Version Control

Group D

ENSE 375

Professor Mohamed El-Darieby

February 22, 2021

University of Regina

Jacob Chapman - 200386601
Nolan Flegel - 200250037
Krupalkumar Patel - 200385434
Renz Rivero - 200377174
Shane Toma - 200396087

1.0 Introduction:

Version control software (VCS) is a family of programs whose goal is to provide a structured way to save and distribute iterations of a software project. The use of a VCS can improve ease of collaboration, efficiency in workflow, and recovery from bugs and security threats. Collaboration is made easier by granting all contributors the freedom to view and download the entire project, by seeing how their specific work fits in with the bigger picture motivation tends to be higher and the work produced tends to fit better. Workflow efficiency is improved through a combination of better fitting code and the concept of conflict resolution and automated merging of code. Advantages in recovery from bugs and security issues come from the fact that it is so easy to go back a version if something opens up an exploit or breaks something in the code. VCS' have a lot of upside and should be considered for use in most projects.

Git is a popular VCS which uses concurrent versioning (CS) to accomplish its purpose. There are a great deal of aspects which are valuable to examine to both understand the applicability of VCS' and Git, these include the history of source control, the toolkit, branching, merging, conflict resolution, and extensibility. This paper will discuss these topics and more in detail to present the functionality of Git as well as VCS concepts.

2.0 History of Source Control

Source control originated in 1975 with a system called Source Code Control System (SCCS). The objective was to provide a simplistic method of saving source code files without creating many copies of the same file. It worked by saving the changes made to the file, also known as Deltas. In 1982 a new system was developed, known as the Revision Control System (RCS). It was an evolution of the SCCS, adding a few more useful features. RCS was free and maintained by the GNU project, this led to it becoming a popular alternative to SCCS.

While useful, these early source control systems had their limitations. Only one developer could work on a file at the same time, publishing changes required a strict hierarchy of versions and required a shared file system. In 1986 a new system was developed to address these issues, the Concurrent Versioning System (CVS). It brought forth a host of new changes, primarily the ability for developers to simultaneously edit the same file and a mechanism to resolve differences between edits. CVS also introduced the idea of code branching, allowing developers to work on multiple tasks at the same time. Lastly, CVS allowed the code repository to work across large networks using a client-server architecture.

These early systems created the foundations for some of the latest and current systems like Subversion, Git and Mercurial.

The first of the modern systems was Subversion, created in 2000. Subversion sought to solve existing problems in CVS by creating a new control system that focused on entire code branches and local copies of the repository. Like its predecessors, Subversion maintained a centralized repository, where a central computer tracks all of the code changes. Following Subversion in 2005 would be Mercurial and Git, created by linux developers Matt Mackall and Linus Tovalds, respectively.

Modern source control systems follow a typical process of getting the latest version of files, make changes to that version and publish the changes so other developers can access them. While these new systems share features of their predecessors, they have continued to refine the process and solved many problems that early code control systems encountered. Both platforms sought to solve the issue of distributed version control where temporal tracking of changes is not ideal.

3.0 Distribution

Centralized systems require network connections to the repository server for any code changes to be made. This method ensures that no changes are lost, however in large organizations this can make accessing the repository slow. Centralized systems have difficulty tracking merges because committing and publishing code changes are inherently the same process with this system. The benefit of controlling the source code with a single server is a simplified integration process and a strict hierarchy of revisions. While Subversion uses a centralized system for version control, Git and Mercurial have implemented a distributed version control method as a way to address the problems with centralized systems.

In distributed systems, developers can work on their local system and maintain separate code repositories. They do not require an active connection to a central repository. This key difference allows developers to incrementally commit their work locally until they are ready to share their changes with others. As a result, accessing and making edits to code is much faster and merging files is easier to track. These systems can manage a significantly larger number of changes and files than existing centralized systems. This method of simultaneous changes makes it impossible to track revisions temporally. The directed graph in Figure 1, illustrates their solution where changes become children of the revision it was based on.

