Version Control System

Reece Donovan

B.Sc. Computer Science - Final Year Project

Prof. Ken Brown

University College Cork

November 15, 2022

Abstract

Declaration of Originality

Acknowledgements

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 What is a Version Control System

A version Control System (VCS) saves modifications done by individual software developers, making the process more accessible since they can track their work over time. In addition, it helps share data between nodes where each node can be kept up to date with updated versions, so there is no need for any merge conflicts.

The advantages provided by VCS include: aiding in collaboration among programmers, improved efficiency when working on larger groups of mixed products; providing an audit trail; easy branching functionality; simplifying team-based development; understanding who has worked on specific pieces or sections during what period in time but also making sure that all changes are appropriately tracked providing transparency within teams while maintaining optimal performance levels through streamlined management processes, helping avoid errors arising from inconsistency between versions such as mismatched documents or programming problems resulting from conflicting edit operations.

2.1.1 What is the purpose of Version Control Systems

For almost all software projects, the source code is an asset that must be protected. For most software teams, the source code is a repository of invaluable knowledge and understanding about the problem domain that developers have collected and refined through careful effort. Version control protects this priceless resource in many ways: it prevents catastrophe or casual degradation from occurring and prevents human error or unintended consequences from affecting its quality.

For example, software developers working in teams continually write new source code and change existing code; these are organized into file folders called "file trees". One developer may be working on a new feature while another fixes an unrelated bug by changing parts of the file tree; each developer may change several parts at once.

Version control can help teams solve these kinds of problems because every individual change made by each contributor is tracked, helping prevent concurrent work from conflicting with one another so that any incompatibility should be discovered and solved without blocking team members' progress further down the line (this will also ensure that any changes being made do not introduce bugs).

Further still, testing for new versions cannot occur until some development work is done beforehand. This means both processes move forward together until they reach their newest version - thus decreasing risk to both sides due to errors along either path.

2.2 Types of VCS

The two approaches to VCSs are Centralized and Distributed. Both approaches are in widespread use today. The centralized approach is based on the client-server model, where a single central repository stores the history of all files. In contrast, the distributed approach provides each user with a full repository copy.

While there is no clear answer for which approach is best, it's important to note that the centralized approach features a single point of failure but also more challenging scaling than its counterpart (Distributed).

2.2.1 Centralized Version Control Systems (CVCS)

A centralized Version Control System is a system that enables developers to work together on the same project by storing the primary copy of files in a central repository. This system keeps track of all files and saves information in the local repository. CVCS are called centralized because there is only one central server or repository.

The server maintains a complete record of issues, while clients only maintain a local copy of the shared documents. All developers make their modifications to the repository through checkout. However, only the last version of the files is retrieved from the server, meaning any modifications made will automatically be shared with other developers.

Users can modify in parallel with their local copy of shared documents and sync with the central server to release their contributions and make them visible to other collaborators. However, because centralized version control systems rely on one repository that includes the correct project version, they must restrict write access so that only trusted contributors can commit modifications.

CVCS has some challenges, such as if the central server is inaccessible, users will not be able to merge their work or save the released modifications. Also, if the central repository is corrupted, everything will be lost. Contributors must be the ones who have writing permissions to perform basic tasks, such as reverting modifications to a previous state, creating or merging branches, or releasing modifications with complete revision history. This limitation affects participation and authorship for new contributors.

So, the main drawbacks of using CVCS are that it requires a network connection to work on the source code, developers must order to contribute to a project, and a single point of failure is an issue when using one server.

