tries to update a file with last change in revision 2, with a file being based on revision 1. An error message is then generated and returned to the user, and none of the changes made in the transaction applied to the repository. If the developers change different files, */src/foo.c* and */src/bar.c* then the changes would be merged, producing a new revision which does not correspond to the working copy of any developer. A transaction could also fail from an internal error, power failure, cosmic rays, etc... The transaction system has to be designed so that the repository is left in a consistent state, and all unfinished transactions at the time of the crash are aborted. The filesystem backend is responsible for storing and committing transactions so that they are guaranteed to be atomic.

fsfs stores the transactions in the transaction directory in the repository, giving each transaction a unique name derived from the base revision of the transaction. The transaction-directory contains one transaction file and a file for every node-revision modified in the transaction. Meta-information is stored in the transaction file, and node-revision data and properties are stored in the node-revision files.

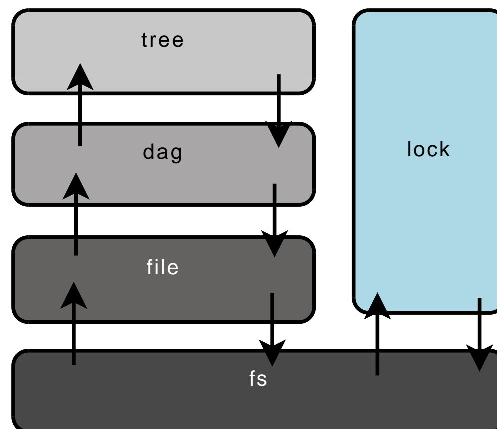


Figure 6.1: fsfs overview

6.4 Revisions

fsfs stores its data in several files. Revision-data is stored by having a file with the zeroth revision stored in the *revs* directory under the name *0*. The zeroth revision contains nothing but an empty root directory. Each succeeding revision is created by adding a new file, with the name of the new revision number to the *revs* directory. Node-revisions are stored in the revision-file for in the transaction to which they belong. The revision-file starts with a header that describes the paths that were changed from the previous revision, and offsets into the file for each node-revision. The node-revisions are stored at the specified offsets. Depending on what type of update was done to the node the revision file might contain the data, properties or both for the updated nodes. Node-properties are stored as a serialized hash of property-names and values. How data is stored depends on several factors, most importantly what the node-type is. Revision-properties are written to a similarly named file in the *revprops* directory.

When a transaction is committed, subversion goes through all nodes stored in the transactions directory, removing any references to the transaction. Then it generates a *proto-revision* file, which looks exactly like the revision-file that would be used for the next revision. If there are no conflicts or locks that hinder the commit of the revision, the proto-revision is then moved to the *revs* directory. Since moving a file is an atomic operation (either the file is moved successfully, or it is not moved at all) operation on most filesystems, it guarantees atomicity for subversion.

6.5 fsfs layers

The fsfs system can be viewed as a set of layers each providing a different abstraction-level of the filesystem backend. These layers do not strictly match the names of the c-modules in the implementation, nor are they mentioned in the documentation, bur are nevertheless, in the authors opinion, a good abstraction used when dealing with the fsfs implementation.

The layers of fsfs, are shown in figure 6.1, from bottom to up: the *file-layer*, the *dag-layer* and the *tree-layer*. In addition there is a *locking* module, which is works directly with the filesystem, and which does not deal with the other layers in fsfs.

6.5.1 The file-layer

The file-layer is responsible for storing nodes, both mutable and immutable to disk. Ie it works with both transaction and non-transaction nodes. In addition it does house-keeping tasks, like updating the counters used for the next copy-id and the next node-id, and is assigned the task of assembling the deltas stored for a files data, so that the upper layers can read a file like an ordinary stream.

6.5.2 The dag-layer

The dag layer lies above the node-layer, and provides all essential filesystem operations, but exposes the filesystem's internal structure. Nodes, as they are exported by the dag-layer are shared between revisions. The client of the dag-layer has to do any cloning itself.

6.5.3 The tree-layer

The job of the tree-layer is to take a filesystem with lots of node sharing going on, the filesystem as it is exported by the dag-layer, and make it look and act like an ordinary, tree based filesystem without sharing.

The tree layer is the one responsible for sharing of files/directories between transactions. To do this it employs copy-on-write semantics; as soon as the caller modifies a file/directory, the appropriate node-revisions are cloned. Any previous references to the changed node are changed so that they reference the new node-revision. If the parent-directory has not been changed before in the transaction, then the tree-layer creates a clone of it and its parents too.

Chapter 7

Design

pfs, the pesto-based filesystem backend, is based on fsfs. It reuses most of the code from fsfs, but the node-layer has been replaced with code that writes to pesto instead of writing directly to disk. This changes how transactions and revisions are stored and has a few other side-effects.

When implementing pfs we have to chose what information should be stored in pesto and what information should not be stored in pesto. As stated earlier, pesto does not support modification or deletion of previously stored data. This makes pesto inconvenient for storage of temporary data.

Locks in subversion are used when a user wants to hinder other users from modifying some part of the tree in the repository. A lock is held by a user until the user either frees it, or the lock is forcibly broken by the administrator. In principle, one could add locks to pesto, by either creating a lock object for each lock, and adding a lock-cancel object as a successor to the lock when it is opened, or having a lock object for all locks, updating its successors when the locks change. However, only the current state of the locks is interesting to the user, and lock history is not really worth saving. If the locks were stored in pesto, there would be no way of removing unlocked locks. We therefore leave the locks on the local filesystem in pfs, and postulate that locks are only valid on a per host basis. If locks are to be implemented in a distributed environment with multiple hosts serving the same repository, then a locking protocol should be implemented without using pesto. Alternatively, pesto could be extended to support locking.

Transactions are, in subversion, persistent objects. A transaction is not deleted until it is either aborted by the client, aborted due to an internal error or deleted by the administrator. This means that a client-program can create a transaction and at a later time reopen and finish it. This could be an argument for putting transactions in pesto, but transactions change often before being committed, and are often aborted before being committed. This makes transactions impractical to store in pesto. We therefore store transactions locally, and state that a transaction can only be reopened on the same host/repository that it was initially created on.

Revisions and revision-properties in subversion are created once and never modified after they are created. They are therefore ideal for storage in pesto. Nodes and node-revisions that are part of a transaction should therefore be stored in pesto. Each node-revision stored in pesto

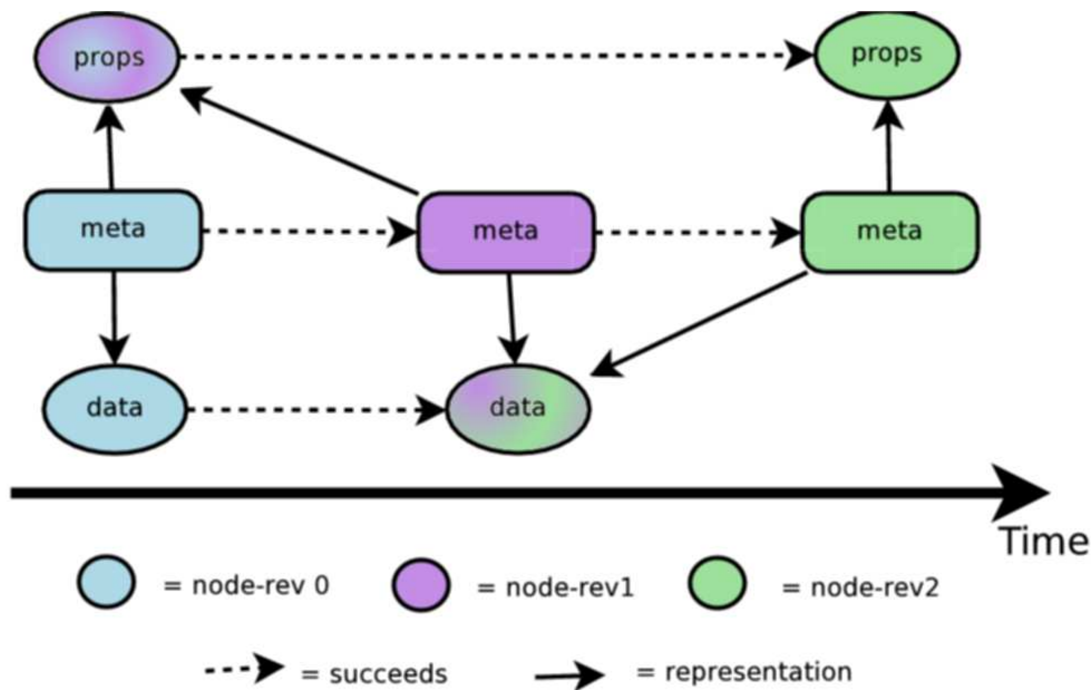


Figure 7.1: Timeline for a node

should be a successor of the previous node-revision. Data and property representations for the nodes should be successors of the previous data and property representations for that node.

7.1 Nodes

A node in pfs, like in fsfs is a set of one or more node-revisions. Each time a change to the node is committed a new node-revision is created. Each node-revision knows the number of node revisions preceding it, and its immediate predecessor.

A node-revision is made up of three parts, all stored as separate pesto objects. The tree parts are, the meta-object, the data-object and the properties object. Data is the file-data of the node, if the node is a file, or the directory entries, if the node is a directory. Properties contains all properties of the node that are viewable by the user/client while the meta-object only contains internal information about the node used by pfs. The meta object contains references to the data and properties objects. When a new node-revision is created only the references to the changed object is changed. This means that the new node-revision will reuse data from the previous node-revision if only the properties change, and vice versa, if the data changes, then the properties-object can be reused. A new version of the meta-object is created for every node-revision.

The timeline of a node can be viewed in figure 7.1. A node with a data and a properties object is created in revision 0. In revision 1 the data object is changed and in revision 2 the properties object is changed.

The translation of pesto objects to pfs-objects to subversion objects is shown in figure 7.2. The node-revision, and representation info are stored as one pfs object, the nodrevision-meta object. The contents of data and properties are stored as separate pesto-objects. Each object is created as the successor of the same type of object of the previous revision of the node.

7.1.1 Noderevision meta-object

Each node-revision is uniquely identified by its copy-id, in contrast to fsfs where a node-revision is only uniquely identified by the combination of {node_id,copy_id} and either its transaction-id or revision number. Each node is also assigned a unique node-id, which is set to the copy-id of the first node-revision of the node. Every node-revision that belongs to that node is tagged with that node-id.

Each node-revision also stores the id of its predecessor, so that it is easy to trace the history of a node. When the node-revision is committed to pesto, it is created as a successor of the parent-id, thereby inheriting all policies associated with the predecessor.

The node-revisions kind and path are stored in the meta-object. Note that the nodes kind should not change from revision to revision, while the path can, if the node is moved or re-named.

7.1.2 node-properties

The properties object contains the pesto-identifier of its predecessor and a list of property-names and the associated property values. When a property is changed a new node-properties object is created. Properties from the predecessor are copied to the new object, and the new property is set/un-set. When a properties object is committed to pesto, it is stored as a successor of the previous version of the nodes properties object, or a as a new object, if the node doesnt have any property-objects yet. This way, the properties object inherits the policies from its predecessor.

7.1.3 node-data

The data object contains the pesto-identifier of its predecessor and a binary-blob with the data for node. When the data-object is committed, it is stored as a successor of the previous version of the nodes data object, or as a new object, if the node doesnt have any previous data.

Directory-entries are stored as a list of {id,path}, where id is the pesto-identifier of the node-revision of the node stored in the directory, and path is the name of the node. Each time one of the entries is updated, the data-object is updated to point to the new directory-entry. This in turn updates the meta-object, which triggers an update of the parent-directory.

