West University of Timișoara
Faculty of Mathematics and Informatics
Department of Computer Science

Gitfs - Version controlled file system

Vlad Temian

Supervisor *dr. Marian Neagul*

Submitted in part fulfilment of the requirements for the degree of
Bachelor Degree in Computer Science, July 2016

Abstract

Gitfs is a version controlled file system. Its purpose is to help non-technical users in creating versioned content without having any knowledge about version control systems.

By using this file system, any user can bring changes to the content of their project, or alter it in any way, and Gitfs will automatically version and share these modifications among all project collaborators, solving any conflicts that emerge. More so, it also shows a history of the content, by simulating snapshots for each and every change.

Gitfs implements a transparent concurrency models and manipulates the content directly, without using another layer. In this way, the user's change will be versioned in without to notice about it. Beside that, using a multi-layered cache system, Gitfs will not add any performance overhead for the final user.

Abstract

Gitfs este un sistem de fișiere ce oferă posibilitatea de versionare. Scopul său este de a ajuta utilizatori non-tehnici în crearea de conținut versionat chiar și în lipsa cunoștințelor despre sistemele de versionare.

Cu ajutorul său, orice utilizator poate aduce modificări de orice tip asupra conținutului proiectului său și Gitfs va crea o versiune unică pentru fiecare din aceste modificări, împărțind aceste versiuni cu toți colaboratorii proiectului. Chiar mai mult, sistemul va salva și expune o istorie a conținutului, prin o acțiune asemănătoare cu capturarea unor imagini, pentru fiecare modificare a conținutului.

Acknowledgements

This project was build under the employment of Presslabs, originally along with Calin Don, Emanuel Danci and Mile Rosu. Presslabs is a web hosting company, which do Wordpress hosting dedicated for publishers. Special thanks to all people involved in the project by fixing issues or giving feedback.

Many thanks to Marian Neagul, my supervisor, who had the patience to work with me and offered his support.

Contents

Chapter 1

Introduction

Gitfs is a version controlled FUSE [RG10] file system, based on git [Spi12]. It is an open source project developed under the Apache License [Sin10]. This file system, which can manage unversioned content in a versioned manner, was developed with the purpose of making life easier for the non-technical user. Beside managing the content, Gitfs also help non-technical users to collaborate by sharing the content among them and solving any conflicts.

1.1 Problem Statement

For many years, creating versioned content for the basic and non-technical user has been painful. For people working in fields such as professional writing, photography and so on, sharing files with different people or even working on them in a distributed manner, was almost an impossible task even with technical background.

These days, in order to make versioned content, you need to be very disciplined and you need to save a different version of the file in a different location, using unique names for each and every change. In order to distribute the content, you need to use a file sharing application (for example Dropbox, Google Drive or Amazon S3), but these come without any merging or reconciliation mechanisms.

If two different people want to edit the same file, one option would be that they distribute it on a common storage device (physical or cloud related). The downside of this solution is that the only file version which will be visible on the storage device will be just the one that was last distributed.

Another way to approach this problem would be by using external online tools, such

as Google Docs or Adobe Creative Cloud, in order to keep track and sync your work. But these also come at a price in both time and money. Most of these are commercial only, painful to setup and only support certain types of content.

1.2 Solution

The easiest way to solve this problem is to let the file system do all the work.

In order to better explain how this works I will give you a more concrete example of our solution. We will be using two different directories: one for the current representation of the content (which will be named 'current') and one for different representation of the content at certain points in time in chronological order (we'll call it 'history'). Each write operation will be transformed in a version of the file, adding some information about the change (who made the change, when was the change made and a short message describing what was changed).

After all write operations are finished, the file system can push all of these changes to a remote location. From that location, other file system instances can pull those changes at a fixed or variant period of time. In case of any conflicts, the file system will apply a common merging strategy, like always accepting local changes. By using this strategy, the user has the guarantee that the conflict or conflicts will be, eventually, solved.

1.2.1 Project description

We can split the building blocks of the solution into two main parts: one part which can handle all the versioning problems and another part which will allow us to easily built a fully flagged file system with good concurrency support.

For the versioning part we have adopted git's philosophy. All write operations are grouped and combined together in a single unit of work called commit. Besides content, a commit also contains some metadata like who created the content, when was the content created or when were the changes made and a short message describing these changes. In order to be very efficient and to be able to manipulate the git objects at the lowest level possible, all this while using a small memory footprint, we've chosen libgit2 [Libb], a library that implements all of git's internal representation and its versioned content managing systems. Libgit2 has bindings in all modern high level languages, including Python. So, to have a simple, easy and maintainable implementation, we used pygit2 [Pyg], which provides all the bindings from libgit2 in Python.

For the file system part we didn't want to write any kernel code and we wanted to use a high level language, with decent concurrency support. These are the reasons for which we have choosen FUSE [RG10]. FUSE stands for 'file system in userspace' which is an interface that lets you build a filesytem, withtout any kernel hacking. This interface is implemented by libfuse [Liba], a library written in C that has bindings in any of the major high level languages. The Python implementation of those bindings is called fusepy.

In order to solve any raising conflicts and group all the changes together we need a solid concurrency model. On one side we have the FUSE file system which can spawn multiple threads, one for each of its operations (basically for each system call). We will call these dummy workers. All the write operations need to be registered somewhere and aggregated in a single commit. You don't want to have a commit for each write operation, because when you extract a big archive, containing a lot of files, in the current directory, you can get tons of commits, for every single file. In order to aggregate all the changes, we'll be using a queue shared among all those FUSE threads and a special worker, called the SyncWorker, who has the job to synchronize the local changes with the remote ones. Another important part of the synchronization is to periodically get the changes form the remote repository. This will be achieved by using the FetchWorker, whose job is to bring the changes locally and to notify the SyncWorker in case there are any conflicts that need to be solved.

By combining those parts we ended up with Gitfs, a fully complaint POSIX file system [Lew91] that behaves like a wrapper over a git repository.

1.2.2 General usage

In order to use it, you will need a UNIX like operating system (Linux, FreeBSD, macOS etc), a bare git repository (there are lots of 3rd party services like Github and Bitbucket which offer free git repository, or you can even create one using the git commands) and Gitfs which can be installed from source or from one of our repositories.

After the filesytem has been mounted, using the mount command or by running the binary (see Usage for more details), two directories will be found: current and history. You can now open any content editor (for text, music, graphic or code related) and start to edit your work. Any changes made in the current directory, will be found in history directory, into a new sub-directory (named using a chronological convention: year-month-day-hour:minute). All the changes, across multiple Gitfs or git instances, are automatically synced in the background and any conflict is solved without manual intervention.

These features make Gitfs an ideal tool for the non-technical users to create distributed versioned content.

1.2.3 Presslabs specific usecase

Presslabs is a web hosting company, focused on Wordpress hosting for both firms and independent publishers. A large number of people, who use Presslabs' hosting services, are either non technical or they don't have a development team which familiar with versioning control systems. So, in the cases of these users, a major problem was to manage the code in a versioned manner. This could have lead to various complications and difficult ways to solve the issues during critical situations. For example, when a user has made a code change on the live, running website, and has introduced a fatal error. It may not have been so obvious to restore the website at a certain point in time. You would have needed to heavily rely on code backups and fast restores.

In order to simplify the process, we have created an environment in which any code changes are versioned, using git. It is composed by a central server with git repositories (highly available, with replications using mirroring, in two separated datacenters), Gitfs for SFTP (Secure File Transfer Protocol) server and each website with a Wordpress [Wor] plugin that tracks and manages any changes.

When a change occurs on sftp server, Gitfs will track it and push it to the git server (named git.presslabs.net). The server will notify the corresponding website associated to the change, so the website knows that it needs to update the code. This process can work both ways, from SFTP to website and from website to SFTP. When somebody makes a change to a website (installs a plugin, theme or edit a static file), the plugin will track the changes, push them to git.presslabs.net and from there, Gitfs will send them to the sftp server. In case of any conflicts Gitfs and Gitium will use 'always-accept-local-changes' strategy, in order to solve them. More details about this strategy can be found later in this paper.

The advantages of using such a setup is that a developer, who knows how to use git, can very easily collaborate with non-technical sites administrators. Using git, anybody, with SSH (Secure Shell) access to the git.presslabs.net server, can push git commits and these changes will end up on SFTP and the given website.

Chapter 2

Theoretical background

In the next part of the paper we will talk a bit more about the technologies and the concepts used in the making process, in order to offer a better understanding of the project. We'll start by presenting the building parts of Gitfs, from file system to git internals, and end up also discussing about similar projects.

2.1 File systems

A file system is an abstraction used by the operating systems in order to keep track of files on a storage device. In a more concrete form, you can see the file systems as a collection of data structures and algorithms which helps the operating system to manage data from a storage device. The operating system provides an API, which other programs can use to store, retrieve or delete data, from a storage device.

The operation of associating a file system to a storage device is called 'mounting'. In UNIX based operating systems, you can use the 'mount' command, which will attach a file system to the current file system hierarchy. Using the 'mount' command, you can specify a file system, its type, specific options used by the file system and a mounting point.

2.1.1 High level architecture

The actual structure of a file system can vary based on the implementation, but the majority of UNIX file systems, share a common one. It is composed by two main parts: user space and kernel space.

User space contains the applications (for this example, the user of the file system) and the GNU C Library (glibc), which provides the user interface for the file system calls (open, read, write, close). The system call interface acts as a switch, funneling system calls from user space to the appropriate endpoints in kernel space.