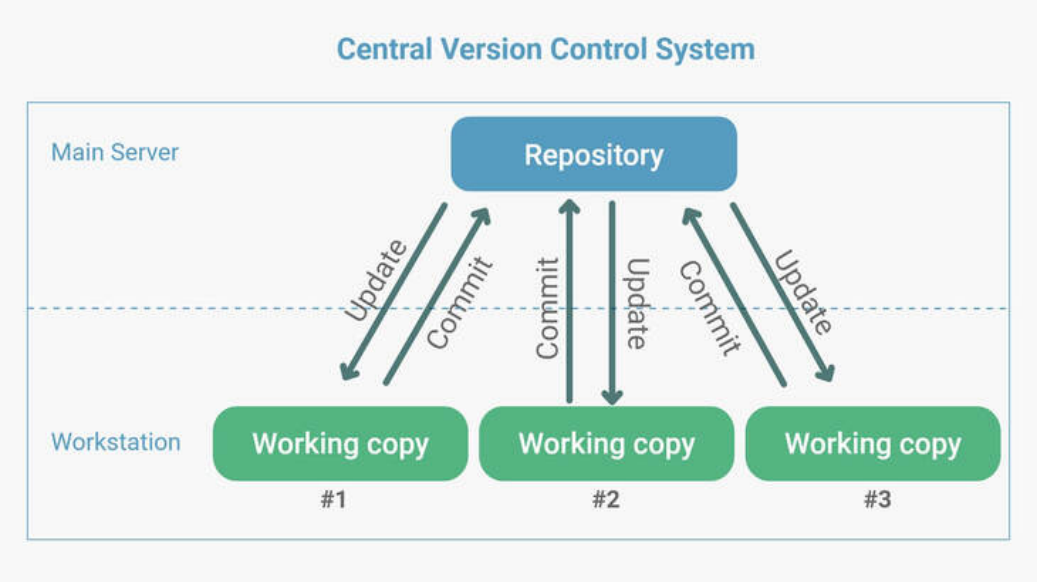

Figure 2.1: Centralized Version Control System

2.2.2 Distributed Version Control Systems (DVCS)

Distributed Version Control Systems (DVCS) were created to overcome the limitations of Centralized Version Control Systems (CVCS), which enable

branching and merging, avoid local VCS operations and allow developer collaboration. Due to the limitations associated with using a centralized version control system, Open Source Software (OSS) projects today broadly adopt DVCS.

DVCS is designed to work in two ways: it keeps file histories locally on each device. However, it can also sync local user modifications with servers again when necessary so that these modifications can be shared with everyone else. In addition, in DVCS, developers can work separately or together on the same project, as they have access to all repositories needed for their task; any repository can be cloned from another, so there is no repository more important than others.

To provide a new way for versioning software artefacts, several Distributed Version Control Systems emerged in the software field, such as `Mercurial`, `Git`, `Bazaar`, etc. These tools have been adopted by many Open Source Software (OSS). The operations in DVCS are much faster than those found in CVSS because they are done locally, while CVSS operations require remote connection; some consider that distributed systems will soon replace centralized ones because they suit more substantial projects with more independent developers who want full functionality even without network connection available, offer advantages like earlier drafts of your work being saved without requiring you releasing them publicly or sharing them with other people etc.

Collaboration between team members and allowing individual developers to be servers or clients are the most important features of version control systems, so developers can work on source code without being connected to a central or remote repository.

DVCS introduces some challenges: it lacks a coherent version numbering system, where there is no centralized versioning server, and uses hash modifications or a unique GUID. So, the lack of a central server makes system backup difficult.

The two most prevalent complaints about the disadvantages of DVCS are that: pessimistic locks are not available and they have weak tools for binary. However, the reasons for the transition from centralized to decentralized version Control Systems are that developers can work offline and work incrementally efficiently because they can make several roles, such as developing new tasks or fixing errors; this also leads to exploratory coding, which gives them more freedom in their workflow while still retaining control over their code's release schedule.

