ENSE375 Project

Introduction to Git - Branching and Merging

**Branch**

Branch will allow us to work in the different versions of a file parallelly. Our work is independent, and we can determine if we want to merge our work into other branches. Separate versions of the same file are functions of the branch. We can branch for different purposes.

**Branch Implementation**

A commit has 40-hexadecimal sha-1 hash. Git will create a master branch by default. A branch is a pointer, and a branch will point to the sha-1 hash. When we are at our master branch every time, we make a commit the branch will move up. Git will know what branch we are at by a pointer called head. Head points to a branch and it is called a symbolic pointer because it only points to a branch.

**Creating Branch**

We can make a new branch with the command “git branch <name of the branch>”. The Git branch command will show all the branches. The asterisk with the branch will tells us what branch we at for example we are at the master branch, it tells us that “HEAD” is to the master branch. We switch to another branch with the command “git checkout <name of the branch>”. If we are to have any new commit to the branch, we will be at the new commit. If we had a file and we were editing that file for each branch, then the file and the information it contains will be different. We can create a branch and checkout a new branch with the command “git checkout -b dev”.

**Fast Forward Merge**

We can merge with the command “git merge <branch name>”. If there is a “direct path” we can use the Fast Forward Merge.

**Delete Branches**

Before deleting the branches, we can check what commits we have merged with “git branch –merged”. We can delete a branch with the command “git branch -d <command name>”. If the branch is not merged, Git will warn us before deleting a branch. If we still want to delete a branch that is not been merged, we can with “git branch -D”.

**3 Way Merge**

When we do not have a direct path, we can merge with 3 Way Merge. For example, if we had three branches, to merge we need to create a new commit called the merge commit. We need to first do a base commit and then do a last commit for each branch.

**Merge Conflict**

Merge conflict will occur when same line on different branches has changed. These branches than merge resulting in a merge conflict. Whenever there is a merge conflict, Git gives us a solution with the command “git merge –abort”. If we do not want to deal with the resolving the conflict this allows us to back out of the merge. We can start the merge with the command “git merge dev”. We can also type command “git status”, this will help us merge, it shows the working directory and the staging area. From here we can edit all the files that cause the conflict. After resolving we can carry forward with our merge commands and other commands.

**Detached Head**

When the head is pointing to the commit instead of the branch and then a commit it is called a “Detached Head”. If we are to type the command “git log” it will show the command history. Now from here we can take a hash. Now when we type the command “git checkout <commit hash>”, now Git will give us a warning that we are in “Detached Head” state. Now to resolve this we can checkout the same branch again. Let us go back to the “Detached Head” solve this with the command “git branch stage”, this creates a stage branch. Then type in the command “git checkout stage”, this will check out our new branch. We have again resolved the “Detached Head”.

**Git Stash**

To get to a clean state we can use the “Git Stash”, what “Git Stash” allows you to do is record the working directory and index of the current state and allows you to go to clean state. The command saves your work directory and will allow you to go back to a working directory. We can look at the stashes with the command “git stash list”. We can apply the stashed again at any time with the command “git stash apply” and this will apply the most recent stash. If we want to apply the “Git Stash” and remove it, we can type the command “git stash pop”. If we want to apply the different stash and not the most recent stash, we can apply the command “git stash apply <label name>. If want a to write information about what the stash contains, we can type the command “git stash save <information>”.

Chart

Description automatically generated

Branch

Diagram

Description automatically generated

Merge

Git - Susan Potter

Git allows to sustain work that is [usually code and could be other types of files] done by more than one person. Git is a distributed workflow tool, and it allows for work to converge or diverge. This report will talk about Git and how it is different from its competitors. This report will go into details about the goals behind invention of Git and how the goals are sustained and achieved. The report will elaborate how this DAGs [Directed Acyclic Graphs], Reference Pointer, Object Model Representation, remote protocol, and tree merging.

Git was started in the Linux Kernel in 2005. The Linux Kernel before the Git had high volume of committers. There was also large volume of contributor involvement. The Linux Kernel was managed with tarballs and patches and the development community were looking for a VCS that would be helpful. This gave a rise to Git in 2005.