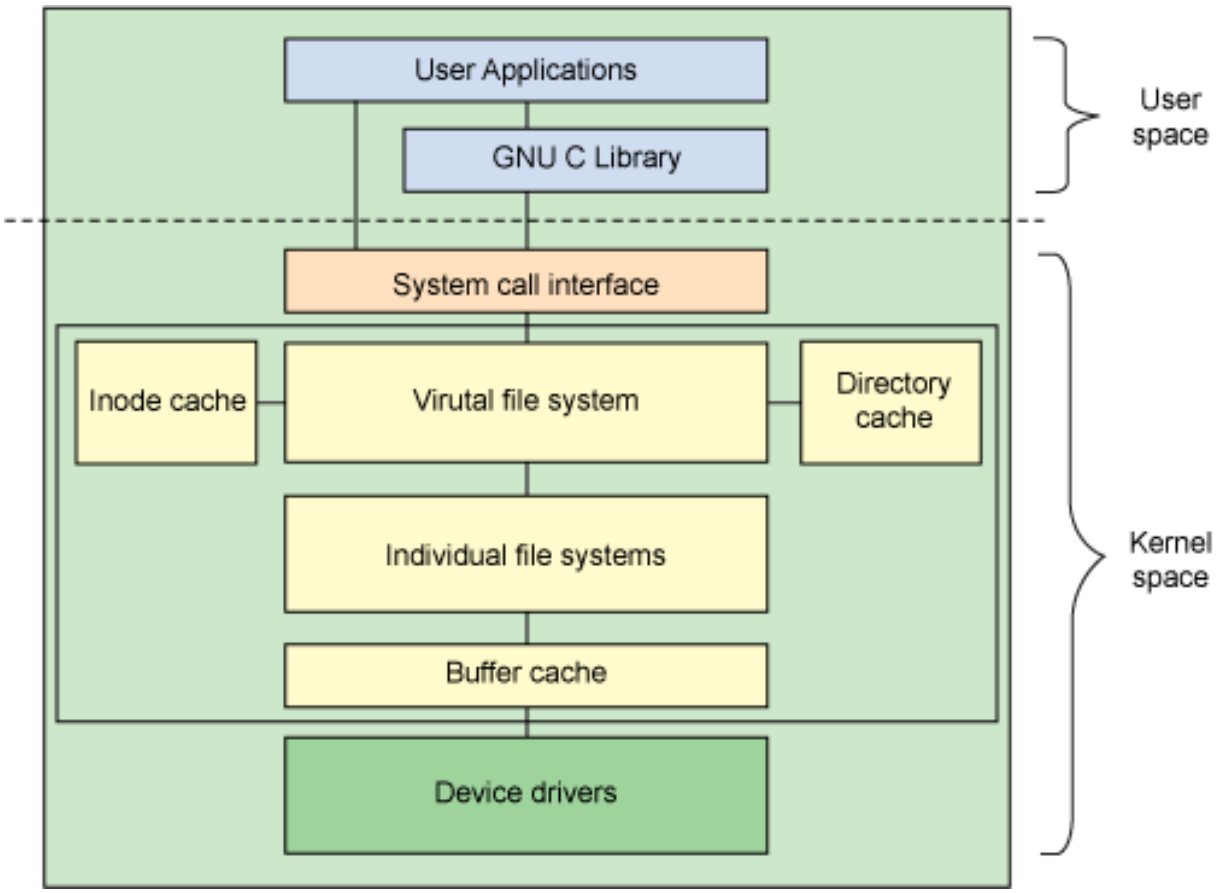

Figure 2.1: High level architecture of a file system [FSA]

As you can see in figure 2.1, an important part of a file system architecture stays in the kernel space. There, the Virtual File System (VFS) behaves as an interface for other file systems. It behaves like a gateway between system call interface and individual file systems, with some caches. It does that by exposing a series of interfaces to individual file systems, optimizing using two caches: dentries (directory cache) and inode. Each one of those caches stores the recently-used objects (implementing a LRU cache mechanism).

A kernel space implemented file system won't interact with a device directly. All write and read operations are passed through a cache, called a buffer cache. This is a very smart mechanism that offers a big performance boost, but with a cost. If you use the system call interface in order to write something in a file, that write is not atomic, meaning that if you close the file, the bits that you are writing are not on the storage device, but in this buffer cache. The kernel will flush periodically this cache on the device, but

also can control the flushing using fsync system call, after you close the file. The buffer cache will use a LRU (Least Recently Used) cache which means that is a finite small cache that will store the most recent data, being optimized for use cases when you use the same data.

Linux uses multiple types of data structure to manage the entire file system:

- superblock: the root of file system, which maintains and describes state in the file system. It contains all the information needed by the file system during certain operations, such as: file system name (e.g.: ext2, ntfs, fat32 etc.), the size of the file system and its state, a reference to a block device and metadata. It is usually stored on the storage device but it can also be created at runtime. Besides that information, a superblock also contains a set of operations. This structure is used in order to manage the inodes within the file system.
- inode: core representation in a file system. Every file or directory is represented as an inode. It contains all the meta data needed by the file systems (including which operations are possible on it) to manage a file. Individual file systems will provide different functions and mechanisms in order to generate a unique inode identifier from a filename and later into an inode reference.
- dentry: data structure used to map the names of files to inodes. It uses a directory cache to keep the most-recently used data closely. The operating system also uses dentries to travers the file system.
- file: or VHS file is just an open file. It keeps the state for a given open file, such at write offset.

In order to dynamically add or remove a file system from Linux, a set of registration functions needs to be used. You can view a list of supported file systems, accessing the /proc file system, from user space. It's not a file on disk, but a virtual file from which you can find out which devices are currently associated with the file systems. If you want to add a new file system to Linux, you need to call register_file system. The function takes only one argument which defines the reference to a file system (file_system_type) using the name of the file system, two superblock functions and a set of attributes. Once a file system is registered it can also be unregistered.

2.1.2 User space

Besides programming an individual file system in the kernel space you can use FUSE. FUSE stands for file system in user space and it is an interface for programs which helps

you export a file system to the Linux kernel. It has two main components: libfuse and fuse kernel module. Libfuse is a C library which allows you to communicate with the FUSE kernel module.

After the FUSE file system is mounted, all system calls which target that mountpoint will be redirected to the FUSE kernel module, acting like a proxy.

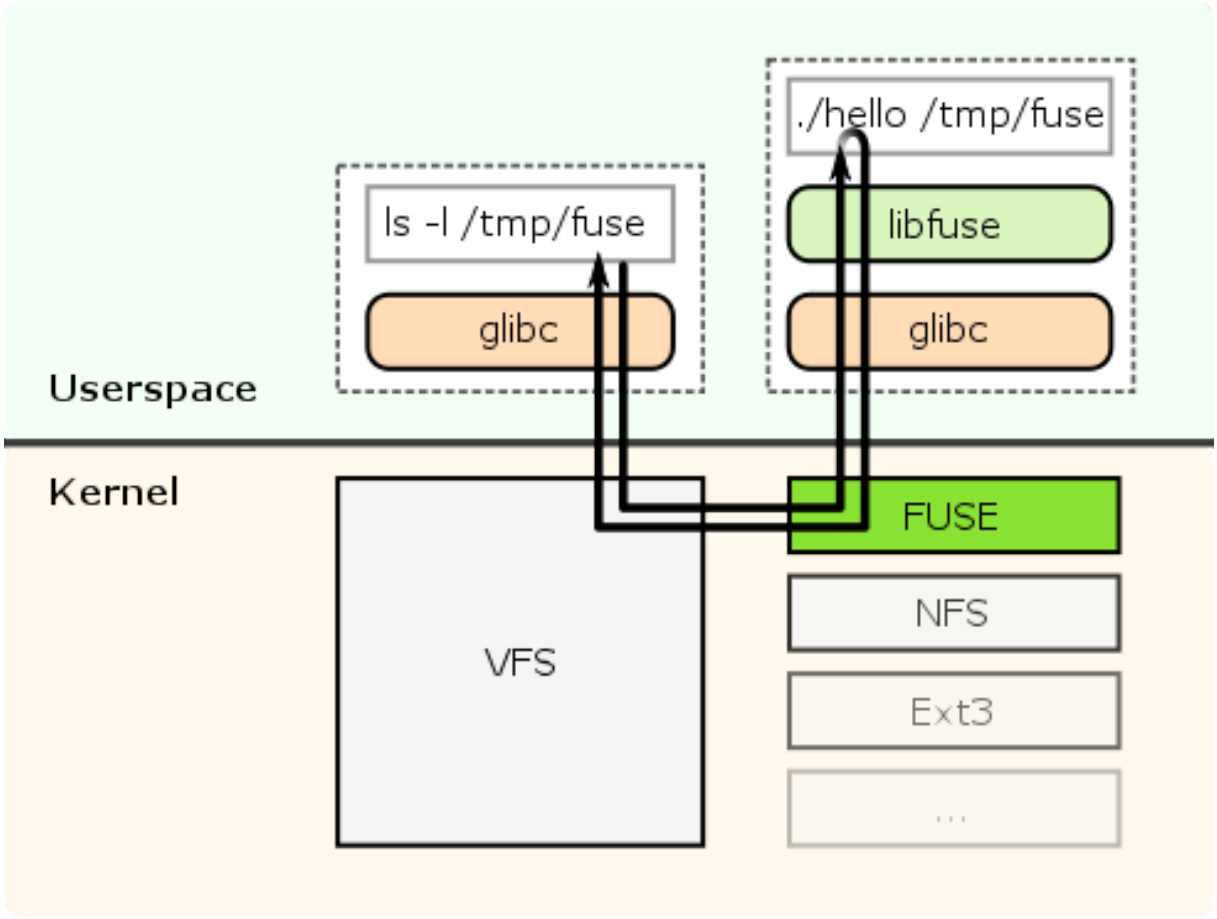

Figure 2.2: System call flow in FUSE [Fus]

Let's take read operation as an example, detalied in 2.2. Let's imagine that the directory /fuse is the FUSE mountpoint. When an application will call the read() syscall with /fuse/file as the path argument, the Virtual File System will trigger the corresponding handler in fusefs. A cache is used in order to optimize the operations flow. If the corresponding data is found in the cache it will be returned. Otherwise, the read() syscall will be forwarded to the libfuse library, which will call the callback defined in userfs for the read() syscall. The callback can make any action and return any data in the supplied buffer. With this mechanism one can implement any desired logic. For example, you can access Facebook's API in order to retrieve some photos, or to get data from an external storage device, or to do some processing from a local file system. After you've finished your work, the result is propagated to libfuse, through kernel and to the ac-

tual application which triggered the read() syscall. All file system operations can be implemented in the same manner.

In order to allow non privileged users to mount the file systems, the fusermount utility was created. It also offers the possibility to customize several parameters when one mounts a file system.

FUSE exposes two different APIs used to develop the file system:

- low-level: using this API the file system will need to manage the inodes and path translations (also caching for the translations), copying the VFS interface very closely. Also, the file system will need to manually fill and use the data structures used to communicate with the kernel. This API is mostly used to develop file system from scratch (take a look at ZFS-FUSE), in contrast to those that are just adding some functionality to existing file systems (extending them).
- high-level: opposing the low-level API, using this level will force the file system developer to deal only with pathnames, not inodes and dentries, copying more the system calls interfaces than Virtual File System interface. Also, there is no need for any caching mechanism for inode-to-pathname translations and the data structures used for kernel communication are already filled. This API is used by the vast majority of FUSE-based file systems, taking care of most of the work in order to let the file system developer build the desired logic.