File-data is always stored as plaintext [1] , not as a delta against another version, as it is done in the other backends . While this does have the obvious drawback of requiring more space, it also has a few advantages. The extra space requirement is not that bad, considering that text-files usually do not require much storage space to begin with, and binary files are stored

[1] Plaintext, as opposed to delta, not as decrypted ciphertext

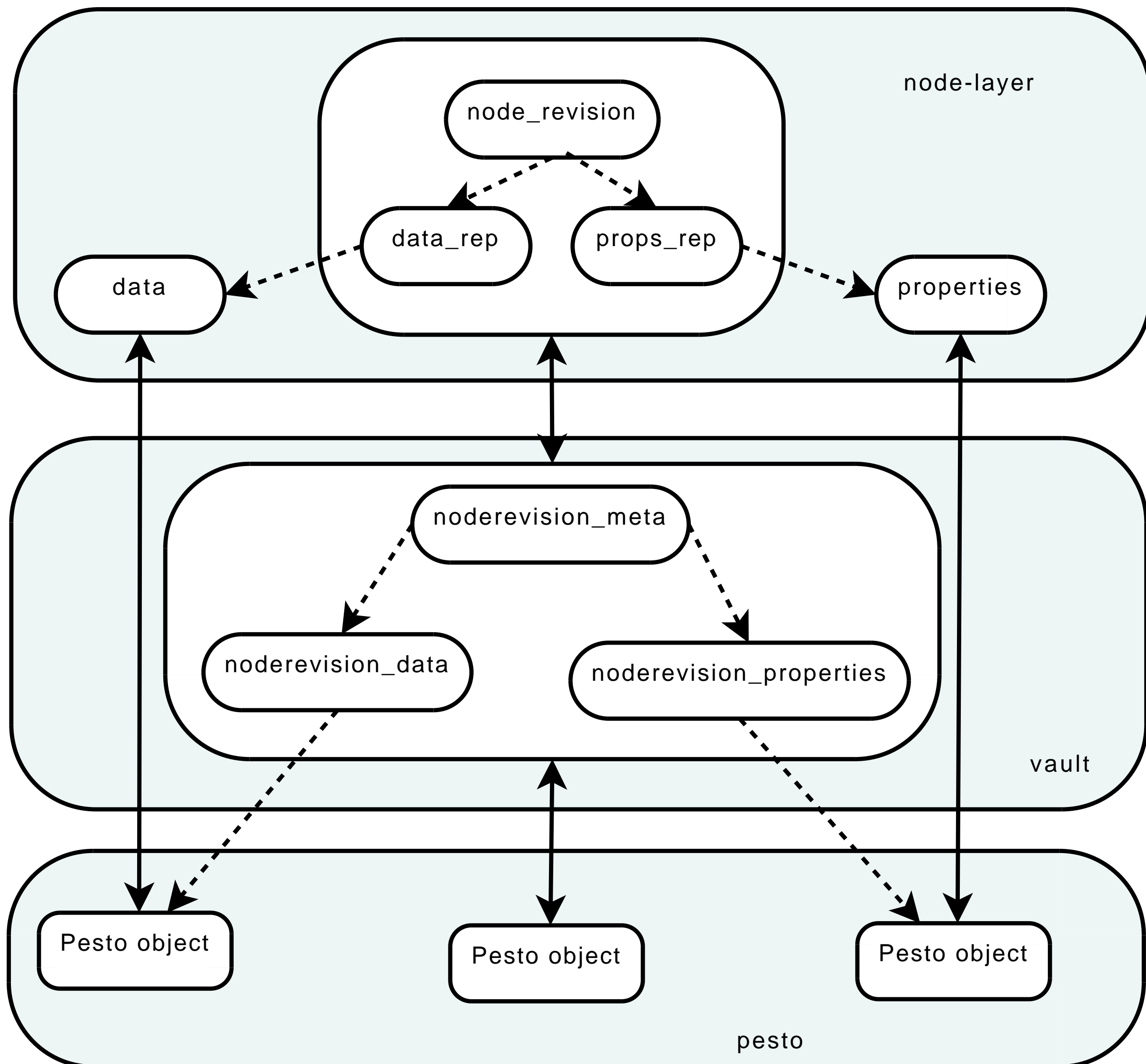


Figure 7.2: Translation of a pfs node-revision into pesto and subversion objects

```
noderevision_meta:
  node_id          (GUID)
  copy_id          (GUID)
  predecessor_id   (GUID)
  node_kind        {dir,file}
  node_path        (path)
  data_id          (GUID)
  prop_id          (GUID)
```

Figure 7.3: The noderevision meta-object

```
node_properties:
  predecessor_guid (GUID)
  properties       ({propname,propval}[])
```

Figure 7.4: The node properties-object

```
node_data:
  predecessor_guid (GUID)
  data             (byte[])
```

Figure 7.5: The node data-object

as plaintext for all storage backends used by subversion. Checking out a file becomes cheaper than when using the other backends, since any revision of any file can be accessed directly in plaintext. While bdb only allows plaintext access to the head-revision, and fsfs only allows plaintext access to the first revision.

Storing a file is also less expensive, in terms of file access, than in the other backends. bdb has to update all/some of the previous versions, when the file is updated. fsfs has to reconstruct the previous version of the file so that it can create a delta against it. pfs only has to write the data directly to the new file.

However, the main reason for always storing file-data in plaintext, is that it should be unnecessary to be able to read the previous revisions of a file to have access to the current revision. This means that it is, for simple commit and checkout operations, only necessary to keep the leaves of the vtree for each pesto-object locally. And that it is possible to give somebody access to a file revision managed by the repository without using subversion.

This can be useful for running a local repository. The local repository can then cache only the most recent revisions of each file, fetching older revisions from the main repository only if they are required. If the main repository should be unavailable for some reason, it would still be possible to commit to the repository and check out the most recent revisions.

Another scenario for which this might be useful is to allow read-access to some files in the repository outside of subversion. The id of the file could be distributed by another channel and all interested parties could, with the correct key, view it, and any future versions, as long as the policy does not change.

7.2 Revisions

Subversion stores some properties for each revision, some of which can be set by the user, while others are set by subversion. This includes the time at which the revision was committed, who committed the revision, etc... It also stores which paths were changed in the revision, and a short log message which is usually written by the user before a commit. To store this information we create a revision-anchor (see figure 7.7) object in pesto.

When the zeroth revision is created a root node (/) is created, and its pesto-id is written to the revision-anchor for the zeroth revision. The pesto-id of the revision-anchor itself is written

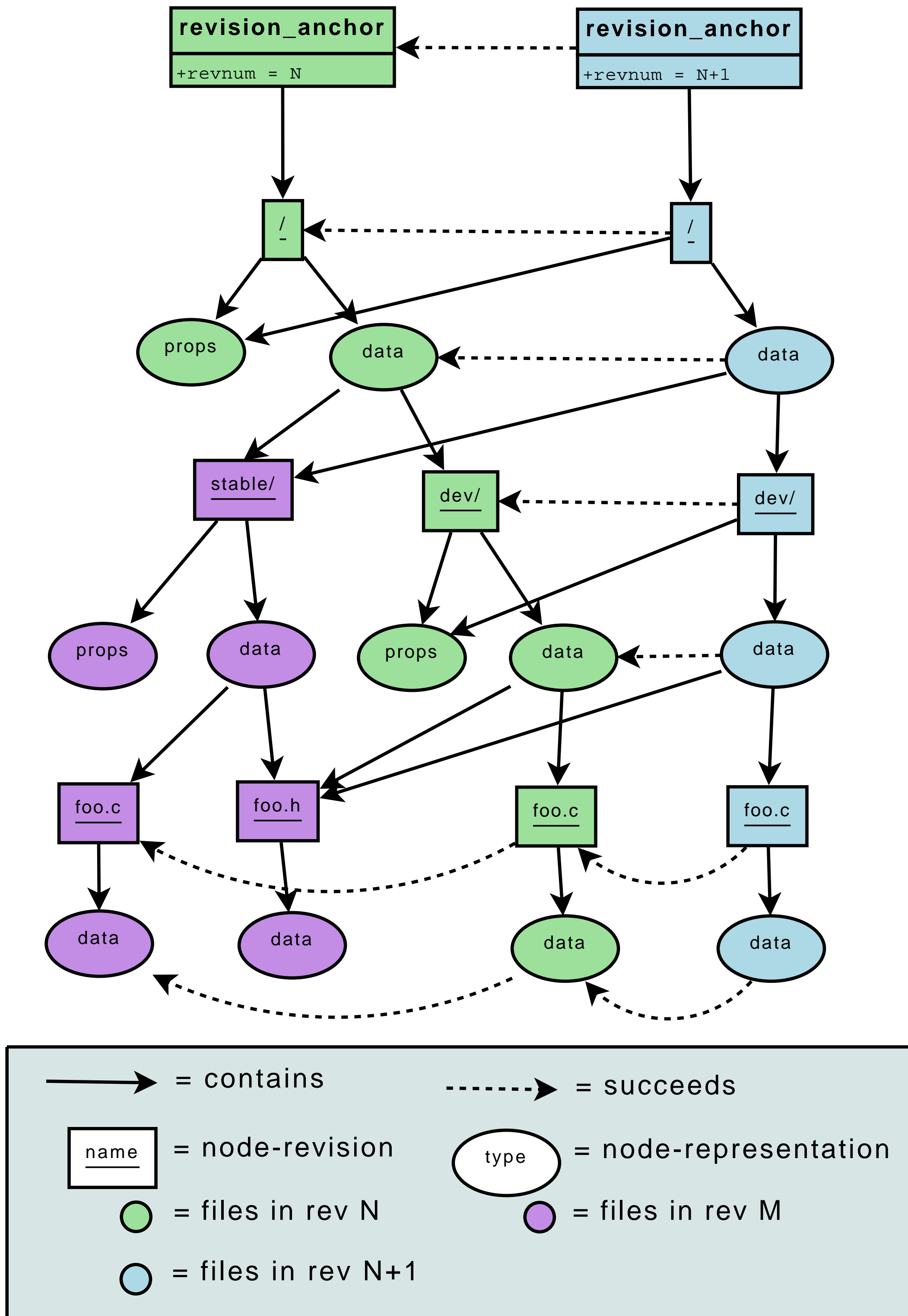


Figure 7.6: pfs revision

```
revision_anchor:
  revision_number (integer)
  id              (GUID)
  properties      ({propname,propval}[])
  root_node_id    (GUID)
```

Figure 7.7: The revision-anchor object

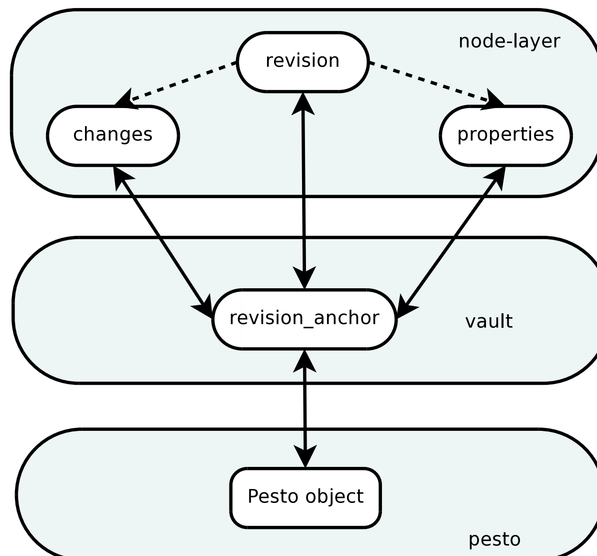


Figure 7.8: Translation of a pfs revision-anchor into pesto and subversion objects

to the disk. All subsequent revisions get their own revision-anchor. When a new revision-anchor is added to pesto, it is created as the successor of the previous revision-anchor. This way it is possible to find the revision-anchor for all revisions when only the id of the zeroth revision-anchor and the revision-number is known.

We store the pesto-id of the revision-anchor for revision zero in the. Future revisions are created as successors of the previous revision. To access a specific revision we use pesto to iterate through all the pesto-tree of the revision-anchor looking for the revision-anchor with the correct revision-number.

To illustrate, suppose that the client tries to access */foo/dev/bar.c* from revision 42. pfs will then first look up the revision anchor for the 0th revision from pesto. This is straight forward since the pesto-id of the 0th revision-anchor is stored locally. Then pfs looks through all successors of the 0th revision-anchor, checking their revision number. If a revision-anchor with revision number 42 is found, its root-node is then used as the starting point to resolve the path. Since all parents of a changed node are updated when the node is changed, we know that the root-node will always be updated if a node is updated.

Node-revisions belonging to a revision are stored in pesto. The node-revision meta-object is stored as the successor of the previous node-revision meta-object. The properties-object is stored as the successor of the previous properties-object, and the data-object is stored as the

successor of the previous data-object.

The figure 7.6 illustrates a simple repository. A file, *foo.c*, has been modified in the revision N, and revision N+1. And the properties for / have been modified in revision N. As we can see from the figure, all the parent-nodes for *foo.c* are updated whenever *foo.c* is updated. A directory, */stable* was created as a copy of */dev* in revision M. We can see that *foo.h* has not been modified after that, as a result of that M, N and N+1 all reference the same node-revision for both */dev/foo.h* and */stable/foo.h*.

The translation of pesto objects to pfs-objects to subversion objects is shown in figure 7.8. The revision-anchor as is seen by the node-layer as three separate objects, the changes, revision and properties objects. They are combined into the revision-anchor object which is stored as one pesto-object.

