

Distributed Version Control

Faisal Tameesh

April 3rd, 2015

Executive Summary

Version control is a cornerstone of modern software development. As opposed to the centralized, client-server architecture implemented by programs like Subversion (SVN), distributed version control is pioneered by Git. This application note will discuss the benefits of distributed version control, followed by specific examples of how to implement the concepts to maximize productivity in a professional software engineering environment.

Keywords

Git - Subversion - Distributed - Centralized - Version Control

Definition

Version control is a method used to track code. Any edits by developers are tracked. Version controlled projects can be “forked” to be used as a basis in new projects, which promotes an open community of development. Version control promotes experimentation on existing code without worry that the pre-existing working code will break.

Background

Software engineering relies upon the same development cycle that other engineering disciplines implement.

- Define the problem
- Do background research
- Specify requirements
- Brainstorm solutions
- Choose the best solution
- Do development work
- Build a prototype
- Test and redesign

Focusing specifically on the last four elements, version control plays a significant role in speeding up the process by maintaining an archive of all the added changes that were committed to the repository. Multiple solutions can be implemented as “branches” in the development and prototyping processes. Lastly, code testing and redesign, formally known as Test-Driven Development, is another cornerstone methodology of software engineering that version control makes more manageable.

History

One of the earliest revision control systems, Source Code Control System (SCCS), was geared towards program source code and other text files. It was originally developed in SNOBOL at Bell Labs in 1972 by Marc Rochkind for an IBM System/370 computer running OS/360 MVT. It was later rewritten by the same person in C for UNIX. Subsequently, SCCS was included in AT&T’s commercial System III and System V distributions. SCCS was the dominant version control system for UNIX until later version control systems, notably RCS, which later became CVS, gained more widespread adoption. Today, these version control systems are obsolete, particularly in the open source community, which has largely embraced distributed revision control systems.

Version Control Architectures

Git is the most popular of the distributed version control systems available today. It is important to distinguish between the concepts of “distributed version control” (like Git) and “centralized version control” (like SVN).

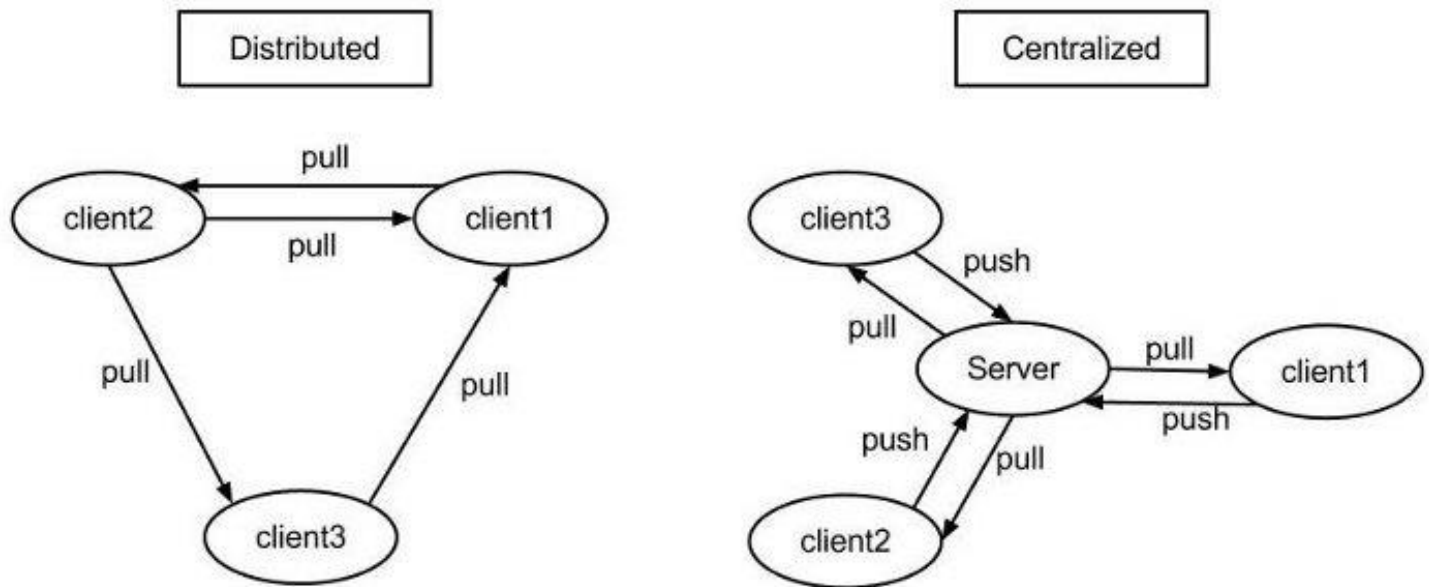


Figure 1. Distributed vs. Centralized version control systems

As can be seen in Figure 1, the distributed version control system does not depend upon having a central server where the repository information is contained. This allows for far greater flexibility in terms of internet connectivity and concurrency. In the centralized scenario, there must always be a connection to the server in order to commit code, whereas in the distributed system, committing the code happens on the local machine to the local version of the code. When sharing code using the distributed architecture, the system will merge the code changes based on the timestamps and changes that took place in the code.