The vast majority of FUSE-based file systems are implemented as standalone applications that link with libfuse and use the high-level API to implement a specific behaviour. Because of the FUSE (kernel module) acts like a proxy to a libfuse based application, the performance is drastically lower than a usual kernel file system. Even though the performance may be lower, the speed of development is drastically improved. Libfuse has bindings in all major high level languages. Also, because FUSE will take care of inodes, dentries and caching the inode-to-pathname translations, the margin for errors is drastically reduced.

2.2 Git

Git is one of the most popular distributed version control systems and it is also free and open source. It was originally created by Linus Torvalds with the purpose of improving Linux development which had become very cumbersome to maintain, due to the large number of patches that needed to be reviewed and integrated in the kernel.

The central component of git is the repository. A repository is a directory which contains all the necessary files that git uses to manage the content and the history. This repository can be bare, containing only the necessary parts without the content itself, or non-bare and including the content.

In order to get a repository on the local system you need to clone it. You can clone a repository from a local machine or a remote one. Git can use different protocols in order to clone a repository (ssh, http/https or git protocol). Every change that was managed by git can be pushed to a remote repository. Each repository, on which you want to push or from which you want to pull changes, has an address and a name. The data structure that encapsulates these attributes is called a remote. You can have multiple remotes on a local repository, with different addresses but with unique local names. The default remote (initially used to clone the repository) is called origin.

Using remotes you can distribute the code and work offline. Changes can be synchronized between multiple remotes and any raising conflicts would be solved automatically (if possible).

2.2.1 Overview

After a remote repository is cloned on the local machine, one can start changing its content. Git will track only the files that are already in the repository. Newly created files need to be manually added as tracked. Empty directories are not track-able.

Any new changes are viewed and tracked by git, but not automatically transformed in a version. In order to group the changes, git uses the concept of stage. Each new change is tracked by git in the so called working stage. In this zone, the changes are very volatile. If you want to pull content from a different remote, git will stop you in order to save the local changes. In order to group them, you need to add the changed files to the staging area, by using ‘git add‘ command.

Once you group the changes together, by moving the files to the stage area, you can create a version. That version is named commit. A commit is a git object, which contains a group of files that were changed, the name of who made the changes, who created the commit and the time of when these happened. Besides this metadata, git will attach an unique hash to it in order to be easily identifiable and a reference to the previous commit, called parent. In this way, git can offer an ordered log (not by time, but the order of changes, because you can rewrite the history and relocate a given commit in its tree of changes) giving you a broader understanding of the changes that were made.

Given the fact that each commit has a parent, you can easily form a chain of commits, called a branch. You can create as many branches as you want, starting with a commit. The default branch is considered to be the master. A branch can be created just locally and pushed to any remote repository afterwards. The last commit from a given branch is called HEAD. You can change the HEAD at any time, changing the state of the local or remote repository.

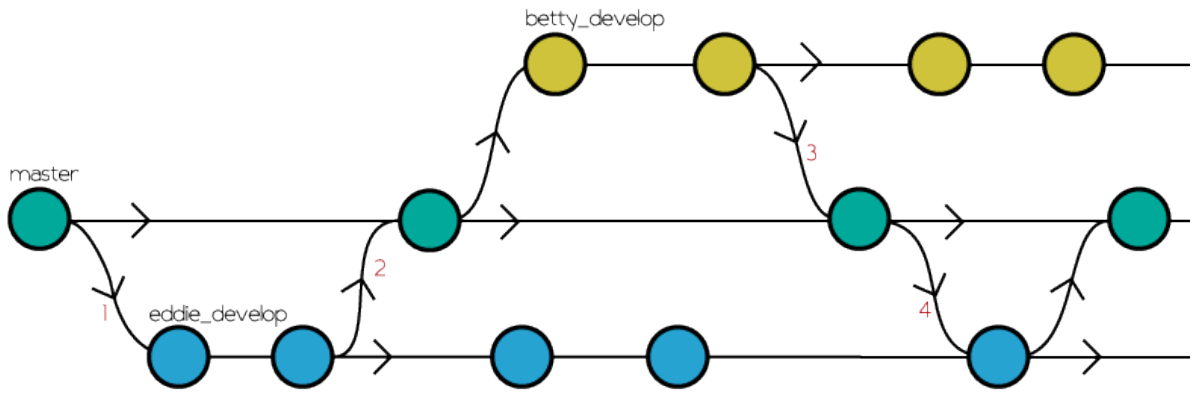

Figure 2.3: Git's chain of commits [Com]

From a commit two branches can diverge, as showed in figure 2.3. If you want to bring commits from another branch to a given current branch, you can do that by using one of two methods: rebasing or merging. Using the rebase method you will apply the current commits (set of changes) to the last commit of the branch you want to rebase into, in the order of which the commits were created, one by one. Any conflicts will be solved at commit level, if this will be the case. The merging method will try to apply the changes without keeping track of the order of the commits. All conflicts will be solved at the end of the operation and will end with a commit. The safest and cleanest method is the rebase method, but a big inconvenience is that it will rewrite the history, creating new commits (with the same data and metadata: date, author, commiter etc.) for each commit applied.

You can use any of these methods in order to apply remote changes to current state of the content. Git can manage the branches from all the remotes and using some porcelain commands, you can decide which changes to be applied to the local content.

Another useful structure offered by git is the tag. A tag is a frozen reference which will point to given state in the repository, just like a commit. Unlike a commit, a tag can't be changed and has no parent. Tags are used in order to froze the state of a given

repository.

You can imagine the entire repository like a tree, with multiple branches. The local repository will point to a certain reference. This reference can be a tag, the head of a branch or just the hash of a commit. In this way, you can easily change the state of your repository, by checking-out on different references. The checkout operation can have multiple modes (soft, hard) and in case of conflicts, you can solve them manually or by accepting the local changes (referred as ours) or the remote ones (referred as theirs).

2.2.2 Internals

Until now we have talked about git more from a functional point of view, like a tool. But git can be viewed like a database as well or, more precisely, like a key-value database. In this way, one can refer to git as a content addressable file system. You can add any type of content into it and receive a key which you can use to retrieve the content back. In order to investigate this further we need to take a look at dot git directory.

Dot git directory contains all the data and the meta data of our repository. The part of meta data is stored in the very descriptive files. Some of the most important are:

- HEAD: the current reference to which your repository points. Usually it will be refs/heads/master
- refs: all known references, grouped by remotes. Here you will also find the mapping between branches and commits.
- index: the staging area with it's meta data, like file names, timestamps and hashes of the files managed by git.

Beside those meta data files, dot git directory hides the entire repository's state and history in a directory called objects. Git will store every file ever changed in two type of data structures: tree and blob. Each such object is represented by an unique hash. SHA-1 hash function is used, over the content and the meta data of an object, in order to generate a hash. That hash is used to map an object to it's content. The content is archived (using the zip format) and stored on the disk, in objects directory using the hash as name for the file.

It's very similar to how an UNIX file system manages the files. The tree objects will correspond to an UNIX directory and a blob to, more or less, an inode or file contents. Every tree object can contain one or more trees, each of them contains a SHA-1 pointer

to a blob or a subtree, having associated file name, mode and type. Git will create those objects by retrieving the state of your repository (staging area and index) and writing tree objects from them. It will create new objects only for changed files and reuse old nodes if their corresponding files were not changed. In this way git can optimize disk memory usage.

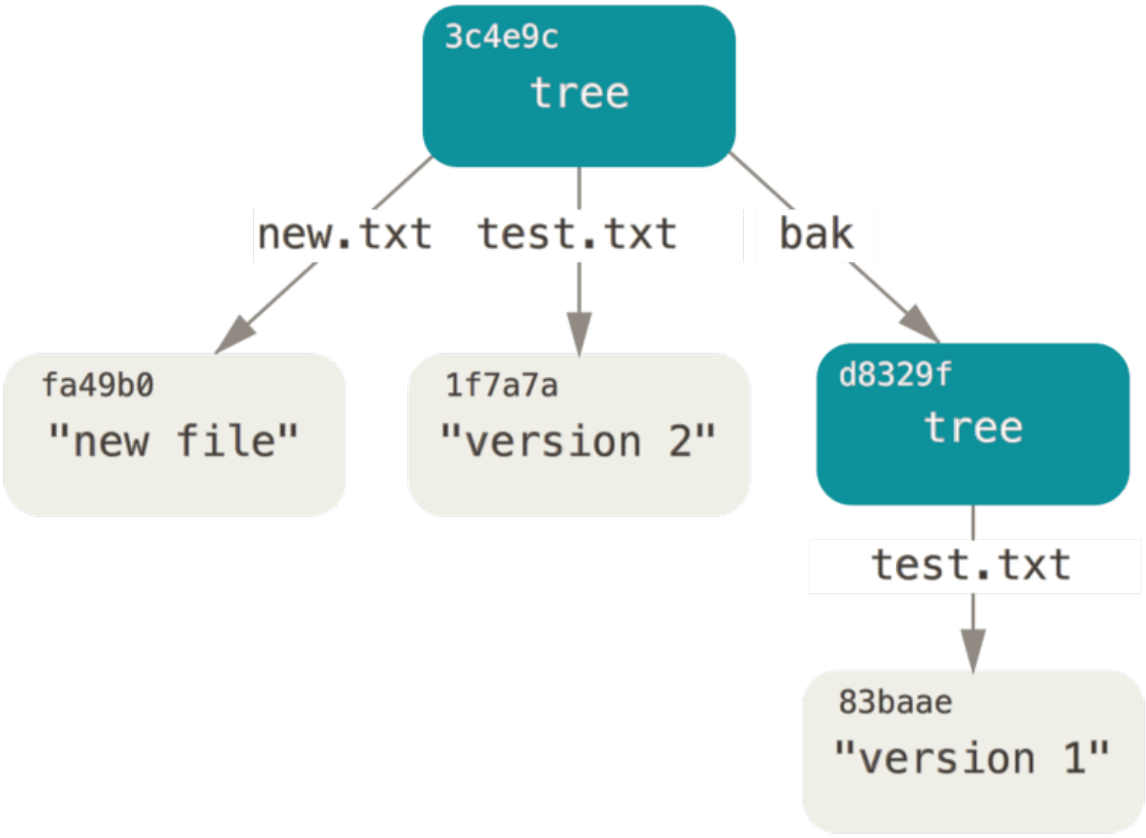

Figure 2.4: Git data model [GOB]

As git will not store empty directories, the leafs will be all the time blobs, not trees, as showed in figure 2.4.