7.3 Transactions

Subversion stores transaction-properties in a transaction. Some of the properties are used as revision-properties when the transaction is committed, while other are used internally by subversion. In addition subversion stores the paths that were changed in the transaction and the node-revisions which were created during the transaction. To access a transaction we define a transaction anchor (see figure 7.10).

The anchor contains the revision on which the transaction is based on, the transaction id, a list of properties for the transaction, paths that were changed in the transaction, and the id of the node-revision of the root node (/) for the transaction.

When a new transaction is started, pfs first finds the newest revision-anchor and bases the transaction on it. The base-revision is set to the revision-number of the revision-anchor. Note that a transaction is guaranteed to update the root-node at some point, a new node-revision for the root-node is therefore created as soon as the transaction starts. Later, when a node is accessed in the transaction, the root-node of the transaction is used as base to find the node.

Meta-objects for modified node-revisions, and modified data and properties objects are also stored in the transaction.

The figure 7.9 illustrates a transaction that changes the data for the file-node */dev/foo.c*. Note that only the meta-objects and representations that were changed are added to the transaction.

The translation from files to pfs-objects to subversion objects is shown in figure 7.11. The transaction-anchor as is seen by the node-layer as three separate objects, the changes, revision and properties objects. The transaction-changes are written directly to the changes file in the transaction, and the properties and transaction objects are combined into the transaction-anchor which is stored as one file.

7.4 pesto abstraction

An abstraction layer is added between the file-layer of pfs and the filesystem. We call this abstraction layer the *vault*. The vault stores node-revisions and revision and transaction anchors. And supports the operations shown in table 7.1.

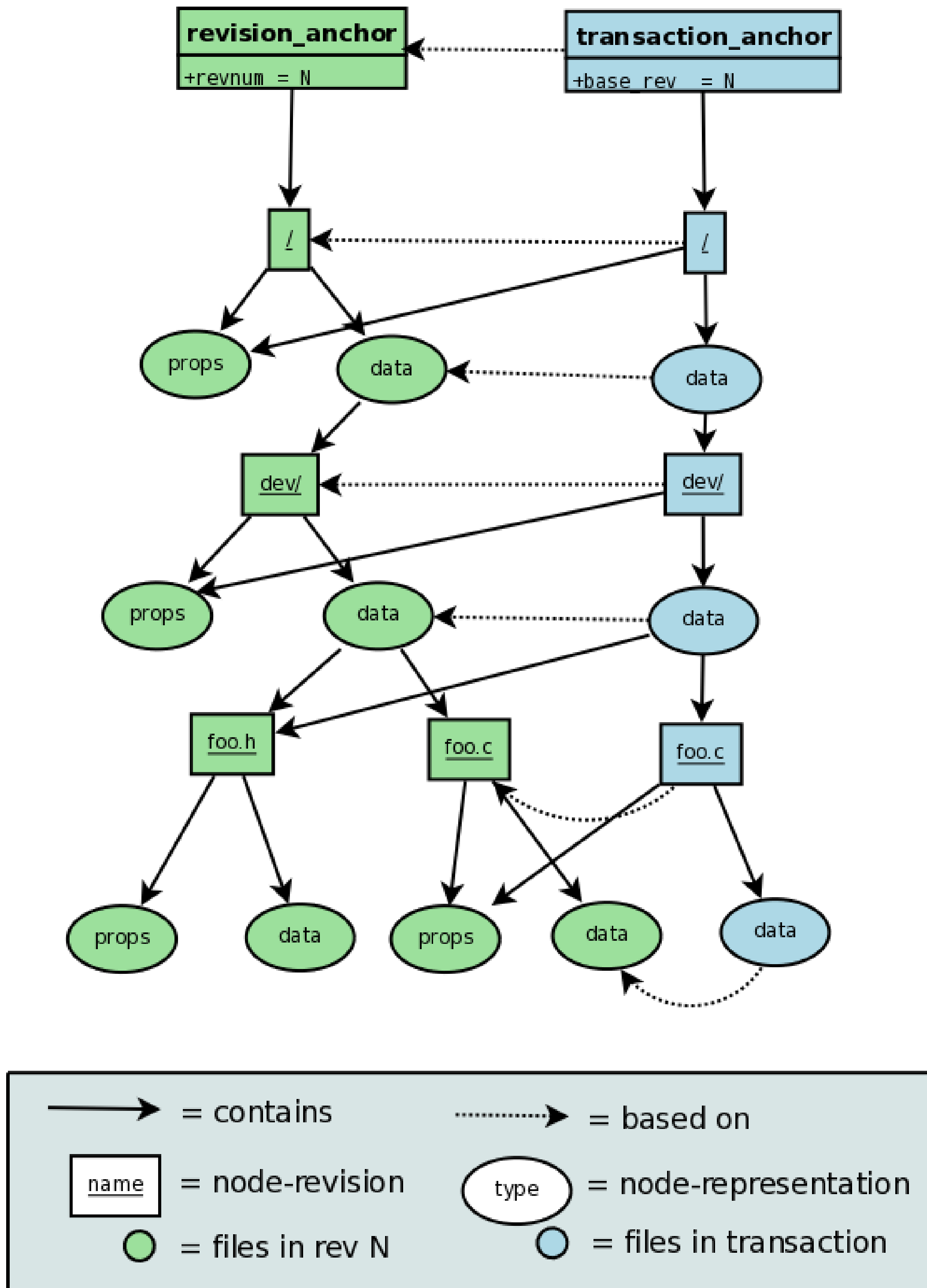


Figure 7.9: pfs transaction

```
transaction_anchor:
  base_revision   (integer)
  id              (GUID)
  properties      ({propname,propval}[])
  changes         ({change,path}[])
  root_node_id    (GUID)
```

Figure 7.10: The transaction-anchor object

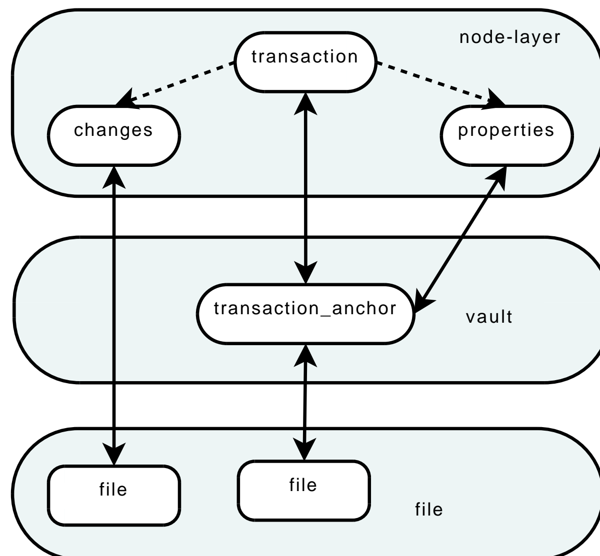


Figure 7.11: Translation of a pfs transaction-anchor into files and subversion objects.

Operation	transaction	revision	anchor	meta	data	props
fetch	true	true	true	true	true	true
put	true	false	true	true	true	true
remove	true	false	false	true	implicit	implicit
pestify	true	false	true	implicit	implicit	implicit

Table 7.1: Vault operations

The operation (fetch,{type,id}) fetches the object with the type type from either a revision or a transaction. The fetched object can be the anchor for a revision or transaction, the meta-object for a node-revision, data-object or a properties-object. If an object is in a revision, it is looked up in pesto, the corresponding pesto object is decrypted and written to a file. The file is then parsed and a new pfs_vault object is created. If the object is in a transaction, it is read directly from a plaintext file in the transaction directory, and a new object is created by parsing the file.

The operation (put,{object,type,id}) takes an object with the type, type from memory and stores it with the id given in id in a transaction. The object can be a transaction-anchor, a node-revision meta-object, data-object or a properties object.

The operation (remove,{object,id}) takes a node-revision meta-object and removes it from a transaction. If the object has any data or properties objects registered with it in the transaction, then they are also removed.

The operation (pestify,{txn_anchor,id}) takes a transaction anchor with the id id and creates pesto-objects from the associated transaction. It iterates over the meta-objects for the node-revisions created in the transaction and adds any data or properties objects which were created in the transaction. Then it writes all meta-objects to pesto, and finally creates a new revision anchor from the transaction anchor, and adds it to pesto.

The structure of pfs can be viewed in the figure 7.12. A new layer, called *vault* has been introduced between the file-layer and the filesystem. This layer works both with data stored in pesto and with the filesystem.

7.4.1 Committing a transaction

Objects are meant to be fetched from the vault before they are used, and put back when they have been changed. When all updates are done, and the transaction is committed, this is done by invoking the pestify() operation on the transaction-anchor, which first pestifies all objects changed under the transaction-anchor, before pestifying the transaction-anchor. The changed objects are only made accessible after the transaction-anchor has been committed, which means that the transaction is only really committed when the transaction-anchor has been pestified.

7.4.2 Checking out a revision

To check out a revision the id of its revision-anchor is first looked up. Then the revision-anchor is fetched through the vault. All node-revisions reachable from the revision-anchor are then fetched from the vault, and their data is passed to the upper layers.

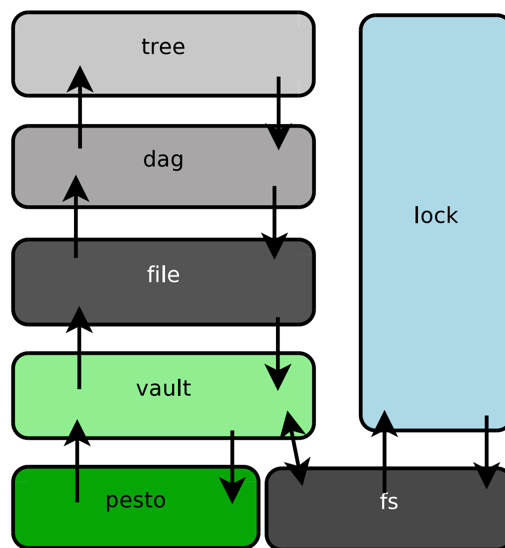


Figure 7.12: pfs overview

Chapter 8

Implementation

8.1 Identifiers

The identifiers used by fsfs have been changed to suit pesto better. Subversion itself does not really care about how the identifiers are stored. All it sees is a long text-string. Identifiers are originally stored as a text string of base-36 numbers. pfs instead uses pesto `GUIDs` which are 128bit keys, stored as a base-16 encoded integer. The `GUIDs` are randomly generated, which gives us 2^{128} different identifiers to chose from, which makes the possibility of a collision, two objects getting the same id, quite unlikely, assuming that the random number generator used is any good. By generating ids randomly, we remove the burden of keeping track of used identifiers from the backend, at the expense of increased disk usage. Using `GUIDs` for identification is also convenient, since it removes the need to convert between ids used in pesto and subversion ids. Alternatively a file could be used to store the mappings between pesto and subversion ids, but access to the mapping-file would have to be serialized, so that two transactions do not try to update the file simultaneously, not to mention that it would be a performance bottleneck.

Nodes are identified by the `GUID` of their first node-revision. The repository is identified by the `GUID` of the zeroth revision anchor. Transactions are identified by the `GUID` of the transaction-anchor. When the transaction is committed the `GUID` of the transaction-anchor is used as the `GUID` for the revision-anchor for the new revision.

8.2 `pfs_vault`

The module `pfs_vault` implements the vault abstraction layer described in the design chapter. The file-layer translates between `pfs_vault` objects and their corresponding subversion objects. Normally the file-layer first tells the `pfs_vault` to create a new version of an object in the transaction. Then it fetches the object from `pfs_vault` and translates to its subversion equivalent. Subversion then modifies the object and gives it back to the file-layer, which writes it back to the `pfs_vault`. When the transaction is finished, the file-layer tells the `pfs_vault` to create pesto-objects from all the objects in the transaction.

```
typedef struct{
  apr_pool_t   *pool;
  GUID_P        guid;
  int           base_rev;

  vault_data_t *data;

  svn_fs_id_t *root_id;
  svn_fs_id_t *base_id;
} vault_txn_t;
```

Figure 8.1: The `vault_txn_t` structure.

Objects that are stored in pesto are decrypted by libpesto and written to a temporary file. `pfs_vault` then parses the object, and unlinks the temporary file. Objects that are in a transaction are read from/written to the corresponding file in the transaction directory. The transaction resides in the *db/transactions* directory of the repository. Each transaction gets its own transaction id, which is a randomly generated `GUID`. The transaction directory's name is set to the id of the transaction. This makes it easy to determine whether an object is stored in a transaction, and makes it easy to get the list of all current ongoing transactions.

Each object in the vault has a memory-pool associated with it. The object is allocated from the pool. When the objects lifetime expires, all its allocated memory can be freed simply by freeing its pool. Although, currently, the lifetime of an object does not exceed any function in the file-layer, it could prove useful for future implementation of a cache.