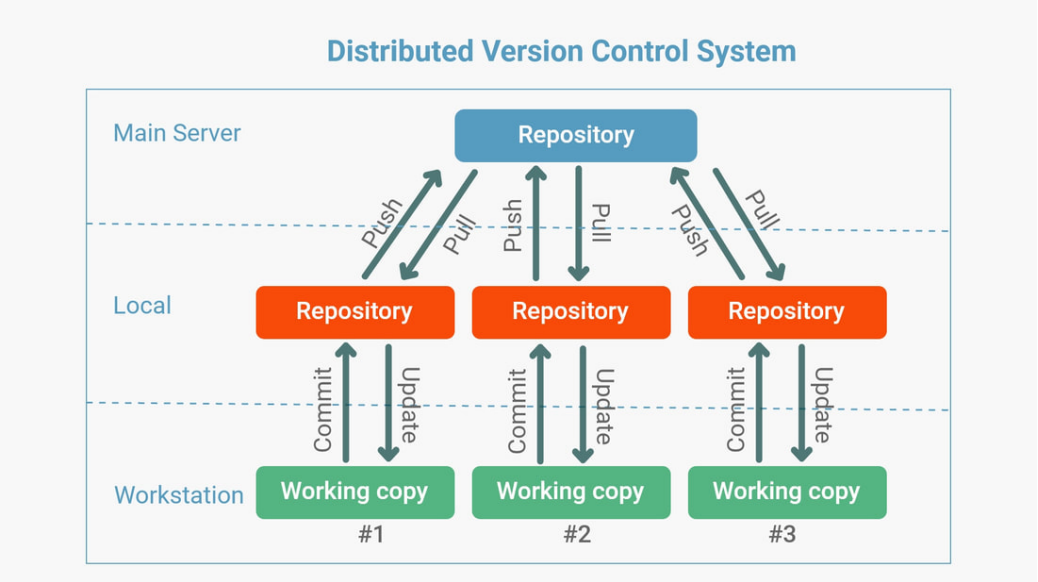

Figure 2.2: Distributed Version Control System

2.3 Existing Version Control Systems

2.3.1 Local Version Control Systems

Early VCS were intended to track changes for individual files and checked-out files could only be edited locally by one user at a time. They were built on the assumption that all users would log into the same shared Unix host with their own accounts, but it was not always possible. As you can imagine, these early systems made it easier for small teams to revisit code states from various points in history.

Source Code Control System (SCCS)

SCCS was released in 1972 and is one of the first successful VCS tools. It was written by Marc Rochkind at Bell Labs, who wanted to solve the problem of tracking file revisions. The tool made tracking down bugs introduced into a program significantly more manageable. SCCS is worth understanding at a basic level because it helped set up modern VCS tools that developers use today.

Architecture Much like modern VCS tools, SCCS has a set of commands that allow developers to work with versioning of files. The basic command functionality are:

- Check in files to track their history.
- Check out specific file versions for review.
- Check out specific file versions for editing.
- Check in new file versions with comments explaining the changes.
- Revert changes made to a checked out file.
- Basic branching and merging of changes.
- Print a log of a file's version history.

A special type of file called a `s-file` or a `history file` is created when a file is tracked by SCCS. This file is named with the original filename prefixed with a `s.`, and is stored in a subdirectory called `SCCS`.

So a file called `test.txt` would get a history file created in the `./SCCS/` directory with a name of `s.test.txt`. When created, the `s-file` contains the original file contents, a header that contains the file's version number, and some other metadata. There are also checksums stored in the `s-file` that are used to verify the integrity of the file (i.e. to make sure that the file has not been tampered with). The `s-file` content is not encoded or compressed in any way, which is a clear difference from modern VCS tools.

Since the content of the original file is now stored in the history file, it can be retrieved into the working directory for review, compilation, or editing. Further changes made to the file such as line additions, modifications, and removals can be checked back into the history file, which increments its revision number.

Subsequent SCCS checkins only store only the deltas or changes to a file as opposed to the entire file content each time. This decreases the size of the history file. Each time a checkin is made, the delta is stored in a structure known as a delta table inside the history file. As previously mentioned, the actual file content is more or less copied verbatim, with special control sequences for marking the start and end of sections of added and removed content. Since SCCS history files don't use compression, they are typically larger in size that the actual file being tracked. SCCS uses a delta method known as interleaved deltas. This is beneficial since it allows constant-time checkouts regardless of how old the checked out revision is - i.e. older revisions don't take longer to checkout than newer revisions.

One important thing to note is that all files are tracked and checked in separately in SCCS. There is no way to checkin changes to multiple files as a part of one atomic unit - like a commit in Git. Each tracked file has a

corresponding history file which stores its revision history. In general, this means that the version numbers of different files in a project will not usually match each other. However, matching revision numbers can be achieved by editing every file in the project at once (even if not all of the files have real changes) and checking them all at one time. This will increment the revision number for all the files to keep them consistent, but note that this is NOT the same as including multiple files in a single commit like in Git. In SCCS, this makes an individual checkin in each history file, as opposed to one big commit including all the changes at once.

When a file is checked out for editing in SCCS, a lock is placed on the file so it cannot be edited by anyone else. This prevents changes from being overwritten by other users, but also limits development since only one user can work with a given file at a time.