As one can observe, by using this mechanism a tree is formed for the current state of the repository (and it will be used to compose other version of the repository). This tree data structure is based on Merkle tree, a type of tree in which all the nodes are uniquely identified by a hash, except the leafs. This data structure is very efficient for lookup operations, which can be done in O(logn) time. In this kind of tree, if a node's hash is changed, all the nodes up to the root need to update their hashes. In this way, git will create a new tree for each commit you create, but in that tree, the nodes that were not changed, will be reused.

Another important object is the commit object. This object will store a reference (SHA-1

hash) to the root of a the current tree (which represents a snapshot of the current state of the repository), a reference to the previous commit (called parent), the author and commiter (can be separate persons) and a blank line followed by the commit message. In this manner, a graph will be formed, as showed in figure 2.5.

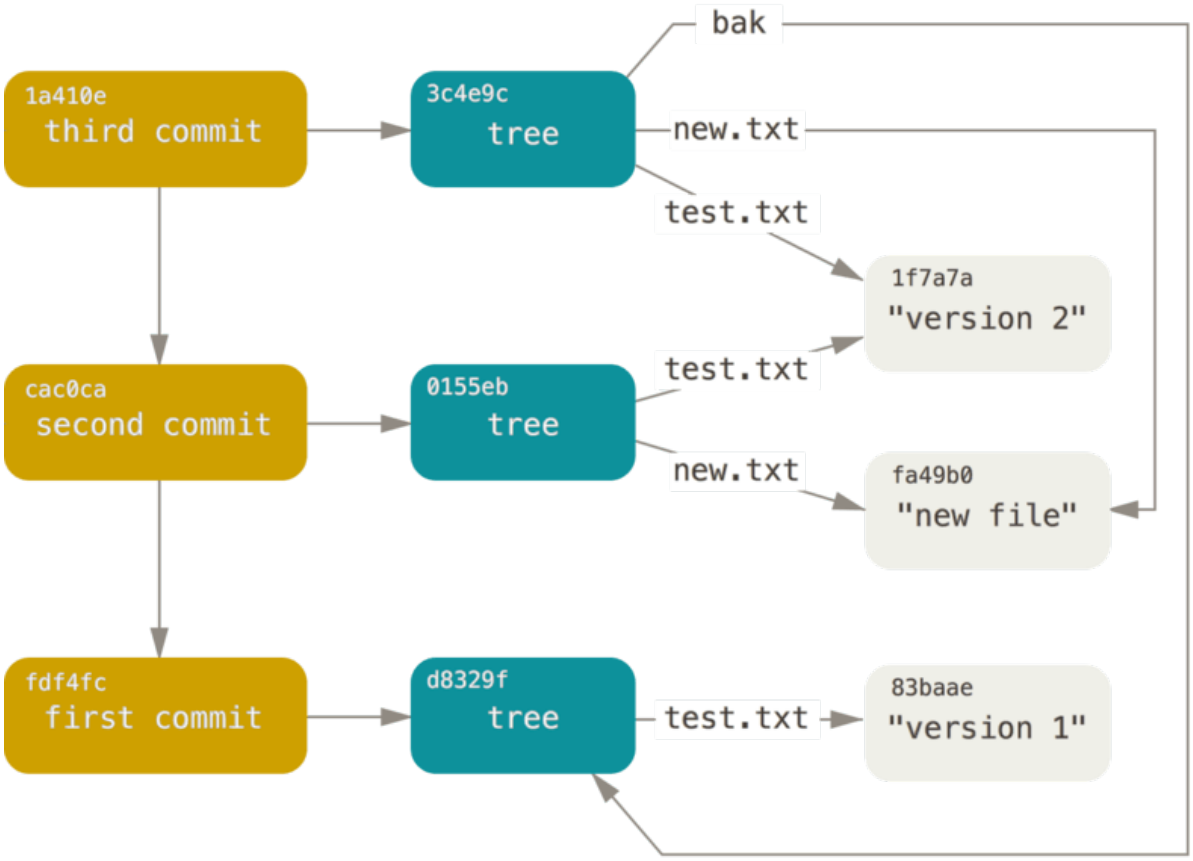

Figure 2.5: Git's objects and their relations [GDM]

2.3 Python

Python is a popular, general-purpose, high-level programming language. It was designed with a strong accent on code readability and an easy to learn syntax which makes it a very easy language to learn and maintain. The language has a lot of constructs intended to create clear and readable small and large scale programs. Using those constructs programmers can express complex ideas much faster, cleaner and in fewer lines of code, compared to other high-level languages as Java, C# or C++. It's ancestor, abc language, was created with the purpose to be a teaching language and Python inherited a large part of it.

One of its first usages (remaining today as an important and large one) was as a scripting language, used for servers administration. Being a high-level programing language it also supports different programming paradigms: object-oriented, functional, imperative or procedural. As a scripting language, it has a dynamic type system and automatic memory management with a large standard library.

Starting first as a scripting language used for server administration, a lot of system programming features emerged, making it a strong candidate for file systems development.

Python is the specification of a language. Its implementation varies, depending on which subset one wants to implement and in what language. The reference implementation in C language is called CPython. It is the most popular one and it is found already installed in almost any Linux distribution. PSF (Python Software Foundation, non-profit) manages all CPython development.

2.3.1 Advantages

1. Python is dynamically typed: it means that you don't declare a type (e.g. 'integer') for a variable name, and then assign something of that type (and only that type). Instead, you have variable names, and you bind them to entities whose type stays with the entity itself. a = 5 makes the variable name a to refer to the integer 5. Later, a = ”hello” makes the variable name a to refer to a string containing ”hello”. Statically typed languages would have you declare int a and then a = 5, but assigning a = ”hello” would have been a compile time error. On one hand, this makes everything more unpredictable (you don't know what a refers to). On the other hand, it makes it very easy to achieve results which static typed languages make very difficult.

2. Python is strongly typed. It means that if a = ”5” (the string whose value is '5') will remain a string, and never coerced to a number if the context requires so. Every type conversion in python must be done explicitly. This is different from Perl or Javascript,for example, where you have weak typing and can write things like ”hello” + 5 to get ”hello5”.

3. Python is object oriented, with class-based inheritance. Everything is an object (including classes, functions, modules, etc), in the sense that they can be passed around as arguments, have methods and attributes, and so on.

4. Python is multipurpose: it is not specialised to a specific target of users (like R for statistics, or PHP for web programming). It is extended through modules and

libraries, that hook very easily into the C programming language.

5. Python enforces correct indentation of the code by making the indentation part of the syntax. There are no control braces in Python. Blocks of code are identified by the level of indentation. Although a big turn off for many programmers not used to this, it is precious as it gives a very uniform style and results in code that is visually pleasant to read.

6. The code is compiled into byte code and then executed in a virtual machine. This means that precompiled code is portable between platforms.

2.3.2 CFFI

There are various tools which make it easier to bridge the gap between Python and C. One of the most used and spread tool is CFFI (C Foreign Function Interface).

FFI refers to the ability of code written in one language (the "host language," such as Python), to access and invoke functions written in another language (the "guest language," such as C). The term "foreign" refers to the fact that the functions come from another language and environment. Depending on the language and its FFI support, you might also be able to access global named variables, automatically convert data types between the host and guest languages, and have code in the guest language invoke functions in the host language as callbacks.

In interpreted languages like python, it's usually not possible to use a library's compile-time features like C preprocessor macros and constants (i.e. things #define'd in the library headers). This is because the FFI accesses the library's binary code (e.g. its .so, .dylib, or .dll) directly, without compiling any code. However, the FFI support in some compiled languages works by compiling down to C code; in these cases, you may be able to use compile-time features. It all depends on the language and how it implements FFI.

CFFI can be used in one of four modes: "ABI" versus "API" level, each with "in-line" or "out-of-line" preparation (or compilation).The ABI mode accesses libraries at the binary level, whereas the API mode accesses them with a C compiler. In the in-line mode, everything is set up every time you import your Python code. In the out-of-line mode, you have a separate step of preparation (and possibly C compilation) that produces a module which your main program can then import.

2.3.3 Fusepy

Fusepy is a Python module that provides a simple interface for FUSE and MacFUSE. It is just one file and it is implemented using ctypes.

The original version of fusepy was hosted on Google Code, but is now officially hosted on GitHub.

Fusepy is written in 2x syntax, but it is trying to also pay attention to bytes and other changes 3x would care about. The only incompatible changes between 2x and 3x are the changes in syntax for number literals and exceptions.

2.3.4 Libgit2

Libgit2 is a library written in C which manipulates the git objects at as low level as possible. It follows git implementation and offers an API which helps developers in creating applications that use git, without the use of the git binary.

A great advantage of Libgit2 is that it is written in C, offering bindings in all major high level languages, using CFFI (C Foreign Function Interface). It offers features like, cloning a repository, creating commits, merging commits and fetching changes from different remote repositories.

2.3.5 Pygit2

One of the languages in which Libgit2 has bindings is Python. The implementation of Libgit2 in Python is called Pygit2. It uses CFFI and Python C Objects in order to bring the functionalists of Libgit2 into Python world.

It is a portable library, having releases for Linux, Windows and MacOS, with a small community, but plenty of contributions.

Because Python is an object oriented programming language, Pygit2 exposes all of Libgit2's data structures as classes. In this way, Pygit2 offers a mechanism which allows you to extend the functionalists from Libgit2. From those data types, the Repository is the one which encapsulates the most of part the behaviour, using other data types like Remote, Branch and Commit.

2.4 Similar Projects

2.4.1 ZFS

ZFS is a combined file system and logical volume manager designed by Sun Microsystems. The features of ZFS include protection against data corruption, support for high storage capacities, efficient data compression, integration of the concepts of file system and volume management, snapshots and copy-on-write clones, continuous integrity checking and automatic repair.

ZFS uses a copy-on-write transactional object model. All block pointers within the file system contain a 256-bit checksum or 256-bit hash (currently a choice between Fletcher-2, Fletcher-4, or SHA-256) of the target block, which is verified when the block is read. Blocks containing active data are never overwritten in place; instead, a new block is allocated, modified data is written to it, then any metadata blocks referencing it are similarly read, reallocated, and written. To reduce the overhead of this process, multiple updates are grouped into transaction groups, and ZIL (intent log) write cache is used when synchronous write semantics are required. The blocks are arranged in a tree, as are their checksums (see Merkle signature scheme).

An advantage of copy-on-write is that, when ZFS writes new data, the blocks containing the old data can be retained, allowing a snapshot version of the file system to be maintained. ZFS snapshots are created very quickly, since all the data composing the snapshot is already stored. They are also space efficient, since any unchanged data is shared among the file system and its snapshots.