Before Git Linux Kernel codebase was managed by two VCSs

* BitKeeper
* CVS

BitKeeper was quite different from the other VCS at that time. BitMover, the creators of BitKeeper decided to end the licenses of Key Linux Kernel Developers. Because of this Linus Torvald began developing the Git. He started with writing scripts to manage email patches. The goal of the scripts was to “abort merges” (Susan Potter, n.d.) instantly. This allowed the “maintainer” to modify the code “mid-path-stream” and do a manual merge. After this there is continuous merge of following patches.

Torvalds has three goals with Git

1. Distributed Workflow
2. Content Corruption Safety
3. High Performance

Even though Git is influenced by BitKeeper, Git is quite different and more powerful with more distributed and local workflows. BitKeeper was not able to do this. Monotone also influenced Git and it was VCS created in 2003.

Distributed Version Control System [dVCS] allow flexibility with workflow. The advantages of the distributed model are

* Collaborators can work offline.
* Collaborators has power to share their work when they think its ready to be shared
* Collaborators can look at the repository history
* Managed work can be published to more than one repository

When Git was developed there were also other dVCS. All this tool allows for flexible workflow and that VCS were not able to do. Some of this tool include Mercurial and SVK [subversion]. Subversion handles server to server synchronization. It can sustain by different developers. In today’s market there are many famous dVCS this includes Git, Fossil, Bazaar, Darcs, Mercurial, and Veracity.

Mercurial - DirkJan Ochtman

This chapter will discuss the Mercurial. Mercurial is a distributed version control system written in python with bits and small amount was return in C for performance. This chapter will discuss the decision related to the design of Mercurial Algorithm and data structures.

**12.1. A Short History of Version Control**

Version control systems allow developers to work software system continuously without

* Passing Full copies of work
* Tracking File Changes

One of the main functions of the VCS [Version Control Systems] is pass information/changes to a tree of files.

Basic Cycle

* Get tree of files
* Work on the changes related to this tree file
* Publish and allow other to access

Getting the tree of files is called a checkout. Repository is where we access and publish our work/changes. Working directory is the result of the checkout. Working directory could also be called working tree and working copy. Update occurs when we update a “working copy” (Dirkjan Ochtman, n.d.) with new files from the repository. Sometimes we will have to merge, this is when we combine work/changes of different users into a single life. The “diff” command allows us to check the changes/work of two revisions for a tree and a file. The common mode of the diff command is to observe at the unpublished local changes/work in the working copy. The work/changes can be published by the commit command. The commit command will save the changes/work that are in the working directory to the repository.

**12.1.1. Centralized Version Control**

In 1975 the first VCS [Version Control System] was explained and it was called SCCS [Source Code Control System]. SCCS saved deltas to a file and this was more efficient than creating copies. SSCS did not publish these changes. Then came the RCS [Revision Control System] in 1982, this was much better SCCS and it was free. It is still being managed by GNU Project.

After RCS was the CVS [Concurrent Versioning System], it was release in 1986. CVS were scripts that changed the RCS revisions of the group files. In CVS more than one user can edit the file at the same time. CVS allowed merges after the concurrent edits. There were edit conflicts. The developers were only allowed to commit new version of a file it related to the most recent version in the repository. If there are any edits done in the repository and the users working directory, they will have first resolve the conflict occurring cause of the edit done on the same lines.

CVS also had branches. This allowed developers to work parallelly and there are also tags that allowed naming for the snapshot that stay the same for referencing. CVS deltas were initially part of the repository on a “shared filesystem” (Dirkjan Ochtman, n.d.). CVS took this out and created a client-server architecture for larger networks, for example the internet.

A VCS was built in the 2000 by three developers. VCS was called Subversion. Subversion was created for solving larger parts of CVS. Subversion worked on trees, worked on edits in the revision should be.

* Consistent
* Atomic
* Isolated
* Durable

Subversion also continues to have the original version of the “checked out” (Dirkjan Ochtman, n.d.) revisions with the working directory. This way comparing the trees to the “checkout-out changeset” is local and it is fast.