8.2.1 Transactions

The `vault_txn_t` (see figure 8.1) structure is the `pfs_vault` equivalent of a transaction-anchor. It stores all necessary information about a transaction. `guid` is the transaction id, `base_rev` is the revision number the transaction was based on. The `base_id` is the id of the / node-revision of the revision the transaction was based on, and `root_id` is the id the / node-revision of the transaction. The `data` part points to the common data shared between transaction-anchors and revision anchors. The `vault_data_t` structure holds the properties for the transaction in the `props` attribute. The properties are stored as a hashtable of {`propname,propvals`}, where `propname` is the name of the property, and `propval` is a subversion string holding the value of the property. The `changes` hash holds the changes done so far in the transaction. The changes are stored in the hash as {`changed_path,change`}, where `changed_path` is the path of the changed node, and `change` is a string that describes what was changed; whether properties or data were changed for the node, the id of the new node-revision, the type of the node, and where the node was copied from, if the change was a copy. `root` holds the `GUID` of the / node-revision.

A pfs transaction is stored in the *db/transactions* directory. Each transaction creates its own sub-directory in the *transactions*-directory. The sub-directory is named with the `GUID` of the transaction, which insures that no two transactions try to write to the same directory. A new transaction creates the *changes*-file, the *txn*-file, and the *props* and *data*-directories.

```
typedef struct{
  apr_hash_t *props;
  apr_hash_t *changes;
  GUID_P      root;
  apr_pool_t *pool;
} vault_data_t;
```

Figure 8.2: The `vault_data_t` structure.

```
typedef struct {
  int           revision;
  GUID_P        parent_guid;
  GUID_P        guid;
  apr_pool_t   *pool;
  vault_data_t *data;
} vault_rev_t;
```

Figure 8.3: The `vault_rev_t` structure.

Each node-revision created in the transaction is stored in the transaction-directory, with the name `$COPY_ID.node`, where `$COPY_ID` is the copy-id of the node-revision. The properties-representations are stored in the *props* directory, with names on the form `$ID.props`, where `$ID` is the `GUID` of the properties-representations. Data-representations are stored analogous in the *data* directory, with names on the for `$ID.data`, where `$ID` is the `GUID` of the data-representation.

The *txn*-file store the `vault_txn_t` and the `vault_data_t` structure, except for the `changes`-hash. Changes are instead written to the changes file. This is done, since the changes of a transaction are usually only read at the end of the transaction. In a large transaction the number of changes can be quite high, and the transaction-anchor is read for every change done in the transaction. If the changes were kept in the *txn*-file, it would mean that for every change pfs would have to read in a quickly growing file, and thereby run super-linearly slower.

8.2.2 Revisions

Revision-anchors are represented by the `vault_rev_t`-structure. `revision` holds the revision number of the revision, `parent_guid` is the `GUID` of the previous revision, or `NULL` if it is the zeroth revision. `guid` is the `GUID` of the revision. `data` holds the data that is common for both a revision and transaction anchor. When a revision is committed its data attribute is simply copied.

`pfs_vault` stores revision-anchors in pesto, identified by the `GUID` of the revision-anchor, and stored as the successor of the revision anchor of the previous-revision.

To read from a revision, it is first decrypted into a temporary file, and parsed from it. When the `vault_rev_t` has been parsed, the temporary file is deleted. A revision can only be written by

```
typedef struct {
  apr_pool_t      *pool;           // pool from which node was allocated
  GUID_P           txn_id;
  GUID_P           id;             // guid of current incarnation
  GUID_P           original_id;    // guid of the first incarnation
  GUID_P           predecessor_id; // guid of the previous incarnation
  svn_node_kind_t  kind;
  char            *created_path;
  int              predecessor_count;
  /* If this node-rev is a copy, where was it copied from? */
  char            *copyfrom_path;
  int              copyfrom_rev;
  /* Helper for history tracing, root of the parent tree from whence
     this node-rev was copied. */
  char            *copyroot_path;
  int              copyroot_rev;
  vault_representation_t *data;
  vault_representation_t *props;
} vault_noderev_t;
```

Figure 8.4: The `vault_noderev_t` structure used by `pfs_vault`.

taking a transaction-anchor and converting it to a revision anchor through the *pestify* operation. The resulting revision-root is the written to pesto.

8.2.3 Node-revisions

`pfs_vault` defines the type `vault_noderev` (figure 8.4) for storage of node-revisions. The corresponding subversion type is `node_revision_t` (8.3). The id of the subversion node-revision has been decomposed into a transaction `GUID`, node-revision `GUID`, and original `GUID`. The `original_id` "points" to the first node-revision of the node. The `id` is the id of the current node-revision, and `txn_id` is the id of the transaction that node-revision is part of. If the node-revision is committed to pesto, then the transaction id is set to `NULL` to indicate that. Rest of the attributes are equivalent to their subversion counterparts, except for the data and properties representations attributes, which point to vault-representations (8.2.3). Note that the vault-representations are part of the node-revision object, and are not stored separately.

The `vault_representation_t` (8.2.3) type stores information about a representation. Its size, checksum, revision in which it was created and the `GUID` of it and its immediate predecessor. Note that subversion keeps a few extra attributes for its representation objects (8.2.3). This is extra information is used by subversion to reconstruct the representation from deltas, but is not needed by pfs since representations are always stored as plaintexts. The transaction id is set by subversion to indicate which transaction the representation is in, which is redundant, since a representation is either in a revision (committed to pesto) or in the same transaction as the node-revision to which it belongs. Note that each node-revision will "adopt" its parents data and properties representations if they are present, until it creates some on its own. A

```
typedef struct{
  svn_node_kind_t kind;
  const svn_fs_id_t *id;
  const svn_fs_id_t *predecessor_id;

  const char *copyfrom_path;
  svn_revnum_t copyfrom_rev;
  svn_revnum_t copyroot_rev;
  const char *copyroot_path;
  int predecessor_count;
  representation_t *prop_rep;
  representation_t *data_rep;
  const char *created_path;
} node_revision_t;
```

Figure 8.5: The `node_revision_t` structure used by subversion.

```
typedef struct {
  unsigned char   checksum[APR_MD5_DIGESTSIZE];
  int             revision;
  svn_filesize_t  expanded_size;

  GUID_P          guid;
  GUID_P          parent_guid;
} vault_representation_t;
```

Figure 8.6: The `vault_representation_t` structure used by `pfs_vault`.

node-revision cannot unset its data or properties representations after they have been initially set.

Storage

Node-revisions use a simple text-format for storage. The attributes are printed in an ordered fashion to a file. Each attribute being preceded by a string identifying which attribute it is. The type, however, is only used as a safety net to catch a corrupt node-revision, and the parser will halt if it gets an unexpected value. If the node-revision does not have a value for an attribute, the string "`NULL`" is printed. If the node-revision has a data-representation, then the `NODE_HAS_DATA` field of the file is set to "`true`" and the data-representation follows. If not the `NODE_HAS_DATA` field is set to "`false`". Analogous, the `NODE_HAS_PROPS` field is set to true if the node has a properties representation. A node-revision file is stored in the transaction-directory, and has the extension *.node*.

```
typedef struct
{
  unsigned char checksum[APR_MD5_DIGESTSIZE];
  svn_revnum_t revision;
  apr_off_t offset;
  svn_filesize_t size;
  svn_filesize_t expanded_size;
  const char *txn_id;
} representation_t;
```

Figure 8.7: The `representation_t` structure used by subversion.

8.2.4 Representations

The representations themselves are stored in separate pesto-objects/files, but all information about a representation is stored in the node-revisions to which the representation belongs.

Properties-representations

Properties representations are stored in a hashtable of {`propname`,`propval`}. Where `propname` is the name of the property, stored as a character array, and `propval` is a subversion string type. Subversion provides methods for reading and writing hashes of this type to/from files. To read a property for a node-revision with its properties representation stored in pesto, `pfs_vault` decrypts the pesto-object identified by the `GUID` stored in the properties-representations part of the `vault_noderev` for the node-revision to a file. Then reads the hash from the file to memory and returns it to the file-layer. The file-layer finds the correct property-field and returns its value. If the properties-representation is stored in a transaction, `pfs_vault` looks through the *props* directory in the transaction for a file named with the `GUID` of the property-representation. Then it parses the hash, and returns it to the file-layer.

Properties, can logically enough, only be set on node-revisions that are part of a transaction. When setting a property, `pfs_vault` will first try to get the properties-representation associated with the node-revision. To set a property `pfs_vault` tries first to read the current property-representation associated with the node-revision, if the node-revision has a property-representation stored in pesto, it is decrypted from pesto, and written to a new properties-representation in the *props* directory in the transaction. The new properties-representation is set to be a successor to the properties representation it was created from. If the node-revision does not have any properties-representation, a new, empty properties representation is created in the *props* directory. If the node-revision already has a properties-representation in the transaction, nothing is done. Then the hash is parsed from the properties-representation, the property is set in the hash, and the hash written back to the properties-representation in the transaction.

Data-representations

Data representations for directories are treated much the same as properties-representations. Directory-entries are stored as a hashtable of {pathname, description}, where pathname is the path of the directory entry, and description is a description of the directory entry, consisting of the full id of the directory entry {node_id, copy_id, revision_number / transaction_id) and the kind of the directory entry. If a node-revision for a directory node does not have any data-representation a new data-representation is created. If a previous data-representation exists, it is copied to the new data-representation, and the copy is set to be the successor of the original. The hash is then parsed from the *data*-directory in the transaction-directory, and the new directory entry written to the hash, whereupon the hash is written back to the data-representation.

If the node is a file, data-representations are not read from pesto before they are modified. Instead an empty file for the data-representation is created in the *data* directory. A new filestream is created to write to the file. The stream is one way, write only, meaning that it is not-possible to reposition the stream, or read from it. While writing to the stream, the number of bytes written are recorded, and a md5 checksum is computed from the written bytes. When the stream is closed, an upcall is done to the file-layer, and the data-representation of the associated vault_noderev is updated, and then written back to the transaction directory.

8.2.5 Caching

Currently no caching is used in pfs. Caching in pfs could be done on two levels. One, pfs_vault could cache the files that were decrypted from pesto on disk. Two, pfs_vault could cache the parsed node-revisions and revision-anchors in memory. Caching transaction-data would be more difficult, since there are no guarantees that more than one process might be working on the same transaction.

8.3 Working with transactions

To start a new transaction, pfs fetches the newest revision-anchor from pesto and creates a transaction-anchor from it. The transaction-anchors guid is randomly generated, and the parent_guid is set to the guid of the revision the transaction is based on. A new node-revision for the root node is created, and the transaction-roots data is updated accordingly.

Each subsequent node-revision that is created is then added to the transaction. When a node-revision gets added to a transaction a new node-file is written to the transaction directory. The predecessor count of the node-revision is incremented, and its predecessor_id is set to the guid of the previous node-revision. If the data of a node-revision changes it is added to the *data* directory, and if the properties of a node-revision are changed, a new file is added to the *props* directory.

8.3.1 Locking

Locking in pfs is implemented on a per-host basis, and no form of distributed locking is currently supported. To lock a node, a hash of the nodes path is created, and a file with the name

set to the hash of the path is created in the locks directory, containing the path of the locked node, and the owner of the lock. When a transaction is committed, all changes done in the transaction are checked against the files. If a path hashes to a file in the locks-directory, the lock is checked to see if the owner is the committer. If not, the transaction is aborted.

8.3.2 Committing a transaction

When all changes in a transaction is finished the client will try to commit the transaction. To do this pfs first acquires an exclusive file lock on the *write-lock* file in the repository. The write-lock is necessary to serialize commits to pesto, so that two processes do not try to commit the same revision simultaneously.

pfs then reloads libpesto's cache. This is done so that any new objects written to pesto will be seen. pfs then tries to fetch the newest revision-anchor. If the revision-anchor is the same as the revision-root that the transaction was based on, pfs will continue with the commit. If not pfs will return control to the tree-layer, which will try to merge the transaction with the newer-revisions. If a merge is not possible, a `Transaction-out-of-date` error will be returned.

If the changes can be merged, or if the transaction is based on the youngest revision-anchor pfs will start the commit. To do this pfs compresses the changes done in the transaction, removing changes to files that were later deleted, or file that were later removed, leaving one change per path. pfs then looks through the *locks* directory, to see if anyone holds a lock on the changed files. In another user holds a lock on the changed path, and the transaction was not started with the *Break-locks* property set, then the commit fails with a `Missing-lock` error.