Writeable snapshots (”clones”) can also be created, resulting in two independent file systems that share a set of blocks. As changes are made to any of the clone file systems, new data blocks are created to reflect those changes, but any unchanged blocks continue to be shared, no matter how many clones exist. This is an implementation of the Copy-on-write principle.

2.4.2 SparkleShare

SparkleShare is an open source software which offers you the possibility to share a directory with more collaborators and create versioned content. It is presented as a standalone application in which an user can setup multiple projects (different directories) which will get synchronized with other collaborators’ changes.

It uses git as storage layer, which gives you the possibility to control where your data is stored, in order to protect your privacy. To make setup easier, the team which developed SparkleShare also developed a series of scripts which automate private git host creation. Beside those scripts, you can use any git hosting services, from Github, Bitbucket (which are proprietary solutions) to Gitlab or Gogs (open source solutions). In order to help you maintain the privacy it also includes client side encryption.

It is written in C# and it uses git command line tool for git operations. Being written in C# it has some drawback for portability, but it can compensate with higher code maintainability and extensibility options, and also performance boost. Unfortunately, the performance boost given by C#, will be annulled by the overhead in using the git command line tool. Spawning processes for each git command can lead to complicate error handling logic and resource heavy usage, because you rely on the git's command line tool implementation.

Beside file synchronization, it also offers the possibility to revert files. It is suitable for small files (from text to media files), but that great when it come to large binary files, full computer backups or large collection of files.

Chapter 3

Project Implementation

In this chapter we'll get into implementation details, extending the solution described earlier in this paper. We'll be presenting the file system layout and how a user can view and create versioned content. The concurrency model and how the content is fetched from remote repositories and merged will be also presented in the chapters below.

3.1 File System layout

When mounting Gitfs, a bare repository and a mount point need to be specified. Gitfs will maintain a local copy of the repository by cloning it into a defined path, named as 'repo_path'. By default, the location of this cloned repository is set to 'varlibgitfs'. Here, Gitfs will create a new directory, obtaining its name from the path (can also be an Uniform Resource Locator - URL) to the bare repository. One can override this default behaviour and specify an absolute path in which the repository should be cloned, using '-o repo_path=absolutepath'. Gitfs uses the git's objects of this local repository in order to retrieve its content and manage all versioning operations. It can clone local, non-bare repositories, but it doesn't support pushing changes to them. For now, it knows to follow only one branch and if no branch is specified, when mounting, it will clone only the master branch and follow it.

Once cloned, the content is retrieved from git objects, using pygit2, and presented in the mount point directory through two directories: current and history. The entire file system will have as owner and group the ones from the users which did the mount operation. This behaviour can be overridden, by specifying at mounting under what group and owner should be mounted.

In current directory the current state of the content is just a mirror for the cloned repository. Here, all changes will become commits in the git repository and will get synchronized with the remote. The second directory is called history, a pretty intuitive name. This directory contains n other directories, one for each day in which a change was made, and those directories will contain m other directories, one for each change in that day. This means that the entire number of directories in the history directory is $n * k(k \in \mathcal{N}, 0 < k < m + 1)$, each one of them representing a state in which the repository's content was at a certain time.

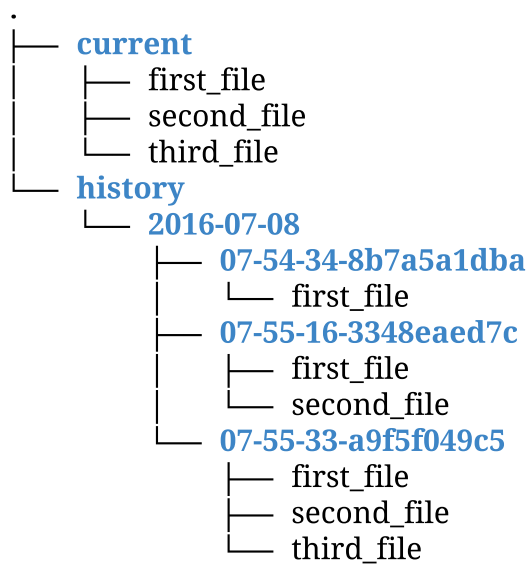

Figure 3.1: Gitfs directories layout

The current and history directories, showed in 3.1, are virtual directories which do not exists on disk. The entire set of operations which is done on the current directory is actually performed on the locally cloned repository. Also, in history directory, all those directories are not physical on the storage device. Those directories are created from git objects of local repository. Those objects are only once loaded in memory and cached, and from them, whenever one uses history directory or one of its directories, the content located in those is generated dynamically from the cached git objects.

3.2 Views

In order to better explain how Gitfs manages the content in those directories and the file system's operations, we can make a parallel example with how a HTTP service handles

the requests.

The entire file system layout can be translated into URLs. Each path in the file system can be viewed as an URL, being part of an HTTP service. When a request is made on a certain URL, a handler will be trigger and a response will be returned. The request can be made with different HTTP (Hypertext Transfer Protocol) methods: GET, POST, PUT, DELETE etc [FGM+99]. For each of those methods the request handler can respond with different content. When a client requests the content from a certain URL it will use GET. When he wants to change it, it usually uses POST, but in order to differentiate the type of the change, other methods can be used (PUT, DELETE or PATCH). The request handler can act like a proxy and send it to other parts of the application. These get to do the actual work and return a response to the request handler which will return it back to the client.

Those parts of the application which implement the business logic are usually called controllers, in a MVC (Model View Controller) framework [Dea09]. Other parts from a MVC framework are: Model (contains logic which deals with data manipulation) and Views (those components are what the client actually see). Traditionally, the Controller serves as a proxy for data, from Model to View. In a modern MVC architecture the business logic can be placed in an intermediary layer, between Models and Controllers, or directly in Models. Having multiple controllers, the request handler needs to decide which controller to instantiate and use, for which URL. A solution to this problem will be to route the request using some predefined routing rules. Those rules can be described using regular expressions. Because the job of a request handler is to route the request to a controller, we'll be calling it Router.

In figure 3.2 the routing mechanism in a web MVC framework can be observed.

As one can observe, this behaviour can be applied very easily to a FUSE based file system. In order to develop a FUSE file system one needs to implement an interface that represents the set of operations which can be performed on that file system. To be more specific, fusepy offers you a base class, called 'Operations', that acts like an interface (but in Python, being weakly typed, it doesn't provide the concept of interfaces). You need to inherit from that class and implement a set of methods that represent the set of operations supported by a file system (read, write, open, close, mkdir etc.).

Because all of those methods will have a pathname as on of the arguments, we can use the model described earlier, inspired by the MVC pattern. The paths to different files or directories can be viewed as URLs and the operation as an HTTP request. We can implement different controllers for different paths and use a router to route a certain operation to a controller.

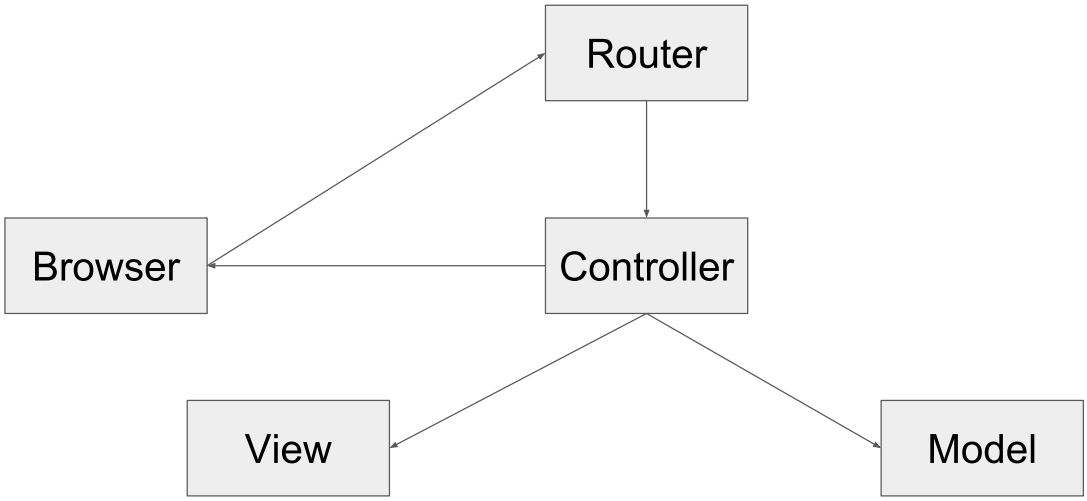

Figure 3.2: Http routing in a MVC framework.

This idea came originally from Django (a popular Python based web framework [Dja]) which implements a variation of MVC called MVT (Model-View-Template). In this pattern, the View actually represents the controller and the Template will be the View (a little bit confusing). That's why, in our implementation, the Views are actually Controllers.

For each directory from the mount point we have implemented a series of views: CommitView, HistoryView, IndexView and CurrentView. In order to reuse the common logic of those views, we have created two classes:

- PassthroughView - a view that doesn't change the behaviour of the file system. It's implemented as a blind proxy to the actual kernel file system by calling, for each operation, the corresponding system call.
- ReadOnlyView - used to restrict the access to any write operation. If a user space application wants to invoke a write operation, an EROFS [ero] will be raised.

Those views are inherited by the actual user facing views:

- PassthroughView
  - CurrentView – is the view that handles the current directory. It is a PassthroughView meaning that all operations will be executed by an individual file system,

without changing the outcome. This view is also responsible for managing the changes and initiate the commit process.

- ReadOnlyView
  - HistoryView – view which handles the history directory and group commits by day. Also, it handles the history/day directories. Using the git objects from the previously cloned repository, it can retrieve commits and, using a hashmap, it will group them by day.
  - CommitView – is the view which handles the history/day/commit directories. In a such form, a given directory represents a snapshot of the repository, when a certain change was made, in a certain day. This kind of structure is used to retrieve content, from an earlier version. It does that in a more tangible way, without the help of any commands. CommitView is read-only because you are not allowed to change the history. In order to restore a file to an older state, you need to copy it from one of those kind of directories and put it in the current directory.
  - IndexView – is the view which handles the directory that represents the mount point. It's responsible only by current and history directory and doesn't implement any custom logic.

All those views inherit from a base class called View. The View class has the purpose of mixing the Operations and LogginMixin classes from fusepy. Beside that, it also offer an easy mechanism for attributes initialization. See figure 3.3 for a better overview of the architecture.

There is one special component which is not in figure 3.3. That component is the Router. In order to implement the routing logic, we need a class which will look a lot like a view. Event if the Router has almost all components of a View, it can't be considered a view, because its purpose is not to implement any file system specific logic, but to act like a proxy for those components which implement.