SCCS has support for branches that can store sequences of changes within a specific file. Branches can be merged back in with the original versions or merged with other branched versions of the same parent.

Basic Commands Below is a list of the most common SCCS commands.

`sccs create <filename.ext>` - Check in a new file to SCCS and create a new history file for it (in the `./SCCS/` directory by default).

`sccs get <filename.ext>` - Check out a file from from its corresponding history file and place it in the working directory in readonly mode.

`sccs edit <filename.ext>` - Check out a file from the corresponding history file for editing. Locks the history file so no other users can modify it.

`sccs delta <filename.ext>` - Check in the modifications to the specified file. Will prompt for a comment, store the changes in the history file, and remove the lock.

`sccs prt <filename.ext>` - Display the revision log for a tracked file.

`sccs diffs <filename.ext>` - Display the differences between the current working copy of a file and the state of the file when it was checked out.

Revision Control System (RCS)

RCS was released in 1982, and it is another early VCS tool. It was written in C by Walter Tichy as an alternative to SCCS, which was still closed source at the time. RCS was released under the GNU General Public License, which allowed it to be used in open source projects.

"RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging, and identification of revisions." - Walter Tichy

Architecture RCS shares many traits with its predecessor, including:

- Handling revisions on a file-by-file basis.
- Changes across multiple files can't be grouped together into an atomic commit.
- Tracked files are intended to be modified by one user at a time.
- No network functionality.
- Revisions for each tracked file are stored in a corresponding history file.
- Basic branching and merging of revisions within individual files.

When a file is set checked into RCS for the first time, a corresponding history file is created for it in the local `./RCS/` directory. This file is postfixed with a `,v` so a file named `test.txt` would be tracked by a file called `test.txt,v`.

RCS uses a `reverse-delta` scheme for storing file changes. When a file is checked in, a full snapshot of the file's content is stored in the history file. When the file is modified and checked in again, a delta is calculated based off of the existing history file content. The old snapshot is discarded and the new one is saved, along with the delta to get back to the older state.

This is called `reverse-delta` since to check out an older revision, RCS starts with the newest version of the file and applies consecutive deltas until the older revision is reached. This method allows for very quick checkouts of current revisions since the full snapshot of the current revision is always available. However, the older the checkout revision, the longer the checkout takes since an increasing number of deltas need to be calculated against the current snapshot.

This is not the case with SCCS which takes the same amount of time to fetch any revision. In addition, no checksum is stored in RCS history files so file integrity cannot be ensured.

Basic Commands Below is a list of the most common RCS commands.

2.3.2 Centralized Version Control Systems

Version control technology continued to evolve, leading to centralized repositories that contained the 'official' versions of their projects. This was good progress, since it allowed multiple users to checkout and work with the code at the same time as well as making commits back into this central repository. Furthermore, network access was required for people who wanted to commit changes they had made locally.

Concurrent Versions System (CVS)

Concurrent Versions System (CVS) was developed in by Dick Grune in 1986, it is also written in C. It was created with the goal of adding a networking element to version control. CVS became the first widely used VCS tool that allowed multiple users to work on the same project at the same time from different locations. This kicked off the second generation of VCS tools, allowing geographically dispersed development teams to work together on the same project.

Architecture CVS operates as a frontend for RCS - it provides a set of commands to interact with files in a project, but uses the RCS history file format and commands behind the scenes.

This meant that for the first time in history, multiple developers could check out and work on the same files simultaneously. It did this by utalising a centralized repository model.

The first step is to set up a centralized repository on a remote server using CVS. Projects can then be imported into the repository. When a project is imported into CVS, each file is converted into a `,v` history file and stored in a central directory known as a `module`. The repository generally lives on a remote server which is accessible over a local network or the Internet.

A developer checks out a copy the module which is copied to a working directory on their local machine. No files are locked in this process so there is no limit to the number of developers that can check out the module at one time. Developers can modify their checked out files and commit their changes as needed. If a developer commits a change, other developers will need to update their working copies via a (usually) automated merge process before committing their changes. Occasionally merge conflicts will need to be manually resolved before the commit can be made. CVS also provides the ability to create and merge branches.

Basic Commands Below is a list of the most common CVS commands.

Apache Subversion (SVN)

Subversion (SVN) was developed by CollabNet in 2000 and is now maintained by the Apache Software Foundation. It is written in C and was designed to be a more robust centralized solution than CVS.