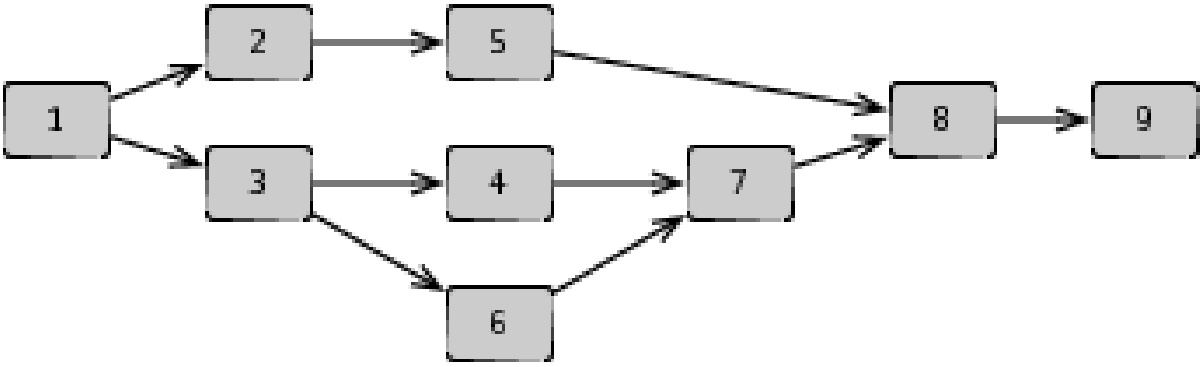

Figure 1: Directed Graph of code branching

Note. Mercurial. *Figure 12.1.* From Distributed Version Control

4.0 Conflict Resolution

Every modern Version Control System (VCS) has an essential feature that allows multiple developers to work on the same project, at the same time, without conflicting with each other. This feature is implemented by allowing multiple developers to program on a branch locally and giving them the ability to push their changes, once ready, to a central place. These changes can then be pulled by other members in the project back to their own local copy and continue their own work with their collaborators' changes in place.

Bringing changes from one branch into another desired branch is done using a merge command. This command will vary depending on the VCS used and it basically collects the changes in all the other branches and applies them over to the current branch. In situations where changes from multiple developers to a file overlaps, the system will have difficulties in processing which version or changes to correctly apply. This is called a merge conflict and when this occurs, it can take a minute or even days to resolve, depending on the amount of files needing to be fixed.

There is little risk to permanent changes as merges can be undone and reverted to a state before the conflict occurred. In Git, the process is simple and users can input the command "git merge --abort" in their command line terminal. To resolve a conflict and go through the merge process, developers can take advantage of Git's mergetool. This tool gives developers a visual representation of the code differences on the command line in the following format:

- LOCAL - the file from the current branch

- BASE - the file before changes in the local and remote branch
- REMOTE - the file to be merged into the local branch
- MERGED - the merge result to be saved in the repository

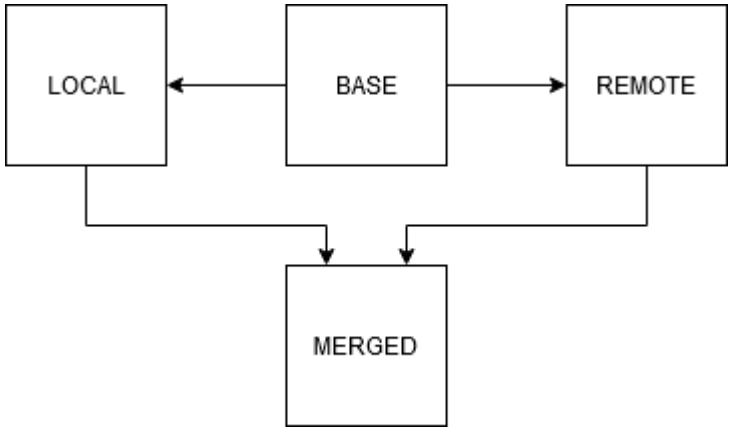

Figure 2: Merging files from different sources

As shown in Figure 2, the tool allows a developer to search line by line the differences in changes between the local and remote branch as well as to easily choose which changes to apply to the end result. After cleaning up the conflict in the files line by line, all that is left is to save and do a regular commit.