The Router is the entry point for our file system. When mounting the file system, fusepy will receive as an argument an Operation based class. The Operation class, from fusepy, has some methods which need to be implemented in order to associate a specific behaviour to the file system. In our case, the Route doesn't need to do that, it only needs to pretend that will implement the method, but under the hood it will use the methods from specific views.

After the mounting operation is complete, FUSE kernel module, will take care of the operations. For each system call, invoked by an user space application, it will spawn a

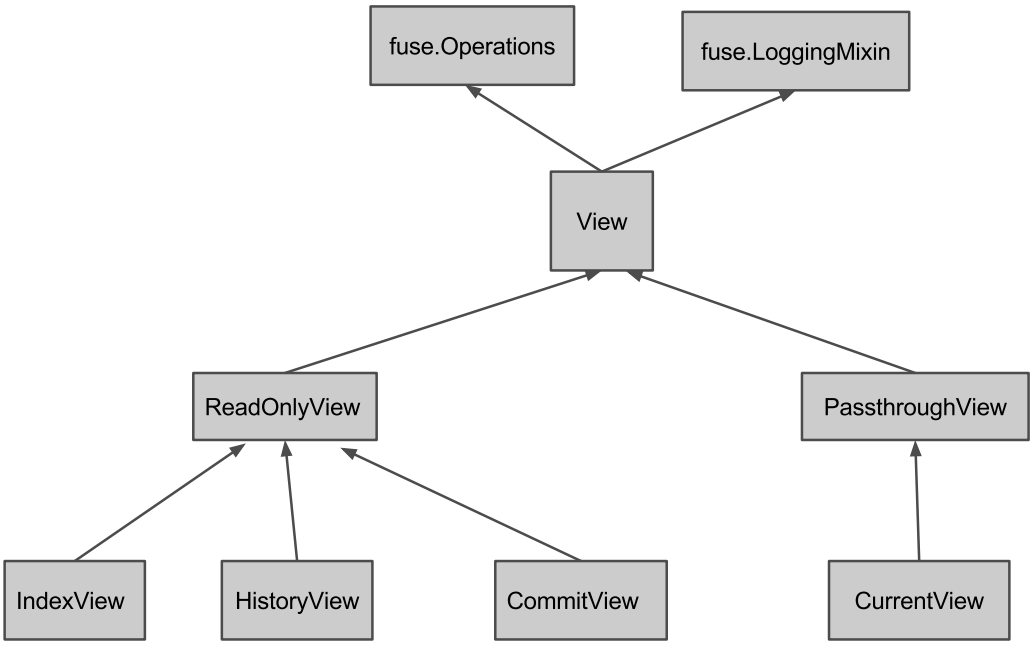

Figure 3.3: Views class diagram

thread in which will use the object passed to it at mounting and try to call the method which implements the requested operation. Python, being a dynamic and weak typed language, we can intercept this call and inject some custom logic.

The router receive what operation needs to execute and a path to a certain file. Based on some given regular expression, it determines which view to use. Instead of initializing a new view for each operation, a cache is used. In that cache are stored initialized views, one for each file path that was requested. The cache is implemented as an LRU cache [lru] in order have a hard limit on the quantity of memory used by Gitfs. The router will search in that cache, based on a cache key composed by the path of the file, and if the path is in cache, it will return the view associated to it, otherwise it will create a new one and store it in the cache.

Once the view is obtain, the Router will call the specific method which corresponds to the requested operation. A complete example is illustrated in figure 3.4.

As you can see from the example, when a read system call is used by a user space application, FUSE will trigger a read operation call, using the Router for that. The Router will check if the path is not in the cache. If not, based on regular expressions, will choose to use the CommitView, because the path to the requested file contains a full snapshot description. The Router will also pass some important information to the view, like the name of the requested file and the commit hashed, processed from the path. With those information, the CommitView will be able to retrieve the content for

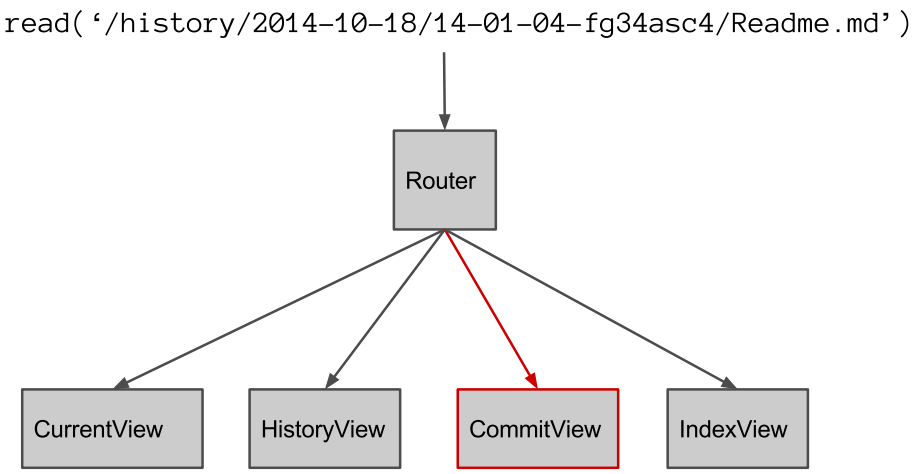

Figure 3.4: Operation call flow

the 'Readme.md' file, from the given date.

A common behaviour is to access the history directory, in order to see the content of a snapshot. In order to list all those directories, grouped by date and commit can lead to serious performance issues. This operations requires the git objects, in order to extract the desired content. Those git objects are on the actual disk, in the cloned repository. To avoid accessing the disk for each directory listing, we implemented an object cache in which we load all those objects once, when the file system is mounted, and every newly initialized view will be able to access them. To keep the cache consistent with repository's changes, it is invalidate at each commit and synchronization task. This process will be described later in this paper.

3.3 Concurrency model and behaviour

In order to offer a better overview of the synchronization mechanism, a presentation of the actual concurrency model is required.

As we have discussed earlier in this paper, FUSE will spawn, for each operation, a thread. This is very useful because the user can open multiple files in the same time or write onto multiple files in the same time. Another reason would be that it makes the synchronization problem much easier to solve, given the fact that those threads share

the same memory. In this situation, we can share between all the threads managed by FUSE, data structures which can lock different tasks or pass information between threads.

Another interesting fact is that a FUSE based file system is a self contain process, running in user space. After the file system was mounted, a process was spawned, representing the entire application. This means, that beside the FUSE threads which share the same memory, each thread spawned by this process will share the same memory between them and between the FUSE threads' memory. Because of that, the synchronization mechanism can be decoupled by the views and router.

To keep things simple and easier, two threads, a queue and a series of locks are being used. We have named those threads workers because they are responsible for the heavy work involved in the synchronization mechanism.

All workers inherit the Peasant class which is nothing more than a specialized Thread with custom logging support.

- FetchWorker - fetches changes from remote repository with a configurable frequency (the option at mount is fetch_timeout, by default seted to 30 seconds). Between FetchWorker and SyncWorker a lock is shared in order to insure that only one fetch process is executed, at a certain point in time. The period between fetches can be configurable, by default being of 30 seconds.
- SyncWorker - is responsible with commit creation, merging local changes with the remote ones and pushing those changes to the remote repository. If any of these operations fail, the file system will be locked into a read-only state in order to prevent any data inconsistency problems.

Between the SyncWorker and FUSE threads are shared multiple data structure. A very important one is the job queue. After each successful change created by any CurrentView, a commit job is put on the queue. In the same time, multiple jobs can arrive on the queue. Those jobs are consumed by the SyncWorker.

For now, a CurrentView will just mark a file as being dirty and ready to be added to a commit, but will handle any commit creation responsability to the SyncWorker. The SyncWorker will aggregate those jobs and if no more commit jobs are being receive, for a given period of time (specified by 'merge_timeout' at mount, with a default of 5 seconds) it will try to start the synchronization process.

CurrentView instances and the SyncWorker share another useful data structure, under the form of an int. It's an atomic long, with the implementation in C and with bindings

in Python. This atomic long is used to keep track of how many files are still open for write or are writing. The tracking part is done by CurrentView which will increment this long each time a write operation will occur (write, chmod, chown, mkdir, rm etc.) and will decrement it when the operation was successfully finished.

The SyncWorker will check for any running write operation, using the data structure described above, and if there are still write operations, even if the merge_timeout has expired, it will wait as many cycles until all write operations are done and all files are closed. Using this mechanism will prevent any failing in merging and further in synchronization process.

If there is no any file open for write or any on going write operation, the SyncWorker will first try to make a commit. It can receive multiple jobs commits, each of which represents on successful change done by a CurrentView instance. Those jobs contains the file which was change, the path to it and the type of the change. In order to compose a meaningful message for the commit, it will try to understand the changes. If there is only one change it will create messages of form ”Deleted <path to file>”, ”Deleted directory <path to directory>”, ”Created the <path to directory> directory”, ”Chmod to <mode> on <path to file>”, ”Rename <old file> to <new file>”, ”Create symbolic link to <link name> for <target>” and ”Update <path to file>”.

Those kind of messages are used only when a commit is compose of only one commit jobs. Otherwise, to understand what type of change was made, is more complicated and for now it uses a message of form ”Update n items”, where n is the number of files or directories which were updated (created, changed or removed). After the commit was created, the SyncWorker will invalidate the commits cache for all views, forcing them to get newly created commit.

Now that the commit was created, it needed to be pushed to the remote repository as fast as possible. In order to make the process as safe as possible, the SyncWorker need to merge any existing remote changes with the local ones. In order to do that, it relies on the FetchWorker to retrieve remote changes. In the beginning of the process, the SyncWorker will set a lock on all FUSE threads in order to delay any write operation. In this way, the merge process is safe, insuring a clean staging area. Usually, the delay is not very much, depending on what kind of changes the SyncWorker needs to sync, being almost unnoticeable by the normal user (around 1-2 seconds), using a relatively small dataset of changes (<= 50 MB). Also, all fetches are stopped in order to avoid merging and race condition errors.

The SyncWorker will check it is ahead or behind. Having a commit created locally it surely is ahead, but it also can be behind if someone pushed other changes to the

remote. Being behind, it means that a merge needs to be done, in order to apply the local changes over the remote ones. The merging strategy will be presented more detailed later in this paper. If the merge fails, it will try 4 more times to do the synchronization process. If it still could not merge and push, an exception will be logged and manual intervention will be needed.

Before merging to start, another fetch is being made, in order to avoid not having all remote changes in the local repository (otherwise, it could lead to push errors and a retry should processed). In case of a successful merge, will try to push the changes on the remote repository. In case of failing, it will retry it after another cycle of timeout, restarting the synchronization process.