If the current user holds all locks necessary for the commit, pfs goes through all node-revisions in the transaction directory. It then *pestifies* the properties and data-representation for each node-revision. For directory data, it looks through all directory entries and changes references to the transaction, to references to the revision to be committed, by removing the `txn_id` from the node-revision ids, and setting the `revnum` part of the id instead. When the data and properties-representations have been written for a node-revision, or if the node-revision has no new properties or data-representations, then the node-revision itself is *pestified*.

When all nodes have been *pestified* the transaction-anchor is *pestified*. To do this the changes are read from the *changes*-file and added to the transaction-anchor as a hash. Then a new `vault_rev` object is created. The `revision` field is set to the `base_revision + 1`, and `parent_guid` is set to the `guid` of the current youngest revision, and the `guid` set to the guid of the transaction-anchor. The `data` attribute of the revision-anchor is set to the `data` of the transaction-anchor. The revision-anchor is then written to a temporary file, which is then added to pesto. This means that all new node-revisions, created in the transaction can now be reached, which in turn means that the transaction is committed.

8.3.3 Aborting a transaction

Transactions can be aborted for several reasons. The client can simply abort the transaction, because the user wants to abort the transaction, ie presses `^C` during commit with the standard subversion client, `svn`. Or the transaction fails because some of the files it tries to change are

locked, or because the users working copy is out of data, and subversion is unable to merge it with the newest revision. Or because the client crashes, or the server crashes.

All aborts except the server crash are easy to handle. Either they happen before the transaction enters the commit-phase, or they happen after. If the abort comes after the transaction has entered the commit-phase, it is simply ignored. If it happens before, it is easy to abort the transaction. Since all of the transactions files are located inside the transaction directory, it is a simple matter of deleting the transaction-directory for the transaction, and cleaning up the memory used by the transaction.

If the transaction is aborted because of a server crash, things are slightly more complicated. The transaction-directory will not be cleaned up, because there is no failsafe way to determine if a transaction is abandoned, i.e not needed by anybody in the future, or if it is a suspended/paused transaction, or even in use. This is because subversion uses a model of several, independent processes that access the same data without any direct communication between themselves.

All of subversions filesystem-backends leave the abandoned transactions in place. To clean up transactions an administrator, or at least a cleanup script is needed, that can check all transactions in subversion, and delete any transaction older than a certain date.

It is worse yet, if the server crashed when a transaction has entered the commit-phase. Some of the transactions-nodes will already be written to pesto, while others are not. The revision-root will, however not have been written to pesto yet, since the transaction is still uncommitted. This means that a transaction cannot be partially committed, since no-one can read the committed node-revisions. This will however clutter up disks-space, both on the repository host, and on all hosts that replicate its pesto-objects.

A possible solution would be to try to re-commit a transaction after a crash. libpesto would return an error if an object with the same `GUID` was written twice, and if pfs ignored this errors and continued the commit, it would be possible to commit only the node-revisions and representations that were not committed initially. Unfortunately this is not possible because of subversions processes model. Other processes would not-be notified if one of the svn-server processes fails, and there is no way of starting the recovery-process.

This is a limitation of subversion, and it is impossible to mitigate this presently. The only thing we can do, is make sure that subversion does not crash, as it is done in both fsfs and bdb backends.

8.4 Interfacing with pesto

The interface to libpesto that pfs uses is implemented in the `pfs_util` module. The interface provides functionality for finding a pesto object by its `GUID`, writing new pesto objects, creating a new pesto backing directory and searching for a specific revision of a pesto object, only knowing the `GUID` of one of the revisions.

8.4.1 Initializing the pesto backing directory

When creating a new pesto repository `pfs_util` creates a new backing directory. To do this `pfs_util` generates a random owner-id, a random share-id and a user-key which it uses to

encrypt/decrypt its data. The owner-id and key are written to the *pv* file, and the share-id is written to the *ps* file in the backing directory. Then libpesto initializes the rest of the directory, creating the *META*, *TMP*, *DATA* and *LEAF* directories.

When a process at a later time accesses a pfs repository, `pfs_util` initializes it by reading from the backing directory, getting and setting the owner-id, share-id and keys right, and initializing libpesto with the proper keys and path to the proper directory.

8.4.2 Writing an object

pesto objects consist of a number of *key value pairs*, *kvps* for short. Common for all kvps is that a kvp can be decrypted separately from the rest of the object. Normally a pesto client only sets the data, name and type kvps. pfs extends the kvps by creating its own kvp, the revision kvp. When writing a pesto object pfs sets the revision-kvp to the revision to which the object belongs. For revision-anchors this is the revision for which the revision-anchor is root for. For node-revisions and representations it is the revision in which they were created/modified.

pesto supports reading from a file-descriptor or from memory. All pesto objects in pfs are currently being written to an intermediate file before being added to pesto. Currently all pesto objects from pfs are written to a file first, and only then added to pesto.

8.4.3 Accessing an object with a GUID

To access a pesto object with a known `GUID` is simple. `pfs_util` simply does a lookup in pesto for the `GUID`. If an object with a matching guid is found, then we have the correct object, otherwise the object is either not created at all, not written, or hasn't been transferred to the host yet.

pesto supports decrypting object to either memory or file. However, currently, pfs only reads from files. To read a pesto object, `pfs_util` creates a temporary file to which the objects is decrypted.

8.4.4 Accessing a version of an object

Sometimes it is useful to be able to access a revision of an object, without knowing the exact guid of that revision. pesto organizes objects in a tree structure, the *vtree*, with the root in the first created revision of the object. To find a specific revision of an object libpesto start with a root-guid, which can be the root of the vtree, or be a root of a subtree of the vtree. libpesto then iterates through the leafs of the tree, decrypting the revision kvp for the leafs. If there is a matching revision `pfs_util` decrypts that pesto-object. If the constant, `PFS_REVISION_LAST` is given, `pfs_util` instead searches through all objects in the vtree, recording the `GUID` of the object with the highest revision, and then decrypts the object with the highest revision number.

8.4.5 Finding revision roots

What is common for revision-anchors, is that there is one version for each revision. Since subversion creates a new revision each time a change is made to the repository the number of

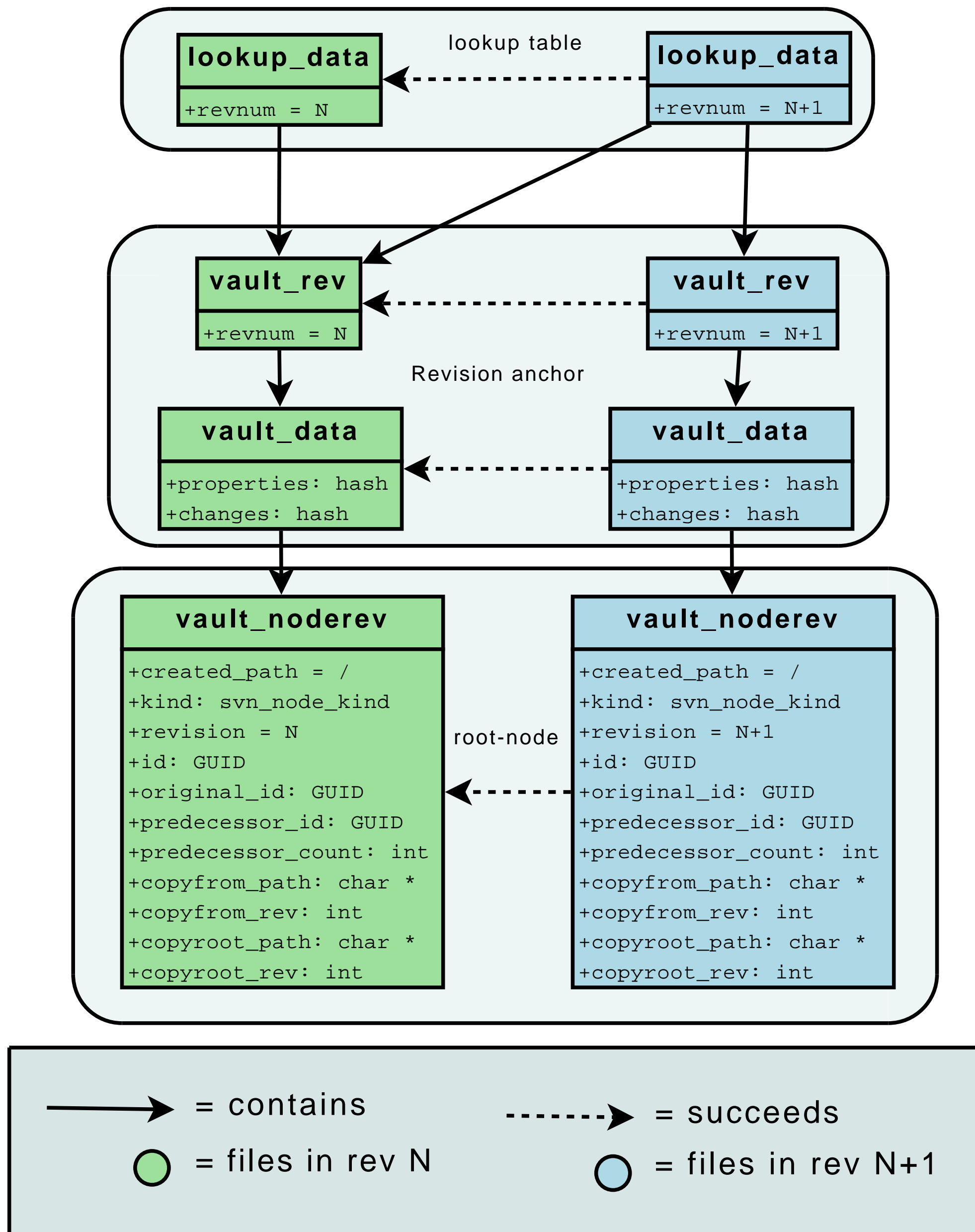


Figure 8.8: The lookup-table with revsion anchors and the root node-revision.

revisions can quickly grow. Initial testing with 1000+ revisions showed that finding a particular revision root by starting from the root of the revision-root vtree and looking for the particular revision was very slow. This was due to the fact that each encrypted revision-anchor up to the desired revision-anchor had to be read, and the revision kvp decrypted. As the size of the repository grows the linear search performed worse and worse. To mitigate this several approaches were tried.

Several caching variants were tried. The problem with caching is that the cache has to be populated at some time. libpesto allows reading of a nodes vtree from the root or from the leafs. Typical access patterns for subversion would be access to the HEAD of the repository, ie, the latest revisions. Which could be easily accommodated by reading the revisions from the last to the desired revision, caching all intermediate results to speed up access to those revisions in the future. The problem, is that subversion wants to access the revision-information for revisions which are copy-roots, ie revisions in which a copy of a node has been made. If a copy of a file currently being checked out was made in the 100th revision, and the current revision is the 10000th, it would mean that 9900 revision-anchors had to be read from disk, and the revision kvp decrypted. If there are "forks" in the revisions, the number would of course be even larger.

The final solution was to use a lookup table that contains the revision-number and guid of the corresponding revision-anchor. This *revision lookup-table* is stored in pesto each time a revision (including the 0th) is committed. Each subsequent lookup-table is added to pesto as the successor of the lookup-table for the previous revision. If a fork happens, two tables will be written, both containing the same-revisions up to the latest revision, which will have different guids depending on which table is used[1]. To check for existence of new revisions, pfs simply checks for any new leafs of the lookup-table, and reloads its lookup table with the highest revision-number.

This approach is much faster than iteration through all nodes in the vtree for the revision anchor, which is a O(N) operation, N being the number of nodes in the tree. Iterating through only the leafs of the lookup table is a O(L) operation, where L is the number of leaf-nodes in the vtree. libpesto caches the meta-data for any file that there is more than 7 revisions of. This makes access to files a O(1) operation, once the guid is known. In other words, to find any revision O(L*1) file access have to be done.

Note, that there is still some "painters algorithm" left in the code. The lookup-table will grow, with one entry each time a revision is added, The time reading a lookup-table will therefore grow with each committed revision. This can be reduced to a minimum, by having two tiers of lookup-tables, where the upper layer is a meta lookup-table, containing guids for lookup-tables, each holding 1000 revisions.

8.5 Functionality

Most of the functions defined in the subversion fs-api have been implemented in pfs. However, some functions were not implemented. The implemented functions provide enough functionality for pfs, so that it can be used for almost all operations supported by subversion. Functions

[1] This is a Good Thing(TM), since the history of what data revisions are based on is clearly visible.

for importing a repository and taking backup of a live repository were however not implemented. Backup functions are unnecessary, because pesto can be used to automate the backup process of a subversion/pfs repository. Pesto can replicate the data to backup storage without any need to pause commits to the repository. Loading a existing repository is also not fully supported. subversion tries to set the properties of a revision, after the revision has been committed when loading a repository dump-file via the `svn load` command. This is impossible since the corresponding revision-anchor has already been created and written to pesto (and possibly already been replicated to other nodes). This functionality is not critical, the commit-log message and timestamp are written before the revision is committed. A warning is printed every time the function is called, but `svn load` is allowed to continue.