5.0 Version History

Version control is a system that tracks and records changes to a set of files over time so developers can reference and recall any version. It is important for developers to have a version control system in place. It allows users to revert files back to a previous state, or an entire project. It allows for the use of tracking changes over time, see which user has last modified files, and many other useful features. This is crucial because if a developer has created a problem in a recent file, they can revert back to a previous version and fix the issue. Using version control is an essential tool for any developer or organization to prevent unintended changes or code deletions.

There are many developers who choose to have a local version control system, which is to copy files from a remote repository to a local directory. This is a common practice because it allows users to make many small changes without publishing unfinished code to others. There are many errors associated with this method; it can be complicated to find where files are located, it is easy to overwrite files, accidently delete files, or move files to places where they shouldn’t be. Without version control, when a file is saved the previous version is altered and it is difficult to recover previous versions file, unless copies of every previous version were made. Using a version control system solves this issue by tracking

changes and managing versions for the developer. Figure 3 shown below displays how version control works locally.

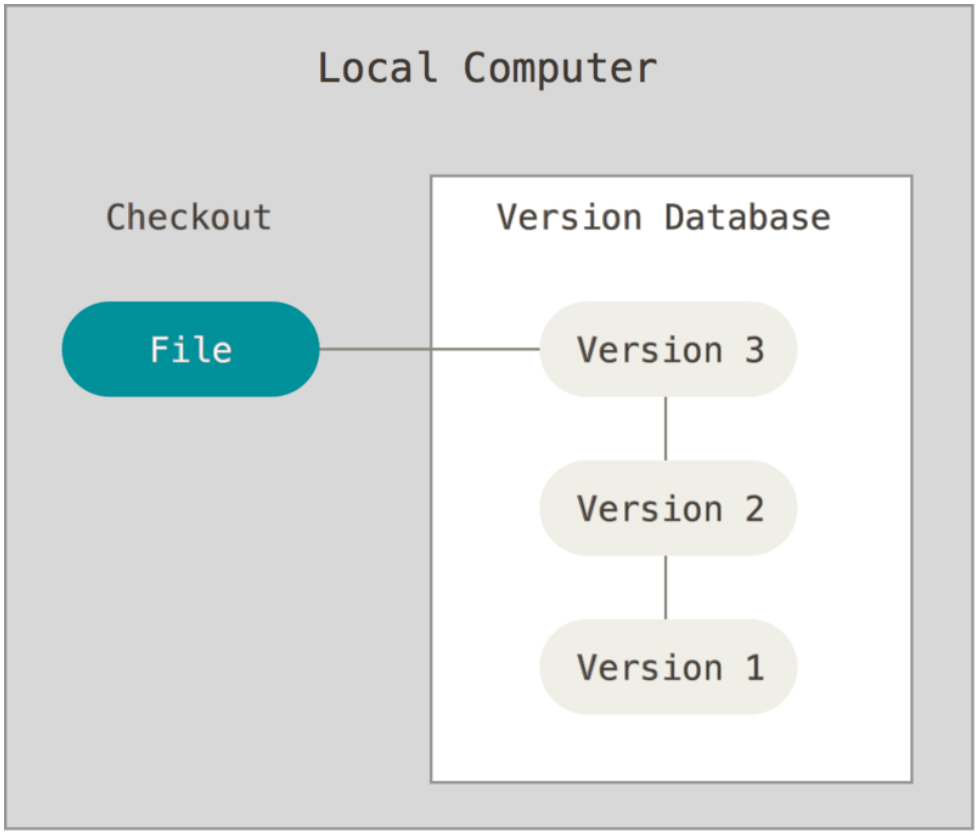

Figure 3: Local version control

Note. Git. *Figure 1.* From Local Version Control Systems

In industry however, having local files in directories and sharing with a team is a problem. One solution would be to email files, which has many faults associated. To combat this, an elegant solution is to have a Centralized Version Control (CVC) set in place. A CVC system is positioned on a server, where versions of files are stored, tracked and maintained. This is common practice in industry, and has been the standard for many years. Figure 4 shown below highlights how a CVC operates.