After a sync has been complete, write operations will be free to execute. Also, the fetches will start again.

This process can start even if there is no commit to create. The SyncWorker works in cycles with the period defined by merge_timeout (default to 5 seconds). If there was no activity on the file system, it will start the synchronization process in order to bring in the current directory any change which was pushed to the remote repository by a different system. If there are in progress write operations and remote changes which needed to be synced, the SyncWorker will wait until all write operations are finished and files are closes.

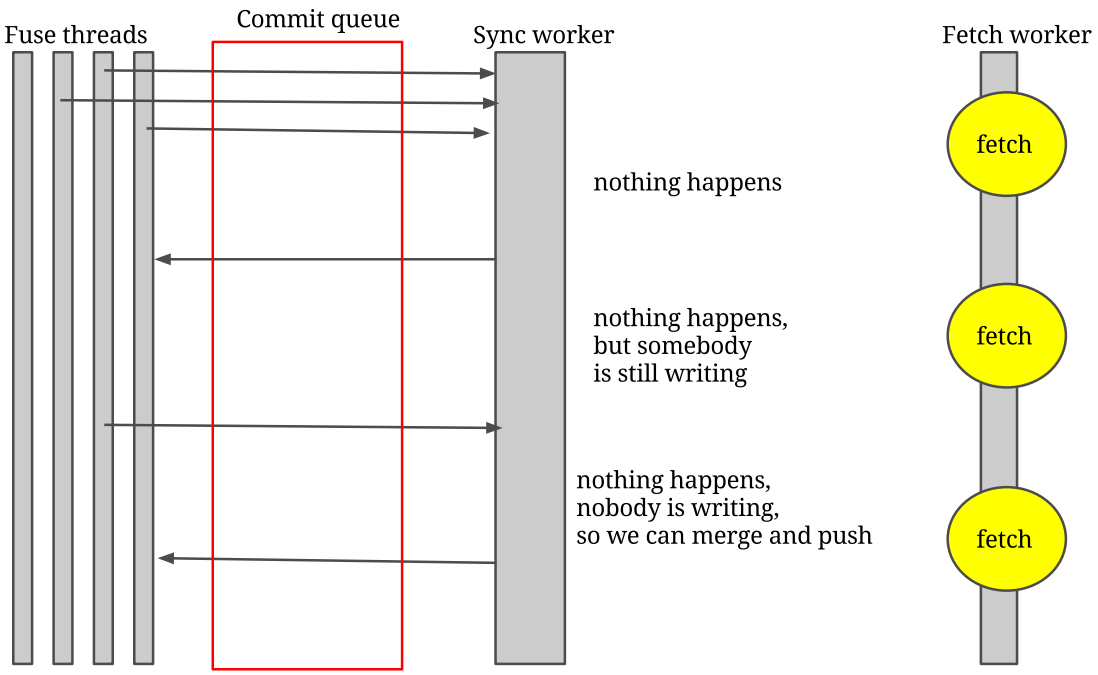

Figure 3.5: Concurrency model of Gitfs

In figure 3.5 are all components involved in the synchronization process.

If any errors will occur in this process, will merging, pushing or fetching, the file system will enter in read-only mode. In this mode any change operations are denied. This is a safe mechanism to ensure that no data is corrupted and that the repository will stay in a state from which will be possible to make it functional. In case of failure (network, human or simple bug), the SyncWorker will retry to sync the changes, using a total of 5 attempts. If there is no success after those, the human intervention is mandatory, otherwise the file system will remain in read-only mode.

Because running multiple Gitfs, each having the same server as origin (with different repositories), could lead to a network flood when multiple FetchWorkers will fetch in the same time and given the fact that if a user is not using the file system, doing fetches in the background means wasting resource, the idle mode was introduced. If Gitfs enters in this mode it will lower down the fetches frequency, changing the period between fetches to 30 minutes (given by idle_fetch_timeout which can be overridden at mount).

This idle mode is triggered after no operation was triggered by FUSE, meaning no active user is using the file system, for more than a given amount of time. This amount of time is given by the SyncWorker and represents the number of cycles that need to pass in order to switch to idle mode. This number is a configurable one and can be overridden at mount (min_idle_times, by default to 10). Given the fact that the default period of a cycle is 5 seconds, the idle mode will be on after 50 seconds of inactivity on the file system.

3.4 Merging

Earlier in this paper we discussed about merging remote changes with local changes. Gitfs uses a strategy called always-accept-mine. This strategy is the only one implemented right now, but the merging strategy component is implemented in an extensible manner.

We will use the situation represented in 3.6 as an example for this strategy. In figure 3.6 there are two branches represented, called remote and local, which have a common branch. On that common branch, commits 1, 2 and 3 will be present on local and remote branch. From commit 3, those branches start diverging. On branch remote commits 4, 5, 6 are created and on local, commits 7 and 8.

The plan is to apply local changes over the remote ones, obtaining a nice chain of commits: 1, 2, 3, 4, 5, 6, 7′ and 8′. In order to obtain that we need to know which was the diverging commit or the common parent.

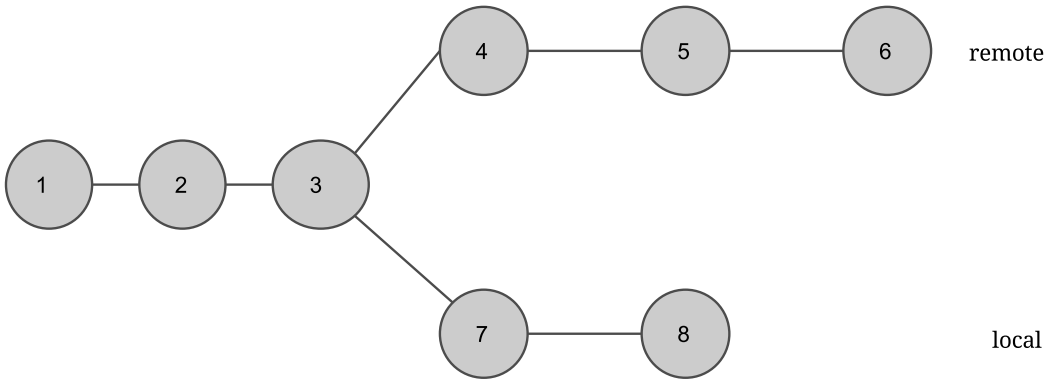

Figure 3.6: Diverging commits

For that we'll use very simple algorithm and two sets, one for each branch. We'll start from the end and the last commits of those branches. Each retrieved commit will be put in the corresponding set, but first we'll do a cross checking. We'll check to see if the commit from the first branch is already in the set of commits for the second branch. The same will be done for the commit of the second branch. If we find any commit in other branches lists, then we can stop, we found the parent.

In our example, we pick commit 6, from branch remote, and check to see if it's present in the set of commits for branch local . It's the first commit so this is not the case. We add it to the list of visited commits. Now we pick 8, from branch local, and check to see if is not already in the set of remote's visited commits. Only 6 is present so we'll add 8 in the set of local's visited commits. We'll continue to do the same for 5 and 7, 4 and 3. Now 3 is in the set of local's visited commits. Next we'll be comparing 3 and 2. 3 is already in a set so we can stop, because we found the last common commit.

This algorithm is a very simple one, in which we rely on the set data structure which offers us an $O(1)$ lookup complexity. Worst case time complexity, for this algorithm, is $O(n)$, where n represents the number of commits in a branch. In normal usage parameters, the number of diverging commits is very small, around 10-20.

Now that we found the last common commit, we'll rename the branches in merging_remote and merging_local in order to avoid braking up the local repository. We switch to merging_remote and use this branch as the active, working one. From merging_local we'll take the commits proceeding the common one and we'll try to merge them on top of merging_remote.

If one want to use another strategy, it needs to implement it by creating a class which inherits from the base strategy (which is locate in the mergers module) and has a method called merge. That method will be called with a reference to the local branch, on to the

remote branch and a reference to the remote repository. After the code base is merged and a new build is released, the newly created strategy can be used. We'll be using Libgit2 for merging, which will try to merge the commits by using a 3 way merge strategy. If it fails, we'll need to handle it manually.

In case of conflicts, Libgit2 will return a data structure which will help us solved them easily. There are three types of conflicts which this strategy knows to resolve:

- delete by us, but not by them. In this case we'll be deleting the file.
- delete by them, but not by us. In this case we'll be adding back the file.
- both changed the file. In this case we'll be overwriting the file with the content from our local repository. Is not the best approach because it can override already merged changes.

After each merge with conflict, a new commit is created, having the same meta data as the local commit which was merged.

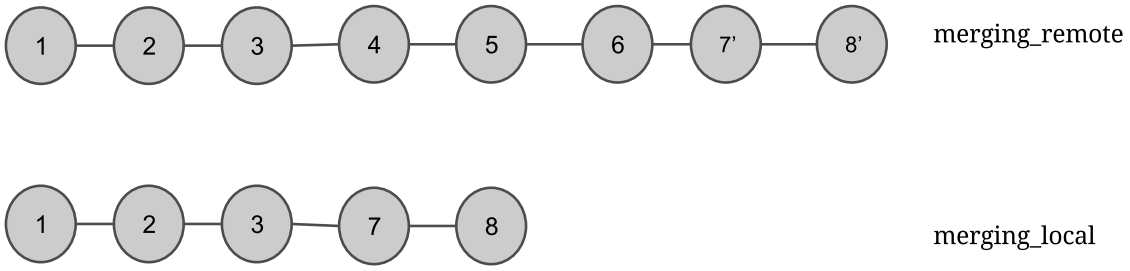

Figure 3.7: Branch layout after merging

In figure 3.7 you can see which and how the commits are being applied. Finally, we'll rename the merging_remote branch in local and remove the merging_local branch. In this way, the repository is not polluted with unused branches.

In case of exceptions, we'll be reverting to the local branch and remove any reference to merging_local and merging_remote.

Always-accept-mine strategy is a strategy in which local changes will be more important than remote ones. In this strategy, if a conflict is detected within a file, with changes coming from remote and local, the conflict will be solved by replacing its entire content with local content of the file.

3.5 Tests

A file system is an important part of a system. If the file system fails, other components may fail. Because of that, Gitfs has a large suite of tests, from integration to unit tests. Besides the components exposed in this paper, Gitfs has more utility components and data structure that need to be tested.

Testing a file system could seem easy at a first glance. If a user space application creates a file, that file should be on the disk and accessible by another user space application. In the context of Gitfs, the entire synchronization flow needs to be tested, in order to see if earlier created changes have arrived on the remote.