Chapter 9

Testing

In this chapter we give a demonstration of how the pfs filesystem backend for subversion, implemented in the last two chapter works.

9.1 Demonstration

9.1.1 Demo configuration

The setup used to test pfs consisted of two computers interconnected over the university LAN. The machines were, *hydra.td.org.uit.no* and *pastaws0.cs.uit.no*. The individual configuration for the hosts are illustrated in the tables 9.1 and 9.2.

pastaws0 hosts the repository and has the modified subversion installed, while *hydra* runs the vanilla subversion, installed from the NetBSD package collection (pkgsrc), and acts as a client accessing the repository on *pastaws0*. All cpu and disk-intensive processes were terminated before doing performance tests.

9.1.2 Normal subversion operations

First we create the repository:

Host	hydra
CPU	Intel Pentium III 700MHz, 256KB L2 cache
Memory	384MB
Disk	2x IDE 7200RPM in Raid-2, mounted with noasync
Network interface	10Mbit
Operating system	NetBSD 3.0

Table 9.1: Configuration of *hydra*

Host	pastaws0
CPU	Intel Pentium IV 3.0GHz, 1024KB L2 cache
Memory	1024MB
Disk	SATA, mounted with noasync
Network interface	100Mbit
Operating system	NetBSD 3.1

Table 9.2: Configuration of *pastaws0*

```
oleg@pastaws0(~)$ svnadmin create repo --fs-type pfs
```

Then we check-out the repository using the *file* protocol for local access.

```
oleg@pastaws0(~WC/)$ svn co file:///home/oleg/repo
Checked out revision 0.
```

Note that pfs automatically creates a new pesto node for us. If we want to use an already existing node, the *META*, *DATA*, *TMP* and *LEAF* directories in the db directory have to be replaced with symlinks to the corresponding directories in the backing-store of the node, and the keys, owner and share ids written to the *pv* and *ps* files.

When the repository is created, we add the project foo.

```
oleg@pastaws0(~/WC/repo)$ cp -r ../../src/foo .
oleg@pastaws0(~/WC/repo)$ svn add foo
A         foo
A         foo/bar.c
A         foo/bar.h
```

And commit it as revision one.

```
oleg@pastaws0(~/WC/repo)$ svn ci -m ``Added foo to repository''
Adding         foo
Adding         foo/bar.c
Adding         foo/bar.h
Transmitting file data ..
Committed revision 1.
```

Then we modify the file */foo/bar.c*, and add a new file, *baz.c* and commit.

```
oleg@pastaws0(~/WC/repo/)$ svn add foo/baz.c
A         foo/baz.c
oleg@pastaws0(~/WC/repo/)$ svn ci -m ``Added baz.c''
Adding         foo/baz.c
Transmitting file data .
Committed revision 2.
```

We then check out the repository from *hydra*, and verify that the directory structure of the WC's is the same.

```
oleg@hydra(~/WC) svn co svn+ssh://pastaws0.cs.uit.no/home/oleg/repo
A    repo/foo
A    repo/foo/bar.c
A    repo/foo/bar.h
A    repo/foo/baz.c
 U   repo
Checked out revision 2.

oleg@hydra(~/WC/repo)$ find ./ |grep -v .svn
./
.//foo
.//foo/bar.c
.//foo/bar.h
.//foo/baz.c

oleg@pastaws0(~/WC/repo)$ find ./ |grep -v .svn
./
.//foo
.//foo/bar.c
.//foo/bar.h
.//foo/baz.c
```

We create the *STABLE* directory for stable branches, copy *foo* to the stable branch, edit *foo/bar.c* and commit.

```
oleg@hydra(~/WC/repo) svn mkdir STABLE
A         STABLE
oleg@hydra(~/WC/repo) svn copy foo STABLE/foo
A         STABLE/foo
oleg@hydra(~/WC/repo) vi foo/bar.c
svn ci -m ``created stable-branch, before changing bar.c''
Adding         STABLE
Adding         STABLE/foo
Sending        foo/bar.c
Transmitting file data .
Committed revision 3.
```

We then check-out the repository on *pastaws0*, and look at the log history of *STABLE/foo/bar.c*.

```
olegj@pastaws0(~/WC/repo) svn up
U    foo/bar.c
A    STABLE
A    STABLE/foo
```

```
A    STABLE/foo/bar.c
A    STABLE/foo/bar.h
A    STABLE/foo/baz.c
Updated to revision 3.
svn log -v STABLE/foo/bar.c

------------------------------------------------------------------------
r3 | oleg | 2007-05-25 15:13:51 +0200 (Fri, 25 May 2007) | 1 line
Changed paths:
   A /STABLE
   A /STABLE/foo (from /foo:2)
   M /foo/bar.c

created stable-branch, before changing bar.c
------------------------------------------------------------------------
r1 | oleg | 2007-05-24 20:47:06 +0200 (Thu, 24 May 2007) | 1 line
Changed paths:
   A /foo
   M /foo/bar.c
   M /foo/bar.h

Added foo to repository
------------------------------------------------------------------------
```

9.1.3 Collaboration

We extract the *GUID* to the data of a file stored in the repository. An external program can then be used to change the rights governing the file. Note that we ask for the information from the repository, not the working copy. This is because the GUID-property is not updated, if the files have been last modified locally.

```
oleg@pastaws0(~/)$ svn pg svn:pfs:GUID file:///home/oleg/repo/foo/bar.c
e994af62d0a8f3f40ccce751cba876b6
```

`e994af62d0a8f3f40ccce751cba876b6` is the GUID for *foo/bar.c* in the last revision.

We can also get the GUID of a previous revision of the file.

```
oleg@pastaws0(~/)$ svn pg -r 1 svn:pfs:GUID file:///home/oleg/repo/foo/bar.c
08286d6d36a3ccf30464d3415f45df79
```

`08286d6d36a3ccf30464d3415f45df79` is the GUID for *foo/bar.c* in revision 1.

9.1.4 Mirroring

In this test, we create a mirror repository, *repo_m*, and transfer the pesto-files from *repo* to it. Since the p2p functionality of pesto is not yet complete, we manually add the files.

Then we check out from *repo_m* and verify that the directory structure of *repo* equals *repo_m*.

```
oleg@pastaws0(~/WC)$ svn co file:///home/helge/repo_m
A    repo_m/foo
A    repo_m/foo/bar.c
A    repo_m/foo/bar.h
A    repo_m/foo/baz.c
A    repo_m/STABLE
A    repo_m/STABLE/foo
A    repo_m/STABLE/foo/bar.c
A    repo_m/STABLE/foo/bar.h
A    repo_m/STABLE/foo/baz.c
 U   repo_m
Checked out revision 3.

oleg@pastaws0(~/WC/repo_m)$ find ./ |grep -v svn
./
.//foo
.//foo/bar.c
.//foo/bar.h
.//foo/baz.c
.//STABLE
.//STABLE/foo
.//STABLE/foo/bar.c
.//STABLE/foo/bar.h
.//STABLE/foo/baz.c

oleg@pastaws0(~/WC/repo)$ find ./ |grep -v svn
./
.//foo
.//foo/bar.c
.//foo/bar.h
.//foo/baz.c
.//STABLE
.//STABLE/foo
.//STABLE/foo/bar.c
.//STABLE/foo/bar.h
.//STABLE/foo/baz.c
```

9.1.5 Decentralized subversion

Finally we see how the pfs backend performs in a decentralized setting. Note, that no consistency control mechanism has yet been implemented. Meaning that the only test we can do,

is see that the changes done to one repository are propagated to all repositories. We reuse the mirror repository from the previous test, but commit changes to it, instead of the main repository.

We will discuss how consistency control, for a decentralized subversion, based on the pfs backend can be implemented in the "Future work" chapter.

```
oleg@pastaws0(~/WC/repo_m/foo)$ echo ``bah'' > qux.dat
oleg@pastaws0(~/WC/repo_m/foo)$ svn add qux.dat
A         qux.dat
oleg@pastaws0(~/WC/repo_m/foo)$ svn ci -m ``added qux.dat''
Adding         foo/qux.dat
Transmitting file data .
Committed revision 4.

oleg@hydra(~/WC/repo)$ svn up
A    foo/qux.dat
Updated to revision 4.
oleg@hydra(~/WC/repo/foo)$ find ./
./
.//bar.c
.//bar.h
.//baz.c
.//qux.dat
oleg@hydra(~/WC/repo/foo)$ cat qux.dat
bah
```

9.2 Testing

The project has been self-hosting, eg. hosting the sources to subversion, libpesto, pfs and this document, as of April the 20th 2007. This has hopefully been enough to weed out most of the bugs from the code. Since the system does what is expected from it, and there are no known bugs, we are left to evaluate the performance of the system, relatively to the other filesystem backends. Hoping for a performance improvement in this part would be naive, pfs stores all of its data encrypted, while the original backends store the data in plaintext. More-so, pfs was not designed with performance in mind, since it is only a prototype, while the other backends have a few tricks that give them better performance. What we hope to prove in the performance tests is that pfs has acceptable performance, ie not an order of magnitude slower than the other backends, and that the performance hit can be attributed to the extra disk i/o and cpu-time used to encrypt/decrypt the files.

To test the performance we need some test data. Simply adding a few files to a repository and measuring the speed of commits and checkouts, and the size of the repository, would not give any meaningful data, because it would not reflect typical usage of a subversion repository, and the update history of objects would be lost. This is crucial to how file-data is accessed in the fsfs and bdb systems, and the size of the repository. Instead we take an already existing project, stored in a CVS-repository, or to be exact, the pesto cvs-repository. This repository

Total CVS Files:	2565
Total CVS Revisions:	20371
Total Unique Tags:	32
Total Unique Branches:	6
CVS Repos Size in KB:	173955
Total SVN Commits:	6779
First Revision Date:	Mar 7 1996
Last Revision Date:	May 24 2007

Table 9.3: Statistics for the pesto cvs repository, as reported by `cvs2svn`.

is well suited, since it is a non-trivial project running over many years, storing both text and binary data. And create a subversion-dumpfile from it, with the `cvs2svn` utility. The statistics for the cvs-repository can be seen in the table 9.3.

To test the performance of the system, we create two repositories, *pesto_pfs* and *pesto_fsfs*, using the pfs and fsfs backends respectively. The tests are as follows:

Efficiency
The first test is simply a comparison, of the space used by the different repositories.

Import
For the second test, we use the "`svnadmin load`" command to load the dumpfile into the repositories, recording the time taken to complete the operation.

Export
For the third test, we export the revisions 100, 1000, 4000 and HEAD from each of the repositories, recording the time taken. An "export" checks out the complete repository as it was at that revision.

Checkout
For the forth test, we check out the HEAD revision of the whole repository. This is different from export HEAD, since the meta-information for files has to be read and written to the working copy.

Commit (subversion source)
For the fifth test, we add the source tree of subversion to *trunk/src* in the working copy, and commit. This test is done to see how well the backends can handle a large workload when committing.

Update (get subversion)
In the sixth test we measure the time taken to retrieve the source tree, added in the previous test, this tests measures how well the backends can handle a large workload when fetching updates.

Commit (single file)
In the seventh test we add a single file, *foo.txt* to *trunk/src* in the working copy, and commit. This test shows us how well the backends can handle a small workload, which is typical for normal use of a version control system.

	fsfs	pfs
Storage efficiency	213MB	2.3GB
Import	263 seconds	10653 seconds
Export R100	0 seconds	4 seconds
Export R1000	5 seconds	51 seconds
Export R4000	26 seconds	277 seconds
Export HEAD	39 seconds	373 seconds
Checkout HEAD	73 seconds	441 seconds
Commit (subversion source)	62 seconds	821 seconds
Update (get subversion source)	31 seconds	364 seconds
Commit (single file)	3 seconds	5 seconds
Update (get single file)	4 seconds	14 seconds

Table 9.4: Results from the tests

Update (single file)
In the eight and final test we update the repository, fetching *foo.txt* from test seven. This test shows how well the repository can handle an update of a recent working copy.

All tests were performed on `pastaws0`.

Results

As we can see from table 9.2, pfs performs worse than fsfs in all aspects. It is approximately 10 times as slow as fsfs, still these results are not to bad, considering that no performance optimizations have been made in pfs. Furthermore, as we can recall fsfs stores every revision as a file, while pfs stores each file in each revision as three encrypted pesto objects; the node-object, the data-object, and the properties-object. This means that pfs has to open/close a lot of files, read the files, decrypt them, and write to temporary files. We can also see that the difference for simple operations, commit/update a single file change is quite small between pfs and fsfs, and it is this type of operations that constitute the typical use of a version control system.