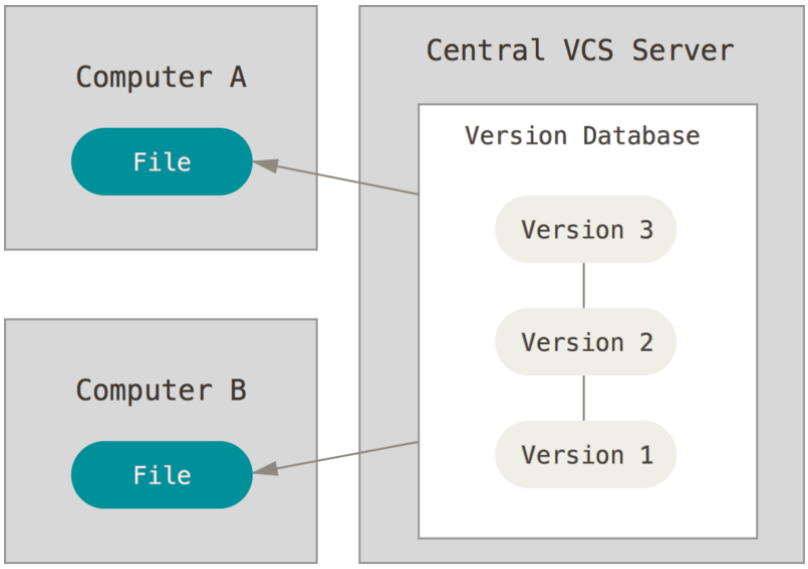

Figure 4: Centralized Version Control

Note. Git. *Figure 2.* From Centralized Version Control Systems.

This setup offers many advantages, for example everyone collaborating on a project has to some degree what is going on. There is little overhead, and administrators have control over who can do what. However, there are some faults with this. One notably being that developers rely on a single server. If the server goes down for some amount of time, no work can get done until it gets back up.

5.1 Commit and Merge Histories

Most VCS software utilize one of the following approaches to track history; Linear history, or an Directed Acyclic Graph (DAG). Git uses a DAG to store its history. Each time there is a commit, the metadata knows about its ancestors. A commit in Git can have zero to many parent commits. Figure 5 shown below shows how Git uses a DAG as its history structure.

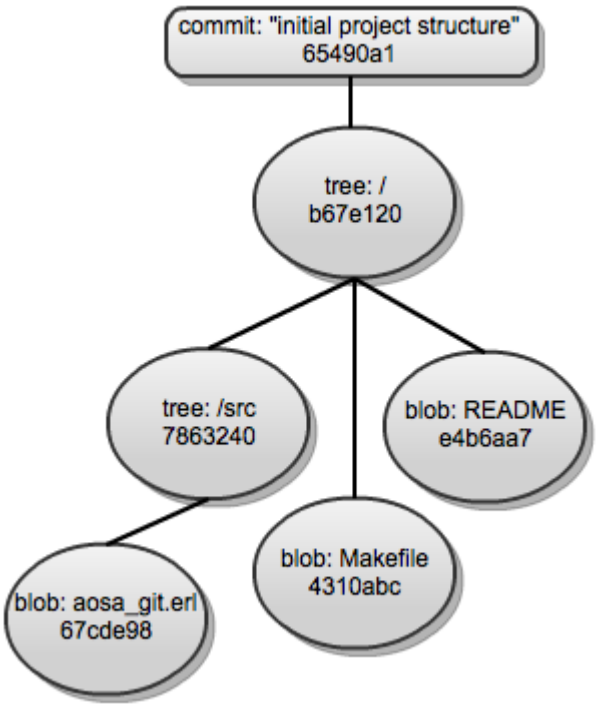

Figure 5: DAG Representation example in Git
Note. Git. *Figure 6.1*. From Commit and Merge Histories.

Git enables full branching capabilities using DAG's to store content. The history of a file is linked all the way to the directory structure to the root directory. When merging two branches, you merge the contents of the 2 nodes in a DAG. The DAG allows Git to efficiently see common ancestors.