To replicate the testing environment as closely as possible to the general setup and to be easy to reproduce this environment, Gitfs uses privileged Docker containers [Mer14]. The entire environment is build using make rules and can be replicated in any testing setup (host, virtual machine or Linux containers [Ros14]). On the official repository, Gitfs is tested automatically whenever a new change is pushed to it. The Drone CI [Dro] is a continuous integration system that integrates with Github which, for every git push, will trigger a job drone. Drone is open source and configurable. Each git repository contains a .drone.yml file (having YAML format [BKEI09]), in which descriptive instructions are given to drone about the environment. Those instructions are very different starting from which Docker image to use as base environment to what testing command to use and what to do after the testing in case of success or fail.

In figure 3.8 the entire development pipeline is described.

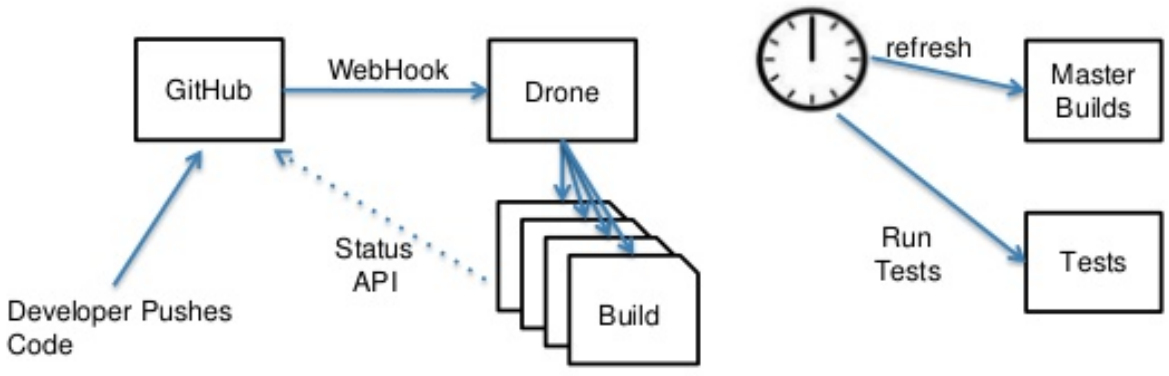

Figure 3.8: Gitfs development pipeline [GDR]

The testing environment contains a bare git repository, its local clone with initial files, directories and commits, and a directory which serves as the mount point. When the

testing process begins, the environment is created by different make rules [Ac15], followed by other make rules in order to run the tests collection. In the end, another make rule will be applied in order to clean the environment.

In this moment, Gitfs has 222 tests which offer a 92% coverage. The coverage percentage comes from unit tests, and doesn't reflect the true coverage [IH14]. In order to assure a better testing coverage, rigours integration tests are needed. In Gitfs every operations which an user space application can perform on a POSIX complaint file system is tested, ensuring the fact that Gitfs is a POSIX complaint file system. Beside those operations, the concurrency model is tested by simulating the collaboration part.

An usual test looks like this: once the environment is created and the file system mounted, we use an user space application to write content to a newly created file. We close the file and check to see that the content is there. We wait until Gitfs is doing the synchronization process. In another locally cloned repository, we fetch the changes which were supposed to be pushed by Gitfs. Finally we checked if the fetched changes are the ones which were made through the user space application. The entire process can be slow, but we are using a custom configured Gitfs instance, in which we set the merge_timeout and fetch_timeout as low as possible (100 milliseconds). Another trick that we use was not to wait an empiric amount of time until we thought that the process will be finished, but to use a custom log parser, which will analyse and block the execution of tests until a certain log message will be created by the SyncWorker and the FetchWorker.

3.6 Configurable Options

In chapters above we also presented different configurable options. Gitfs has more options which can be found below. Those options will be passed to Gitfs in the mounting process, using the -o argument of mount command. Multiple Gitfs options need to be concatenated and split by a comma (-o option1=value1,option2=value2...).

- remote_url: the URL of the remote repository
- branch: the branch name to follow (default: master)
- repo_path: the location where the repository will be cloned and used by Gitfs (default: $/var/lib/gitfs/repo\_path$)
- max_size: the maximum file size in MBs allowed for an individual file. If set to 0, then allow any file size (default: 10MB)

- user: the user that will mount the file system (default: root)
- group: the group that will mount the file system (default: root)
- committer_name: the name that will be displayed for all the commits (default: user)
- committer_email: the email that will be displayed for all the commits (default: user@FQDN)
- merge_timeout: the interval between idle state and commits/pushes (default: 5s)
- fetch_timeout: the interval between fetches (default: 30s)
- log: the path of the log file. Special name syslog will log to the system logger (default: syslog)
- log_level: the logging level. One of error, warning, info, debug (default: warning)
- debug: he switch that sets the log level to debug and also enables FUSE's debug (default: false)
- username: the username for HTTP basic auth
- password: the password for HTTP basic auth
- key: the path of the SSH private key. NOTE: the public key is constructed by appending .pub to this path and the file MUST exist (default: $HOME/.ssh/id_rsa)

Chapter 4

Conclusion

4.1 Summary of Thesis Achievements

Since the purpose of this thesis was to build a tool which will help non-technical users to create versioned content without having knowledge about any version control system, it is considered that its purpose has been achieved.

Gitfs is an elegant solution to this problem, offering a natural way to view and edit any type of content (from text to photos or music) using the file system interface and hiding the entire complex versioning process in an easy to understand concurrency model.

In order to create and maintain Gitfs, multiple contributions, to already mentioned libraries were made. Beside those, tools were created to automate releases and binaries publications.

4.2 Applications

Gitfs is a functional product, already used by Presslabs for almost two years, where more than 200 clients use it to manage their website code. Since its original launch, in October 2014, 15 other releases were made.

Besides Presslabs, other people have showed interest for Gitfs. On Github, 1025 people stared the repository and 63 people have forked it in order to make contributions. From those numbers it is very hard to tell how many are using it, but having 11 direct contributions and around 20 indirect contributions through bug reports, it is considered that Gitfs is doing what it was meant to do and that is helping people.

This file system was presented in multiple countries, within different conferences (local Timisoara meetups, Python conference in Iasi 2014, PyCon Sweeden 2015, PyCon Italy 2015, OpenTech in London 2015) with the biggest one being EuroPython 2015.

4.3 Future Work

Even if the project is mature, a lot of improvements can be made:

- Refactor the concurrency model to use fewer locks between threads and communicate more using queues. Right now, there are more than 10 locks, in order to ensure that everything is running correctly, which translates to unmaintenable code.
- Decouple the synchronization mechanism from the file system part.
- Reuse the synchronization mechanism with different frontend (instead of file system).
- Follow multiple branches and remotes.
- Allow users to control when a commit, push or fetch is made.
- Allow users to change file system options at runtime, using configuration files.
- Add support for LFS (git's large file storage), in order to offer a better support for big files.
- Improve commit's message. Right now, when multiple files are changed or created, a single message showing how many files were affected is created.
- Add support for multiple platforms. Right now, Gitfs has official binaries only for Ubuntu and other people have created packages for Fedora, ArchLinux and MacOS.

Bibliography

[Ac15] John G R Aham-cumming. *The GNU Make Book*, volume 2015. 2015.

[BKEI09] O Ben-Kiki, C Evans, and B Ingerson. YAML Ain't Markup Language (YAML™) Version 1.2. *Language*, pages 1–100, 2009.

[Com] Git's chain of commits and it's branches. `https://grapefruitgames.files.wordpress.com/2013/05/unitygitdiagram.png`. Accessed: 2016-07-07.

[Dea09] John Deacon. Model-view-controller (mvc) architecture. *Computer Systems Development*, pages 1–6, 2009.

[Dja] Django is a high-level python web framework. `https://www.djangoproject.com/`. Accessed: 2016-07-07.

[Dro] Drone CI is a continuous delivery platform built on docker, written in go.

[ero] Write permission was requested for a file on a read-only file system. `http://linux.die.net/man/2/access`. Accessed: 2016-07-07.

[FGM+99] Roy Fielding, James Gettys, Jeff Mogul, Henrik Frystyk, Larry Masinter, P. Leach, and Tim Berners-Lee. RFC2616 - Hypertext transfer protocol–HTTP/1.1. *Internet Engineering Task Force*, pages 1–114, 1999.

[FSA] High level system arhitecture. `http://www.ibm.com/developerworks/library/l-linux-filesystem/figure1.gif`. Accessed: 2016-07-07.

[Fus] System call flow in fuse. `https://lastlog.de/misc/fuse-doc/doc/html/490px-FUSE_structure.svg.png`. Accessed: 2016-07-07.

[GDM] Git's data model. `https://git-scm.com/book/en/v2/book/10-git-internals/images/data-model-3.png`. Accessed: 2016-07-07.

[GDR] Github, drone relationship. `http://image.slidesharecdn.com/rancherlabsoverview-aug15onlinemeetup-150814183957-lva1-app6892/95/building-a-scalable-ci-platform-using-docker-drone-and-rancher-20-638.jpg?cb=1439585795`. Accessed: 2016-07-07.

[GOB] Git's internal objects. https://git-scm.com/book/en/v2/book/10-git-internals/images/data-model-2.png. Accessed: 2016-07-07.

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test sute effectiness. *Proc. ICSE*, pages 435–445, 2014.

[Lew91] Donald Lewine. *POSIX Programmer's Guide*. Number March 1994. 1991.

[Liba] Libfuse - the reference implementation of the linux fuse (filesystem in userspace) interface. https://github.com/libfuse/libfuse. Accessed: 2016-07-07.

[Libb] Libgit2 is a portable, pure c implementation of the git core methods. https://libgit2.github.com/. Accessed: 2016-07-07.

[lru] Least recently used cache overview and implementation. http://mcicpc.cs.atu.edu/archives/2012/mcpc2012/lru/lru.html. Accessed: 2016-07-07.

[Mer14] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[Pyg] Pygit2 a set of python bindings to the libgit2 shared library. https://github.com/libgit2/pygit2. Accessed: 2016-07-07.

[RG10] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. *Proceedings of the 2010 ACM Symposium on Applied Computing SAC 10*, page 206, 2010.

[Ros14] Rami Rosen. Linux Containers and the Future Cloud. *Linux Journal*, 2014(239):Article No.2, 2014.

[Sin10] Andrew Sinclair. Licence Profile: Apache License, Version 2.0. *International Free and Open Source Software Law Review*, 2(2):107–114, 2010.

[Spi12] Diomidis Spinellis. Git, 2012.

[Wor] Wordpress - open source cms, written in php. https://wordpress.org/about/. Accessed: 2016-07-07.