In rare cases, the centralized system also demonstrates some concurrency issues. When there is a large number of commits to the centralized system, some commits may occur before others, which, if incorrectly managed, will lead to the loss, or even incorrect modification, of existing code. While there are systems that manage problems that may arise with such architecture, they still happen, since concurrency is a difficult problem to fully solve in such systems.

The distributed system has no such problems. This is because each person has a local version of the code. Whenever code is merged, it happens only between two versions of the code: the local version and the remote version. These changes can be merged via some available algorithms that the program does automatically; the user simply has to provide which algorithm to use for the merging process, and if he/she does not, a default one will be used.

Merging Strategies

In Git, these algorithms are called merging strategies. There are some basic elements to consider when discussing merging strategies.

- 1) The Source: This is the change set to merge from.
- 2) The Destination: The change set to merge to.
- 3) The Ancestor: The change set which is the nearest parent of The Source and The Destination.

Recursive: This is the default strategy. The algorithm looks at the ancestry of both versions until a common ancestor is found. It then uses this information as a basis to continue merging the two versions.

Octopus: This merge strategy is used when there are several trees that need to be merged. This is used in large projects where many branches have had independent development and all of the parts must now come together. This is the default when the merge command is invoked manually, rather when merge is automatically called upon “pulling” code from a remote repository. In the latter case, the recursive algorithm is still the default.

Ours: This strategy pulls in the Destination’s head (which is the most recent version of the change set), but throws away all the changes the head introduces. While this seems counterintuitive, this is used to maintain the history of a branch without any of the effects of a branch.

Subtree: This is useful when an API or library is to be used as part of an existing project. The subtree will have its own repository separate from the root’s repository. The subtree will be responsible for maintaining the API’s repository, while the root’s repository will be the application code that uses the API.

Application Note Overview

In this application note, setting up a Git repository and using the basics will be covered. Also, theoretical and practical reasoning will be provided to explain why things are done the way they are presented; often times, in software engineering, conventions are followed to keep the development process robust and more efficient.

Tutorial

Set Up

This tutorial will assume operating under a UNIX-based system. Using the appropriate package manager on the system (apt-get for Ubuntu Linux or Homebrew for Mac), install git by issuing a command that resembles the following:

```
apt-get install git
```

Initialization

This or the following step must take place in order to begin using version control. In order to begin using git, an empty or pre-existing directory must be initialized. For initialization of a pre-existing directory, enter the following command.

```
git init
```

To initialize a directory that has not been created yet, simply enter the name of the directory after the aforementioned command.

Cloning

The step is optional, because a user may opt to begin development without using another repository as a basis. In the event that an external repository is to be cloned, simply find the URL that specifies where the repository is to be found. In this example, the following repository may be used for learning purposes:

```
https://github.com/faisaltameesh/Drone.git
```

Once a URL has been specified, we can now clone it.

```
git clone https://github.com/faisaltameesh/Drone.git
```

This, like the previous step, can create a directory that has been initialized to use git, along with including any of the files that are in the most recent version of the repository that the URL points to. The user is now ready to begin editing files and then committing the changes to his/her local repository.

For the following step, it is assumed that the user has made some changes to the file in the repository, namely 'server.js'.

Adding the file(s)

After the edits, the user is now ready to add the file to the content stage for the next commit. Before a user can commit anything, he/she must add the files to the commit stage. To do so, the following command is implemented.

```
git add server.js
```

If interested, the user may add all the files that have been changed at once by issuing the following command.

```
git add -A
```

The '-A' flag indicates that entire working directory should be added to the staging area.

At this point the user is now ready to commit the changes to the repository. This means that as long as the repository is safely stored, any changes to the code, including the unchanged code that was first cloned, will be accessible by invoking the right commands.

Committing the file(s)

The user can now commit the file(s) to the repository with a brief message, stating the changes or fixes that took place.

```
git commit -m "Fixed timeout bug when the server connects over https."
```

This is an example of a commit message. Any message may be used and will be visible whenever the logs are accessed.

Viewing the log

```
git log
```

Reverting to a previous version

If the user would like to revert to a previous version without affecting the HEAD pointer points (this is used to manage the repository's history), the following command may be used.

```
git checkout <version_hash>
```

The checkout command can be used in conjunction with the hash specified at the beginning of each commit's log post when viewing the repository's history. At that point, the user may decide to utilize some of the older code, edit some files, etc... Afterwards, the same commit process can take place to make sure that the changes are stored.

Conclusion

Git is a very widely used program by software engineers around the world. While it may seem deceptively simple to use the program (and for basic purposes, it is!), the program is very powerful and is capable of maintaining very large code bases. Git's documentation is plentiful and well-organized. There are also many tutorials available online for those interested in taking their version control skills to the next level.