6.0 The Toolkit

Inside Git, there are many command-line and UI tools on various operating systems (including Windows). Most of these tools are built on top of the Git core toolkit. The Git toolkit is composed of two parts which are plumbing and porcelain. The plumbing consists of low-level commands that allow for basic content tracking and manipulation of the DAG's. The porcelain is the smaller subset of Git commands that most Git users use for maintaining repositories and communicating between repositories.

Git was originally created for a VCS, rather than a user-friendly VCS. Therefore there are lots of commands that are not user-friendly, which are the plumbing commands. The more user-friendly commands are the porcelain commands.

Examples of Git plumbing commands are:

- Git-apply
- Git-merge-file
- Git-commit-tree
- Git-read-tree
- Git-merge-base

Some examples of user-friendly Git porcelain commands are:

- Git-add
- Git-pull
- Git-push
- Git-init
- Git-status
- Git-help

7.0 Branching and Merging

Branching and merging form the logical basis of concurrent versioning. While version control systems can be made without concurrent versioning features, collaboration suffers heavily without these features. Concurrent versioning systems allow for easy collaboration between multiple parties and gives a way to take a project in multiple directions at the same time, so it is possible to have a release version, development version, and experimental version all in the works in one easy to use version control system. Concurrent versioning is one of the primary advantages of using Git, branching and merging are used to provide structure to this flexible version control system.

Git uses a directed acyclic graph (DAG) to store its commit and merge history. This means that every commit and merge can be thought of as a node which holds metadata about its parents. For example a merge of two branches will produce a node (which really represents a directory) with two known parents while a simple commit will typically only have one known parent. Git's use of DAG's makes it so history is tracked all the way to the root directory (or first node), this history is linked to every relevant node as commits are added. This makes Git effective for finding common ancestors between nodes which is helpful in merging.

Branching is the first applied concept that is used for concurrent versioning. Every branch can be forked so multiple edits of a primary branch can happen concurrently, this results in large improvements in efficiency since all contributors can see what has been committed previously. For the developer to know what branch they are working on Git makes use of a pointer called the “HEAD” pointer which points at the branch currently in use. To create a new branch the command “git branch <newBranchName>” must be used, “git branch” with no name included will show all the branches of the project as well as which branch the HEAD pointer is pointing to. The HEAD pointer will not move to a newly created branch unless explicitly commanded to do so, to do this the command “git checkout <branchName>” must be used. Once the HEAD pointer is set to point to a new branch, all commits will be performed on said branch. The diagram below shows a project branching. At times branches will cease being of any use to the project, rather than keeping a collection of dead branches to clutter up the project directories it is possible to delete branches. “Git branch -d <branchName>” allows branches to be deleted, it is important to ensure no useful work is being deleted as the delete cannot always be undone.