We can try to improve the performance of pfs by using a memory based filesystem for the transaction in the repository in pfs. Such filesystem implementations exist for both Linux and NetBSD, respectively *tmpfs*[15] and *mfs*[14]. These filesystems are temporary, ie they are not persistent over a restart, and use the virtual memory/ram of the host instead of the disk, but otherwise behave as any normal filesystem. Since only access to transaction-objects will be improved, we just reran the tests that use transactions, ie import and commit.

As we can see in table 9.5, we have improved the time needed to commit changes to the repository, but importing a repository still takes quite some time. Using mfs does not improve checkouts/updates, but we believe that the performance can be improved for these functions to, and we therefore use the gnu-profiler[17] to find the bottlenecks in pfs [1]. The profiler works by adding a hook to the entry/exit-points of each function, and recording the time spent in

[1]The complete traces from gprof are available on the attached CD.

	pfs with mfs	pfs
Import/load	10145 seconds	10653 seconds
Commit	524 seconds	821 seconds
Commit single file	4 seconds	5 seconds

Table 9.5: Results from using pfs over mfs and writing to disk

	commit	checkout
`read()`	61.17%	79.09%
`open()`	13.69%	2.10%
`write()`	8.13%	2.58%
`decrypt()`	0.04%	8.17%
`crypt()`	1.21%	0%

Table 9.6: The most used functions when committing and checking out from a pfs repository as shown by gprof.

each function. We perform two tests, in the first test we create an empty pfs repository, and add the source-tree of subversion to it, and commit. In the second test, we check out the repository created in the first test. This way we can see which functions are most used to write data to the repository, and to read data from the repository.

As we can see in table 9.6, `read()` dominates both commit and checkout, and the cpu overhead of decrypting/encrypting is not that severe. We can probably speed up the process by keeping in-memory cache for the node and props object of the most accessed files, and probably also keep the data object in memory, at least for directories. pesto supports functions for decrypting directly to memory, so further overhead can be saved there. This way the performance of pfs can probably be brought closer to that of fsfs.

Chapter 10

Discussion

In the previous chapter we demonstrated that the pfs backend provides the same functionality as the other backends, and that the performance of pfs, although worse than that of fsfs is within acceptable limits. Performance of pfs can probably be improved beyond what it is now, by simply caching some of the data, and using more efficient storage formats, both in pfs and libpesto. Most important is it that the system scales as well as the official filesystem backend when the repository grows in size. One should also keep in mind, that with a pfs repository it is easy to set up several mirrors, so that the number of developers per repository goes down, and as a result, performance increases. In the tests, the size of pfs repository turned out to be 10 times the size of the fsfs repository, and we expect it to be even more for repositories that have more file-updates. Delta compression could always be put back into pfs, but we feel that the flexibility of storing the complete file for each version, outweighs the added costs for disk-space. Disk space is, at the time of writing, very cheap, and the cost per gigabyte will probably continue to fall. A 0.5TB SATA disk can now be had for as little as 1000NOK. What about devices with limited storage space, like laptops? Adding several TB of storage to developer-laptops does not seem as a reasonable investment for a software company to allow developers to work from their cottage in the mountains. Thankfully, pfs does not require that. Typically users of a version-control system will only be interested in the most recent revisions of a file. pesto could easily be configured to only keep the leafs, and a few levels upstream of the leaf nodes in the version tree on machines with limited storage, or it could start by removing the oldest versions of files with many versions, when the device starts running out of space. Should an operation require these files, pfs can be easily changed to try to retrieve the files from a backup-server. if an operation should require them.

In the previous chapter we have also demonstrated the new functionality of subversion when using a pfs backend. We have shown that pfs can be used to automate backups of a subversion repository. A script that takes backups of a bdb-based repository will have no choice, but to copy the complete repository, each time a backup is made. With the fsfs repository, it is possible to be more clever about how the backup is done, by only copying the newest revision, and revision properties files. Still, it is possible to create a repository with an inconsistent state. With pfs, only the changed files and directories will be replicated. And they will only be replicated when they are created. Pesto can also be used to set up more intelligent replication policies, by giving another pesto-node responsibility for replicating the data. The repository then sends the updated files to that node, which then distributes it to multiple backup-nodes.

If we give some of the backup-nodes read access to the data, and install subversion/pfs on those, it is possible to use the nodes as a backup-mirror. Pesto can be configured to prohibit these nodes from making new versions of the stored files. By doing this we can avoid a situation where one of the backup mirrors has been compromised, and is used to corrupt the whole repository. Like in the backup scenario, polling is not necessary, when using pfs, since updates will be pushed to mirrors automatically. Several pesto-nodes can get the responsibility of distributing the updates to the mirrors, thereby offloading the main repository. These nodes can even be organized as a tree, to further improve content distribution.

Files stored in a pfs repository can be accessed by other means than via the subversion client. By getting the `svn:pfs:GUID` property of a file, it is as simple as giving another pesto-node read access to the file with that GUID. The file will then be automatically sent to that node, and its owner can read the file.

Finally, we have shown that pfs can also be used in a decentralized setting, with one exception, updates that are made to an offline node, can "disappear", if other updates are made at the same time. This is clearly not good enough for use in a busy environment with many offline users. We hope, that in the future, some sort of consistency control can be added to pfs, which prevents this. However, a decentralized setting can be useful for a private repository used by a single developer. The user could have one repository on his/hers main workstation, one on hers/his laptop, and one at home, and storage/distribution nodes on the company's sever(s). The user can then access the repository from his/hers laptop, or at home, although the workstation on which the latest changes were made, has gone offline. As long as a chain of machines that can download and store the files can be constructed, so that at least two machines at a time are online long enough to transfer the changes, the user gets the updates. The user is assumed to know, that he/she changed the some file previously, and not edit it, unless it is updated to the newest revision. However, should an accident happen, and the user overwrites a file, or several files, it is a simple matter of getting the GUID of the current version of the file, and use a pesto-client to view the different versions of the file, and fetch the correct version, which will have the revision key-value-pair set to the previous revision, and get the file outside of pesto, and merge the changes back into the repository.

Chapter 11

Future work

11.1 Performance improvements

The performance of pfs can be improved. Currently each time an immutable subversion node-revision is read, it is first read from pesto storage, decrypted, and written to a file, and then desereialized. Mutable node-revisions, are not read from pesto, but have to be read from the transaction-directory and parsed. This process is done each time the node is accessed. Likewise, the revision anchor has to be read, decrypted and deserialized, and the transaction anchor has to be deserialized.

This creates a severe io overhead, results from gprof [17], shown in table 9.6, show that most of the time is spent read()'ing from disk, both for commit and checkout. This hurts performance, and should be avoided.

11.1.1 Caching

To solve this problem, we suggest that a multilevel cache is implemented.

The lowest cache-level is the *pesto-cache*, which caches pesto-files in plain-text. Each time a pesto-file is accessed with a GUID, the cache checks if a decrypted version of the file exists, and returns it, if it finds one. If not, it fetches the file from pesto, decrypts it, stores it in the cache, and returns it. This cache saves io and cpu-time, since it is unnecessary to read files from pesto and decrypt them. The pesto-cache could be implemented as part of libpesto, or be a part of pfs, as a separate cache-module, and should store files to disk.

The middle cache-level is the *noderev-cache*, which caches subversion node-revisions, that have already been deserialized in memory, as `vault_noderev` objects. If the accessed node-revision is not present, it is fetched from the pesto-cache, if it is part of a revision, ie immutable, and stored in pesto, or it should be read from the transaction-directory, if it is a transaction node. When a transaction-node is updated, it should be marked as dirty/flushed from the cache. The noderev-cache should be implemented as part of the `pfs_vault` module.

The upper cache-level, the *directory-cache*, caches directory data. When the directory-contents for a given path is requested from pfs, the directory-cache should be checked first, and return the cached directory contents, or fetch the directory-contents from either the transaction-directory, or the *pesto-cache*, depending on whether the node-revision is committed, or in a

transaction. When the directory-contents of a node-revision in a transaction are updated, it can either be flushed from the cache, or the cache contents could be updated too. The directory-cache should store its data in memory, and be implemented as part of the file-layer of pfs.

11.1.2 Ramfs

Another performance hack, that can be used in conjunction, or as replacement for caching, is to write transaction-data, and temp-files to a ram-based filesystem. Both linux and NetBSD support ram-based filesystems, *tmpfs*[15] and *mfs*[14] respectively. As we have seen in the "testing" chapter, mfs does improve the performance of pfs. What must be kept in mind though, is that the size of all ongoing transactions is limited by the virtual-memory available at the server.

11.2 User-friendly features

pfs was implemented in a limited timeframe, and therefore lacks some "polish" to be usable by non-developers.

11.2.1 Error handling

The standard-way for subversion to present users with error-messages, is to create a string describing the error, and return it to the user. This depends on how the users accesses the repository, ie, if `svnserve` encounters an error, it returns an error over the network to the client, or it the apache web-dav module encounters the error, the error-message is given to the user by the web-server.

If an error occurs in pfs, an `assert` condition is triggered, and a brief message describing which line the error occurred on is printed to standard out, and the program dumps core, and terminates. This is more useful for developers, since the exact location of the error, and the condition under which the error occurred, including the stack-trace and core of the program is preserved. However, this is not the best way to present users with errors, firstly because errors wont be transmitted when the repository is accessed remotely, which would be the common-case for normal use, and secondly because non-harmful errors, ie user-errors, like accessing a non-existent revision will be viewed as a critical-program failure, and not as a user mistake.

Errors should be classified into groups, and propper error-messages should be returned. A compile-time switch should be added to enable asserts instead of error-messages for debugging.

11.2.2 Installation

The current way of building subversion/pfs is a hack to subversions original build-scripts. A way of enabling/disabling libpesto and specifying the location of libpesto from the configuration script has to be added. Installing subversion/pfs is at this time quite complicated, as can be gathered from the installation guide (Appendix B).

11.2.3 Repository creation

When the repository is created, with `svnadmin create`, it automatically creates a new backing-store for pesto, and initializes a new node. This is not always what the user wants, and to change the type of repository, the user has to manually create symlinks to his/hers pesto backing-store, setup keys, node-ids and write the guid of the root-node. All this should be controllable with parameters to `svnadmin create`. Unfortunately, subversion does not have any mechanisms for passing parameters to the creation routine of the filesystem, so mechanisms for this have to be added. Alternatively an interactive script could be created, to automate the repository creation process.

11.2.4 Policy interface

Currently, to change a policy of a file in pfs the user has to get the `svn:pfs:GUID` property of the file from the repository and use an external tool to change the policy. Instead, properties could be set on files, and their policies changed during the commit. But before doing that, we have to consider a few pros and cons. This approach is more user-friendly for the user, since policies are handled transparently. However this means that there is no single interface for changing policies for files stored in pesto. It also means that the user gets no feedback on the policy change before the file is committed.

11.2.5 Revision dependencies

When new pesto objects are created, they are usually distributed in a first in, first out, order. Connection problems, or other transmission errors could disturb this order, and create a situation where a node gets objects out of order.

pfs does not see a revision, before its revision-anchor is accessible. During a commit, pfs writes the revision-anchor last, so that it has the most chance of being received last. However, this is not always the case. If a client tries to check out the revision, while some of the files in the revision have not yet been received by the repository, the checkout will fail, missing one or more files.

To solve this, each revision-anchor should carry with it a list of GUIDs that were created in that revision. Before checking out a revision, pfs should verify that all files needed by a revision, and implicitly all files needed by previous revisions, are present. If not, it should give a warning to the user that the operation might fail. The user could then either wait, for the problem to resolve itself, or if impatient, use the `--force` switch to try to complete the operation, ignoring the missing files.

11.3 Distributed/decentralized subversion

Last, but not least, the distributed/decentralized subversion is still far from complete. Currently a pfs repository can be used for mirroring, with a single commit repository without problems. Having multiple repositories that allow commits, is however more problematic. Firstly, subversion locks are only supported per repository, so a locked file/directory, only

grants exclusive access to that file/directory for that repository. Secondly, all sorts of consistency problems creep up with multiple commits, since updates committed to one of the repositories, are not necessarily distributed to the other repositories before a commit at the other repositories. While this most certainly can be useful to a developer committing to his/hers laptop while traveling, and then syncing the repository with a **private** repository at work/home, this is not very useful for multiple developers.