Architecture Similar to CVS, SVN uses a centralized repository model. Remote users must have a working network connection to commit their

changes to the central repository.

Subversion introduced the functionality of atomic commits which ensured that a commit would either fully succeed, or be completely abandoned if an issue occurred.

In CVS, if a commit operation failed midway, for example due to a network outage, the repository could be left in a corrupted and inconsistent state. Furthermore, a commit or revision in Subversion can include multiple files and directories. This is important since it allows users to track sets of related changes together as a grouped unit, instead of the past storage models that track changes separately for each file.

The current storage model that Subversion uses for tracked files is called `FSFS` or `File System atop the File System`. This name was chosen since it creates its database structure using a file and directory structure that match the operating system filesystem it is running on. The unique feature of the Subversion filesystem is that it is designed to track not only the files and the directories it contains, but the different versions of these files and directories and they change over time. It is a filesystem with an added time dimension. In addition, folders are first class citizens in Subversion. Empty folders can be committed in Subversion, whereas in the rest (even Git) empty folders are unnoticed.

When a Subversion repository is created, a (nearly) empty database of files and folders is created as a part of it. A directory called `db/revs` is created in which all revision tracking information for the checked-in (committed) files is stored. Each commit (which can include changes to multiple files) is stored in a new file in the `revs` directory and is named with a sequential numeric identifier starting with 1. When a file is committed for the first time, its full content is stored. Future commits of the same file will store only the changes - also called the `diffs` or `deltas` - in order to conserve space. In addition, the deltas are compressed using `lz4` or `zlib` compression algorithms to further reduce their size.

By default, this is actually only true to a point. Although storing file deltas instead of the whole file each time does save on storage space, it adds time to checkout and commit operations since all the deltas need to be strung together in order to recreate the current state of the file. For this reason, by default Subversion stores up to 1023 deltas per file before storing a new full copy of the file. This achieves a nice balance of both storage and speed.

SVN does not use a conventional branching and tagging system. A normal Subversion repository layout is to have three folders in the root:

- `trunk/`
- `branches/`

- `tags/`

The `trunk/` folder is used for the production version of the application. The `branches/` folder is used to store subfolders that correspond to individual branches. The `tags/` folder is used to store tags which represent specific (usually significant) project revisions.

Basic Commands Below is a list of the most common SVN commands.

Perforce Helix Core

Perforce Helix Core is a proprietary VCS created, owned, and maintained by Perforce Software, Inc. It is typically set up using a centralized model although it does offer a distributed model option. It is written in C and C++, and was initially released in 1995. It is primarily used by large companies that track and store a lot of content using large binary files, as is the case in the video game development industry. Although Helix Core is typically cost prohibitive for smaller projects, Perforce offers a free version for teams of up to 5 developers.

Architecture Perforce Helix Core is set up as a server/client model. The server is a process called `p4d` which waits and listens for incoming client connections on a designated port, typically port 1666. The client is a process called `p4` which comes in both command-line and GUI flavors. Users run the `p4` client to connect to the server and issue commands to it. Support is available for various programming language APIs, including Python and Java. This allows automated issuance and processing of Helix Core commands via scripts. Integrations are also available for IDEs like Eclipse and Visual Studio, allowing users to work with version control from within those tools.

The Helix Core Server manages repositories referred to as depots, which store files in directory trees, not unlike Subversion (SVN). Clients can check-out sets of files and directories from the depots into local copies called **workspaces**. The atomic unit used to group and track changes in Helix Core depots is called the **changelist**. Changelists are similar to Git commits. Helix Core implements two similar forms of branching - **branches** and **streams**. Branches are conceptually similar to what we are used to - separate lines of development history. A stream is a branch with added functionality that Helix Core uses to provide recommendations on best merging practices throughout the development process.

When a file is added for tracking, Helix Core classifies it using a **file type** label. The two most commonly used file types are text and binary. For binary files, the entire file content is stored each time the file is stored. This is a common VCS tactic for dealing with binary files which are not amenable to the normal merge process, since manual conflict resolution is usually not possible.