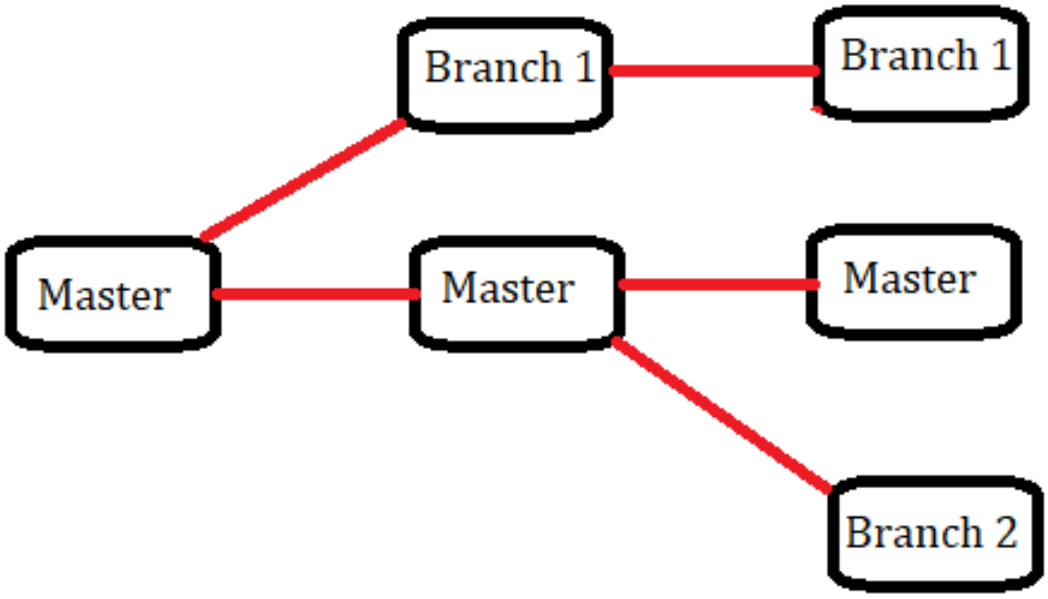

Figure 6: Diagram Showing Branching from Master into Branch 1 and Branch 2

Branching without the ability to merge is not particularly useful. Merging involves combining multiple branches into a single harmonized version of the project, this entails including all changes made to different files in all branches being merged, not including duplicate work, and using conflict resolution strategies for edits which are incompatible. Edits may be incompatible for a number of reasons, commonly it is because two branches edited the same piece of code in different ways so it must be determined which version should be used. Merging patterns include fast-forward merging and three-way merging. Fast

forward merging is when the master branch is left behind by a different branch, by merging the branch which is ahead with the master branch what actually happens is the master “fast forwards” or catches up to the node at the end of the branch being committed. In short, the branch being merged becomes the same as the master branch (see diagram below). Three-way merging is used when parallel branches need to be combined into a single branch (see diagram below). The command for performing a merge is “git merge <branchName>”, Git will automatically use the appropriate merging strategy based on the state of the directory tree (branches).