Subversion includes the tags and branches are part of project tree. Subversion project includes

* Tags
* Branches
* Trunk

This is very instinctive for the user who do not have any experience with VCS. There are problems with the conversion tools because tags and branches are structural in other systems.

All the Control Systems that were talked about centralized. They can exchange changes. They can use other computers to keep track of the history of the repository. Distributed version control system will do something different; it will keep the copies of all/most of the repository history on computers that have a working directory of that repository.

**12.1.2. Distributed Version Control**

Subversion was better than the CVS however there were still improvements needed in the Subversion and this are

1. In all the CS [Centralized Systems] committing a changeset and publishing was quite similar because the repository history was centralized in one place. Network was a requirement for committing changes.
2. CS repository entry also requires one or more than one network round trips, and this made the system slow when it is compared distributed systems.
3. Systems mentioned before do not have the ability to track merges. When working in large groups, it becomes important that the version control system can record the changes in a new version. This way the nothing will get lost and the following merges can use this information.
4. The centralization that VCS needs are not realistic and there is single place for uniting/bonding. dVCS is more realistic with better organization and allows the developers to push around and merge changes.

To solve these 4 points there were tools created. These tools include Git, Mercurial, and Bazaar. Git and Mercurial were started in 2005 by Linux Kernel Developers. Tools can handle large volume of changesets in large volume of files. Matt and Linus both were influenced by the Monotone VCS. Bazaar was different created separately from Git and Mercurial. It also gained popularity at the same time and was adopted by Canonical with their project.

Distributed Version Control System also have problems, and this are.

1. There is no canonical view of the history, centralized system had this.
2. Changesets is committed parallelly, this does not allow the non permanent order revision with repository.

The solution is to use a DAG [Directed Acyclic Graph] of changeset rather than using the linear ordering. What this mean sis that the new commit changeset is the child revision of the base revision. None of the revision can be independent or and cannot depend on the descendant revisions. We have three type of revisions

* Root Revisions [No parents and repository can have more than one root]
* Merge Revisions [More than one parent]
* Head revisions [This revision has no children]

Repository will start with a root revision that is empty and this will go towards changesets. Then it will end in one or more of the heads. When any two users commit independently and one of the users decide to pull in the changes from the other user, they will have to explicitly merge the changes of the other into a new revision. This will be followed by a commit that will be a merge revision.

DAG will solve problems that are difficult to solve in a cVCS [Centralized Version Control System]. The merge revisions will record the data about the merged branches of the DAG. The graphs could also help wit parallel branches, and merging to a smaller group, and merging into a specific branch that is called a canonical.

The DAG needs the system to track old changesets and their relations. To support the exchange of the date of the changeset, the changesets can keep track of the parent changeset. Changeset do need an identifier to do this. Systems can use UUID and if not something like UUID. Git and Mercurial use SHA1 hashes. This also has extra properties that are useful and that is changeset ID can validate changeset contents. Since the parents are also part of the hashed data, the history related to the revisions can be validated using the hash. Timestamps, commit messages, Author names, and other changeset data that describes and give information about the other data is also hashed like the file contents of a new revisions and this can also be validated. Timestamps are reported at commit time, and they also do not carry forward linearly in a repository.

This could be difficult for people who are user of centralized VCS. There is no integer to name a revision worldwide, it is a string that contains a 40-character hexadecimal string. Global ordering does is not available. There is local ordering, and that is DAG. If by mistake a user starts a new head of development, and user commits contrary to parent revision that has a child can lead confusion when the user is comfortable with VCS warnings.

There are tools that could help with visualizing the tree ordering. Mercurial offers “changeset hash” that explicit and shot version to help with the identification. Mercurial also offers “local-only linear” to help with the identification. An integer that is monotonically climbing, expresses the order of changeset while entering the clone. A clone-to-clone order could different and this why it cannot be used for non-local operation.

A picture containing text, antenna

Description automatically generated

DAG

Reference

<https://www.youtube.com/watch?v=FyAAIHHClqI&ab_channel=DavidMahler>

<http://aosabook.org/en/git.html>

<http://aosabook.org/en/mercurial.html>

<https://www.atlassian.com/git/tutorials/saving-changes>

<https://git-scm.com/docs/git-stash>