For text files, only the deltas (changes between revisions) are stored. Text file history and deltas are stored using the RCS (Revision Control System) format, which tracks each file in a corresponding `,v` file in the server depot. This is similar to CVS, which also leverages RCS file formats for preserving revision history. Files are often compressed using gzip when added to the depot and decompressed when synced back to the workspace.

Basic Commands Below is a list of the most common Perforce commands.

2.3.3 Distributed Version Control Systems

In a distributed version control system, all copies of the repository are created equal - there is no central copy of the repository. This design principle encourages commits, branches and merges to be created locally without network access and pushed to other repositories as needed.

Git

Git was created in 2005 by Linus Torvalds (also the creator of Linux) and is written primarily in C combined with some shell scripts. It is widely considered the best VCS tool due to its features, flexibility, and speed. Linus Torvalds originally wrote it for the Linux codebase and it has grown to become the most popular VCS in use today.

"You can do a lot of things with Git, and many of the rules of what you should do are not so much technical limitations but are about what works well when working together with other people. So Git is a very powerful set of tools, and that can not only be overwhelming at first, it also means that you can often do the same (or similar) things different ways, and they all 'work.'" - Linus Torvalds

Git repositories are commonly hosted on local servers as well as cloud services. Git forms the backbone of a broad set of DevOps tools available from popular service providers including GitHub, BitBucket, GitLab, and many others.

Architecture Git is a distributed VCS. This means that no copy of the repository needs to be designated as the centralized copy - all copies are created equal. This is in stark contrast to the second generation VCS which rely on a centralized copy for users to checkin and checkout from.

What this means is that developers and coding partners can share changes with each other directly before merging their changes into an official branch. This allows team collaboration to take on a flexible distributed workflow, if desired.

Furthermore, developers can commit their changes to their local copy of the repository without any other repositories knowing about it. This means that commits can be made without any network or Internet connection. Developers can work locally offline until they are ready to share their work with others. At that point, the changes can be pushed to other repositories for review, testing, or deployment.

When a file is added for tracking with Git, it is compressed using the `zlib` compression algorithm. The result is hashed using a SHA-1 hash function. This yields a unique hash value that corresponds specifically to the content in that file. Git stores this in an `object database` which is located in the hidden `.git/objects` folder.

The name of the file is the generated hash value, and the file contains the compressed content. These files are called Git blobs and are created each time a new file (or changed version of an existing file) are added to the repository.

"The object database is literally just a content-addressable collection of objects. All objects are named by their content, which is approximated by the SHA1 hash of the object itself. Objects may refer to other objects (by referencing their SHA1 hash), and so you can build up a hierarchy of objects." - Linus Torvalds

Git implements a `staging index` which acts as an intermediate area for changes that are getting ready to be committed. As new changes are staged for commit, their compressed contents are referenced in a special index file - which takes the form of a `tree` object. A `tree` is a Git object that connects blob objects to their real file names, file permissions and links to other trees, and in this way represents the state of a particular set of files and directories. Once all related changes are staged for commit, the index tree can be committed to the repository, which creates a `commit` object in the Git object database.

A commit references the head tree for a particular revision as well as the commit author, email address, date, and a descriptive commit message. Each commit also stores a reference to its parent commit(s) and so over time a history of project development is established.

As mentioned, all Git objects - blobs, trees, and commits - are compressed, hashed, and stored in the object database based on their hash value. These are called `loose objects`. At this point no diffs have been utilized to save space which makes Git very fast since the full content of each file revision is accessible as a loose object.

However, certain operations such as pushing commits to a remote repository, storing too many objects, or manually running Git's garbage collection command can cause Git to repackage the objects into `pack files`. In the packing process, reverse diffs are taken and compressed to eliminate redundant content and reduce size. This process results in `.pack` files containing the object content, each with a corresponding `.idx` (or index) file containing a reference of the packed objects and their locations in the pack file.

These pack files are transferred over the network when branches are pushed to or pulled from remote repositories. When pulling or fetching branches, the pack files are unpacked to create the loose objects in the object repository.

Basic Commands

- `git init` - Initialize a Git repository in the current directory (creates the hidden `.git` folder and its contents).
- `git clone <git-url>` - Download a copy of the Git repository at the specified URL.
- `git add <filename.ext>` - Add an untracked file or changed file to the staging area (creates corresponding entries in the object database).
- `git commit -m 'Commit message'` - Commit a set of changed files and folders along with a descriptive commit message.
- `git status` - Show information related to the state of the working directory, current branch, untracked files, modified files, etc.
- `git branch <new-branch>` - Create a new branch based on the current checked-out branch.
- `git checkout <branch>` - Checkout the specified branch into the working directory.
- `git merge <branch>` - Merge the specified branch into the current branch checked-out in the working directory.