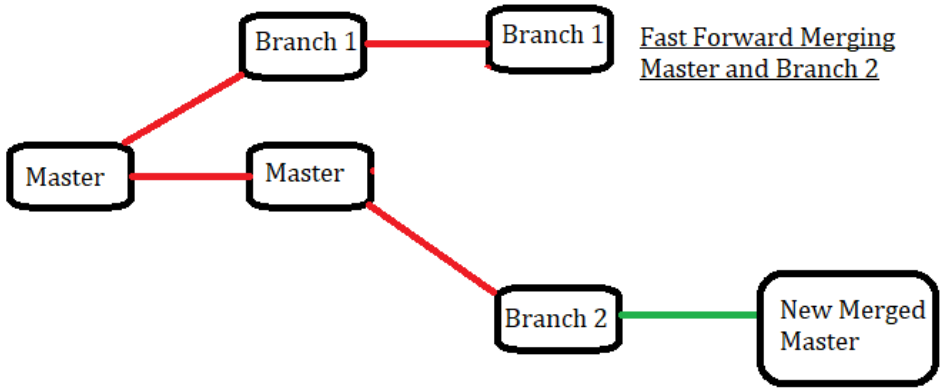

Figure 7: Diagram Representing Fast-Forward Merge

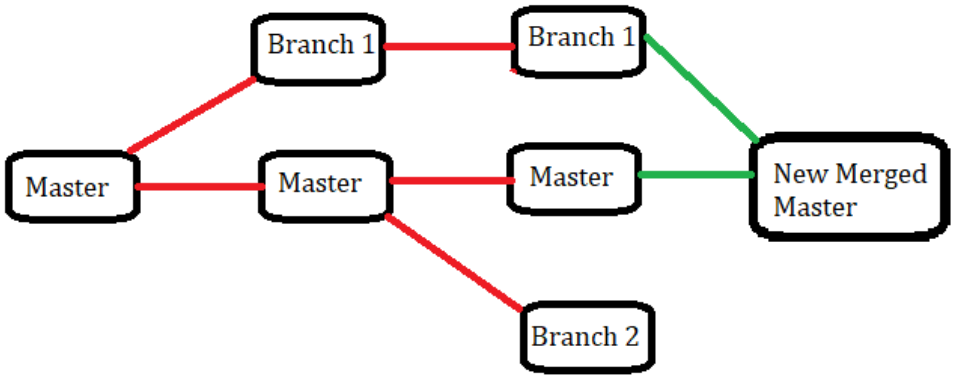

Figure 8: Diagram Representing Three-way Merge

8.0 Extensibility

The term extensibility means adding changes to the existing system. Both Mercurial and Git functionality can be expanded through extensions. Mercurial's extensions are written in Python for accessibility and ease of development, however developers are required to use the Mercurial API when creating extensions. These extensions are provided primarily in the form of modules. They are a powerful tool for users who want advanced functionality.

Git extensions are not restricted to Python, it is possible to use any programming language to extend Git. Developers can customize Git and create many new unique features. While this can be a powerful advantage, it also creates a complex extension library that is difficult to maintain. This fragmentation in the platform also means there are multiple methods of installing extensions.

Hooks are additional features for extensions, commonly used for continuous integration notifications and responses. Other hooks can trigger scripts and modules that allow for more complex behavior for the version control system. Hooks are frequently created for predetermined actions and can be used to control events and stop processes. Hooks have also been used to create logs whenever a set of actions occur and record user information and key details.

9.0 Comparison of Use Cases

Git and Mercurial share many of the same features and qualities. These versioning systems were created around the same time, for the same purpose: managing the linux kernel repository. Both systems offer distributed repositories, code branching, version

histories, extensibility and both use a directed graph data structure. The differences between Git and Mercurial begin with how these features were implemented and the overall popularity of the platforms.

Mercurial is easier to learn, the commands are simple and only have one purpose. The embedded documentation is clear and comprehensive, developers are able to easily find commands and understand how to use them. Mercurial's history is permanent and cannot be edited, this prevents any accidental deletions.

Git commands are more complex and offer greater flexibility. Individual commands have additional options that provide different functionality to the base command. As a result Git is more difficult to learn, however leads to more features for developers. Git's version history differs from Mercurial by allowing developers to edit previous commits and reorder the version history. This functionality provides developers with greater control over the repository while enabling the possibility for unintentional deletions. Recovery from deletions is possible in Git as all changes are tracked, however requires the use of an additional command. The increased functionality of Git also increases the complexity. It is important for developers to have a greater understanding of the commands because misuse can have unintended consequences.

Git has been widely adopted by developers and organizations. While Mercurial has less mainstream recognition, it is regularly maintained and used by many organizations. In recent years, Git has become the industry standard version control system. As a result there are more extensions and plugins available and more tutorials and support for Git. Both platforms are powerful, modern version control systems that offer complete tool sets for developers and organizations. Git is more accessible due to wider adoption and offers a robust set of features and greater control to developers.

References

Apple Juice Teaching. (2014, July 12). *Git Guide: How to Finally Resolve Merge Conflicts!*. YouTube. https://www.youtube.com/watch?v=CKAdoAR0ykc

Breniser (Red Hat), B. (n.d.). *How to resolve a git merge conflict*. https://opensource.com/article/20/4/git-merge-conflict

Git. 1.1 *Getting Started - About Version Control*. (n.d.). http://www.git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Mahler, D. (2017, June 21). *Introduction to Git - Core Concepts.* YouTube. https://www.youtube.com/watch?v=uR6G2v_WsRA

Mahler, D. (2017, September 18). *Introduction to Git - Branching and Merging*. YouTube. https://www.youtube.com/watch?v=FyAAIHHClqI

Mahler, D. (2018, March 27). *Introduction to Git - Remotes*. YouTube. https://www.youtube.com/watch?v=Gg4bLk8cGNo

Ochtman, D. (n.d.). *Mercurial*. http://aosabook.org/en/mercurial.html

Potter, S. (n.d.). *Git*. http://aosabook.org/en/git.html

Waatz, A. (2020, April 14). *How to Add Collaborator to Repository in Github 2020*. YouTube. https://www.youtube.com/watch?v=p49LRx3hYI8