To allow multiple developers to use a decentralized subversion, some sort of consistency control has to be implemented. Since we do not want to limit ourselves to nodes that are always online, we have to assume that nodes can be offline for prolonged time. Ie, we cannot simply say that all nodes have to be online for a commit to take place. We also state that all of the repositories/nodes are equal, ie no single master repository/node.

11.3.1 Consistency control

What happens during disconnected operations, is that a node is unaware of any commits done to the repository. This is not a problem for checkouts done from the disconnected node. The users that have checked out data from it, will simply lag behind, until they check out from an updated node. However users should still be allowed to commit to the disconnected node. Suppose that two pesto nodes, A and B are used to access the same subversion repository. The users a and b are using nodes A and B respectively. Suppose that both nodes are updated to revision R. Then they become disconnected. User a commits, and A gets to the revision R_A. b commits and B gets to the revision R_B. The revision R has now been *forked*.

Currently pfs chooses one of the committed revisions to be the official revision when A and B become connected again, discarding the changes done in the other. To avoid this, updates done to the offline repository should be merged with the updates done to the connected repositories.

To solve this problem, two approaches could be taken. One could only allow one user/node to commit to the repository at a time (pessimistic consistency control), using some sort of distributed locking mechanism. The problem with this approach, is that it would be impossible to commit to an offline/disconnected node. Distributed revision control systems are not used to offload the central repository because of many commits. A scenario that big would involve so many developers, that project management would be of greater concern, than revision control. Instead distributed revision control is used to allow developers to work with the code when they are traveling or otherwise offline, or have bad connectivity. Pessimistic consistency control, is therefor not the way to go. An optimistic consistency control mechanism should be used instead. Users should be allowed to commit to the repository, not knowing for sure if the commit can cause inconsistencies or not. When an inconsistency happens, ie two users commit at the same time, and create two different revision anchors for the same revision, it should be resolved automatically, if possible. It is possible to merge the namespace [5] more or less automatically, and subversion can merge files automatically. This can be used to create merged files in the users working directory when the user checks out a "forked" revision. Subversion should also forbid the user to commit before the conflicts have been resolved manually.

11.3.2 Branching à-la git

An alternative to consistency control, is to let each node have its own repository, like it is done in GIT [7] and DARCS [10]. A pesto-node can then be used by a group of developers, or even as the private repository of a single developer. The updates created at other nodes, can be shown in a separate directory, *"/others"* in the root of the repository. Users can then create their own branches, which will show up under their name in the *"/others"* directory. Pesto can be used to specify which nodes to send and receive updates from, to avoid having to many directories in the *"/others"* directory. Subversion supports merging of data from two directories in the same repository. Developers could therefore merge branches from the *"/others"* directory into their own branch.

To implement this some changes have to be made to the filesystem backend. Instead of choosing one of the revision-anchors at random, when there are several revision-anchors for a revision, subversion can instead look at the creator of the revision-anchor. If the creator is the local node, then the root of that revision, is used as *"/"*. If not, it is added under the *"/others"* directory. Since the backing pesto-files for the subversion-nodes that are the same across "branches" are the same, only the changed files have to be exchanged.

The user can then use his/hers personal repository when disconnected, and when connectivity is again reestablished, the users branch is updated on all other connected repositories automatically. Some sort of message could probably also be added, to notify interested parties that new updates are available.

Chapter 12

Conclusion

We started this project with the task of creating a version-control system using pesto. We specified the requirements for such a system in the *"Requirements"* chapter. Then we modified the already existing storage backend for subversion, fsfs to create a new backend, pfs, which we designed and implemented in the *"Design"* and *"Implementation"* chapters. In the *"Testing"* chapter we evaluated the functionality and performance of the system, and concluded that the system had all necessary functionality, and that the performance was acceptable, but could be improved by utilizing caching and more efficient storage formats. In the *"Future work"* chapter we outline what should be done next with pfs, and how we belive it should be implemented.

To summarize; we have shown that it is indeed feasible to use libpesto as the basis for a version-control system. We have shown that, once the peer-to-peer functionality of pesto is fully in place, the pfs backend can be used to create a decentralized version-control system.

We also been able to test how libpesto performs as the backend for a real-world application. Originally files in pesto were only meant to have a few successors. The pfs backend stores each file-update and revision as the successor of the previous file-update or revision. This means that the number of successors to a file/revision can quickly grow to thousands. This uncovered some serious performance bottlenecks and bugs in libpesto which have now been fixed.

Bibliography

[1] Disconnected Operation In The Pesto Storage System
Doctorate dissertation
Feike W. Dillema

[2] The Taste of Pesto
Feike W. Dillema, Tage Stabel-Kulø

[3] Version Control with Subversion
C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick
Publisher: O'Reilly
ISBN 0-596-00448-6
http://svnbook.red-bean.com

[4] Pesto Flavoured Security
Feike W. Dillema, Tage Stabel-Kulø

[5] Conflict Resolution for User-Selected Names in Collaborative Systems
Simone Lupetti, Feike W. Dillema and Tage Stabell-Kulø
The 11th International Conference on CSCW in Design
April 26 -28, 2007, Melbourne, Australia

[6] Consistency in Partitioned Networks: a survey
Susan B. Davidson, Hector Garcia-Molina, Dale Skeen
Published in:
Computing Surveys, Vol. 13, No. 3, September 1986

[7] Collaborating With Git
John Loeliger
Linux Magazine June 2006
http://www.jdl.com/papers/Collaborating_Using_Git.pdf

[8] Pragmatic Version Control Using CVS
Dave Thomas, Andy Hunt
Publisher: The Pragmatic Programmers
ISBN 0-974-51400-4

[9] RCS — System for Version Control
Walter F. Tichy
Purdue University
Department of Computer Sciences
West Lafayette, Indiana 47907
http://agave.garden.org/ aaronh/rcs/tichy1985rcs/html/index.html

[10] darcs: distributed source code management system
Project homepage: http://www.darc.net

[11] The SVK version control system
http://svk.bestpractical.com/view/HomePage

[12] SVK visual guide
Russel Brown
http://pickscrape.woobling.org/svk-visual-guide.pdf

[13] Proceedings of the FREENIX Track:
1999 USENIX Annual Technical Conference
Berkeley DB
Michael A. Olson, Keith Bostic, and Margo Seltzer

[14] A Pageable Memory Based Filesystem
Marshall Kirk McKusick, Michael J. Karels, Keith Bostic
http://fmg-www.cs.ucla.edu/classes/239_2.fall98/papers/pageable.html

[15] tmpfs: A Virtual Memory Filesystem
Peter Snyder
http://www.solarisinternals.com/si/reading/tmpfs.pdf

[16] The Pesto Library
Feike W. Dillema
Api-specifications for pesto, can be found on the attached CD.

[17] gprof, the GNU profiler
http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

[18] The NetBSD project
Project homepage: http://www.netbsd.org

[19] Unix inodes
http://www.bellevuelinux.org/inode.html

Appendix A

The CDROM

The attached CDROM contains all material related to the work. This includes:

- The thesis itself in PostScript and PDF formats.
- The api-specification for pesto can be found in the `pesto_api` directory.
- The modified source-code for subversion, in the `src` directory.
- The changes to subversion in the `changes` directory. This is the source to the `libsvn_fs_pfs` module itself, and the changes done to the build-scripts and other source-files to include it in subversion.
- The required libraries to compile and run the project are in the `required` directory. The directory contains:
  - libpesto.
  - modified c-web package to compile libpesto.
  - The apache protability runtime, version 1.28.
  - Subversion, version 1.4.2
  - Apr-utilities, needed to compile apr.

Appendix B

Installation instructions

We only provide installation instructions for Linux and NetBSD.

B.1 Building libpesto

Before subversion can be installed, libpesto has to be built. libpesto is written using *literate programming*, and literate-programming tools are needed to build it. libpesto uses a modified version of cweb, which is included in the *required/cwebx* directory.

B.1.1 Building cwebx

cwebx can be built by simply running bsd or gnu make in the cwebx directory. Only `ctangle` is required to build libpesto, and any errors building `cweave` can be ignored.

B.1.2 Building libpesto

To build libpesto the makefile in the libpesto directory has to be changed, so that the `CTANGLE` parameter is set to the correct path to ctangle. Then it is as simple as typing `make` to build libpesto. Some of the test programs can fail to compile, but it is enough to verify that `libpesto.a` has been created.

B.2 Building subversion

B.2.1 Required dependencies

To install subversion the Apache Portability Runtime (APR) has to be installed, we recommend that it is installed from the package management system of the operating system, if not it can be downloaded from http://apr.apache.org. APR is dependent on expat, which can be downloaded from http://sf.net/projects/expat.

B.2.2 Building subversion with pfs

After all of subversions dependencies have been installed, we can install subversion. First we edit *"build.conf"* file, located in the root-directory of the modified subversion sources. Look for a section like this:

```
# OLEG
[lpesto]
type = lib
external-lib = -L/full/path/to/libpesto -lpesto
```

And change external-lib to the **full path** where libpesto sources and the `libpesto.a` library are located.

Then we edit *"Makefile.in"* in the root-directory of the modified subversion sources. Look for a section like this:

```
# OLEG
INCLUDES = -I$(top_srcdir)/subversion/include -I$(top_builddir)/subversion\
           -I../path/to/libpesto\
           @SVN_NEON_INCLUDES@ \
           @SVN_APR_INCLUDES@ @SVN_APRUTIL_INCLUDES@ @SVN_SERF_INCLUDES@
```

And change `../path/to/libpesto` to the path to libpesto, a full or relative path can be used here.

Then the following commands should be run to install subversion from the subversion-source directory.

```
./autogen.sh
./configure
make
make install
```

Optionally, the `--prefix=INSTALL_DIR` argument can be given to the configure script, to specify where subversion should be installed.

Appendix C

Usage

To get proper help on subversion, pleas use the "help" command for each subversion utility, and read the subversion-book[3].

C.1 Creating a repository

C.1.1 Creating a repository

To create a subversion/pfs repository with the name "NAME" this command should be used:

```
svnadmin create --fs-type pfs NAME
```

C.1.2 Creating a mirror repository

To create a mirror repository for an existing repository with the pesto-backing store in */pesto* do the following: first create a new subversion/pfs repository, *repo*:

```
svnadmin create --fs-type pfs repo
```

Then link *FAMS*, *LEAF*, *TMP*, *DATA* and *META* directories to the directories in the backing store.

```
(~)$ cd repo/db
(~/repo/db)$ rm -rf DATA FAMS LEAF META TMP
(~/repo/db)$ ln -s ~/pesto/{DATA,FAMS,LEAF,META,TMP} .
```

Setup the ownerid, ownerkey and shareid correctly:

```
(~/repo/db)$ echo ($SHAREID) > ps
(~/repo/db)$ echo ($OWNERID, $OWNERKEY) > pv
```

And write the correct identifier of the repository to the uuid and root files:

```
(~/repo/db)$ echo $ROOTID > uuid
(~/repo/db)$ echo ($ROOTID) > root
```

C.2 Normal use

C.2.1 Checking out a repository

Checking out a subversion/pfs repository is no different from checking out a normal subversion repository. We show an example using the *"file"* an *"svn+ssh"* protocols.

Using the file protocol:

```
(~)$ svn checkout file:///path/to/repo
```

Using the svn+sah protocol:

```
(~)$ svn checkout svn+ssh://host.holding.the.repo/full/path/to/repo
```

C.2.2 Updating a repository

Updating a subversion/pfs repository is no different from updating a standard subversion repository:

```
(~/repo)$ svn update
```

C.2.3 Committing changes to the repository

Again, committing changes to a pfs repository is no different from committing to a standard repository.

```
(~/repo)$ svn commit
```

C.2.4 Getting the log history of a file

Like in normal subversion, we can get the log history for a file/directory.

To get log of a directory:

```
(~/repo/dir)$ svn log .
```

To get log of a file:

```
(~/repo/dir)$ svn log file.dat
```

C.2.5 Get changes between two revisions for a file

We can get the delta between a revision, and the current working copy.

```
(~/repo/dir)$ svn diff -r 2 file.dat
```

Or we can get the delta between two revisions of a file in the repository:

```
(~)$ svn diff -r 1:2 file:///path/to/repo/dir/file.dat
```

C.3 Get the GUID of a file stored in an pfs repository

To get the file of a file in a pfs repository we have to get the `svn:pfs:GUID` property. Note that we have to get the property directly from the repository, since it might not be up-to-date in the working copy.

```
(˜)$ svn pg svn:pfs:GUID file:///path/to/repo/dir/file.dat
```

We can even get the guid for a particular revision of the file by giving the revision switch (`-r`):

```
(˜)$ svn -r 1 pg svn:pfs:GUID file:///path/to/repo/dir/file.dat
```