- `git pull` - Update the working copy by merging in committed changes that exist in the remote repository but not the working copy.
- `git push` - Pack loose objects for local active branch commits into pack files and transfer to remote repository.
- `git log` - Show the commit history and associated descriptive messages for the active branch.
- `git stash` - Save all uncommitted changes in the working directory to a cache so that they can be retrieved later.

Mercurial

Mercurial was created in 2005 by Matt Mackall and it is written in Python. It was also started with the goal of hosting the codebase for Linux, but Git was chosen instead. It is the second most popular distributed VCS after Git, but is used far less often.

Architecture Like Git, Mercurial is a distributed version control system that allows any number of developers to work with their own copy of a project independently from others.

Mercurial leverages many of the same technologies as Git, such as compression and SHA-1 hashing, but does so in different ways.

When a new file is committed for tracking in Mercurial, a corresponding `revlog` file is created for it in the hidden directory `.hg/store/data/`. You can think of a `revlog` (or revision log) file as a modernized version of the `history files` used by the older VCS like CVS, RCS, and SCCS.

Unlike Git, which creates a new blob for every version of every staged file, Mercurial simply creates a new entry in the revlog for that file. To conserve space, each new entry only contains the delta (changes) from the previous version. Once a threshold number of deltas is reached, a full snapshot of the file is stored again.

This reduces the lookup time when applying many deltas to reconstruct a particular file revision.

These file revlogs are named to match the files that they track, but are postfixed with `.i` and `.d` extensions. The `.d` files contained the compressed delta content. The `.i` files are used as indexes to quickly track down different revisions inside the `.d` files. For small files with low numbers of revisions, both the indexes and content are stored in `.i` files. Revlog file entries are compressed for performance and hashed for identification. The hash values are referred to as `nodeids`.

Whenever a new commit is made, Mercurial tracks the all file revisions in that commit in something called the `manifest`. The manifest is also a revlog file - it stores entries that correspond to particular states of the repository.

However, instead of storing individual file content like the file revlogs, the manifest stores a list of filenames and nodeids that specify which file revision entries exist in each revision of the project. These manifest entries are also compressed and hashed. The hash values are again referred to as `nodeids`.

Lastly, Mercurial uses one more type of revlog called a `changelog`. The changelog contains a list of entries that associate each commit with the following information:

- Manifest nodeid - Identifies the full set of file revisions that exist at a particular time.
- Parent commit nodeid(s) - This allows Mercurial to establish a timeline or branch of project history. One or two parent ID's are stored depending on the type of commit (normal vs merge).
- Commit author
- Commit date

Each changelog entry also generates a hash known as its `nodeid`.

Basic Commands

- `hg init` - Initialize the current directory as a Mercurial repository (creates the hidden `.hg` folder and its contents).
- `hg clone <hg-url>` - Download a copy of the Mercurial repository at the specified URL.
- `hg add <filename.ext>` - Add a new file for revision tracking.
- `hg commit -m 'Commit message'` - Commit a set of changed files and folders along with a descriptive commit message.
- `hg status` - Show information related to the state of the working directory, untracked files, modified files, etc.
- `hg update <revision>` - Checkout the specified branch into the working directory.
- `hg merge <branch>` - Merge the specified branch into the current branch checked out in the working directory.

- `hg pull` - Download new revisions from remote repository but don't merge them into the working directory.
- `hg push` - Transfer new revisions to remote repository.
- `hg log` - Show the commit history and associated descriptive messages for the active branch.

BitKeeper

Advantages

Disadvantages

Darcs Advanced Revision Control System (Darcs)

Advantages

Disadvantages

Monotone

Advantages

Disadvantages

Bazaar

Advantages

Disadvantages

Fossil

Advantages

Disadvantages

2.4 Summary of Key Features

2.4.1 Repositories

2.4.2 Commits

2.4.3 Branching and Merging

2.4.4 Pulling and Pushing

Chapter 3

Design

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusion