Politecnico di Torino

Integrazioni dei Sistemi Embedded

Course notes

Author: Maurizio Capra

CHAPTER 1

Versioning

In computer science, version control (versioning) is the management of multiple versions of a set of information. It is mainly used to develop engineering or computer science projects to manage the continuous evolution of digital documents such as software source code, technical drawings, text documentation, and other relevant information on which a team works. Version control systems are a software tool category considered necessary for software development teams to manage code changes over time. The version control software relies on special type databases (incremental-based) to track any code changes. If any team member makes a mistake, the developers can go back to the previous version and, by comparing it with the current one, find the problem minimizing the time and teamwork.
The source code represents a repository of valuable knowledge about a specific problem domain perfected with great effort for the team. The versioning protects the source code from catastrophic events and team members' errors. The source code is organized in a folder structure with a tree-like topology. Team members can work at the same time on different parts of it. In this scenario, version control is fundamental to track code changes and avoid any type of conflict. In fact, if more than one member modifies a file, a conflict occurs. The versioning system raises an error, but developers have to fix the issue. In this case there are two possibilities:

- variations are in the same file, but in different portions of the code, somehow decoupled from each other. In this case the developers can merge the changes.
- variations are in the same file and in same code portions. In this case it is necessary to understand if changes can coexist, or code must be rethought.

Note: every time a commit operation is performed, it is **mandatory** to insert a meaningful comment that motivates the changes and helps to trace them.

Most Version Control Systems (VCS) involve a certain vocabulary that must be known:

- **Repository (repo)**: The database storing the files;
- **Server**: The computer storing the repo;
- **Client**: The computer connecting to the repo;
- **Master**: The primary location for code in the repo. The master is the main line;
- **Branch**: Banch is a different line from the Master. The user can work in parallel on the Master and the Branch and then eventually merge them. When a Branch is created, it represents a copy of the Master;

- **Merge**: Apply the changes from one file to another, to bring it up-to-date. For example, users can merge features from one branch into another;
- **Add**: Put a file into the repository for the first time, i.e. begin tracking it with Version Control;
- **Head**: The latest revision in the repo;
- **Conflict**: When pending changes to a file contradict each other (both changes cannot be applied).

There exist mainly two types of VCS: centralized and distributed. In the former, the main repository is located on a remote server, and the user can download just a copy of the data locally, called working copy. As soon as one of the co-workers commits a change, the others can update their working copies and see the modifications. In the latter one, instead, users download the whole repository on a local workstation. This does not contain just the data, metadata, and history that allow trace file changes. When a user commits his changes, these are transferred only in the local repo, co-workers cannot see them. Only after he pushes the committed changes on the remote server co-workers to have access to them. In case of a catastrophic event on the server, the distributed VCS can recover the whole repository from a generic user. Two examples for such VCS types are Subversion for the centralized and GIT (or GitHub) for the distributed one.

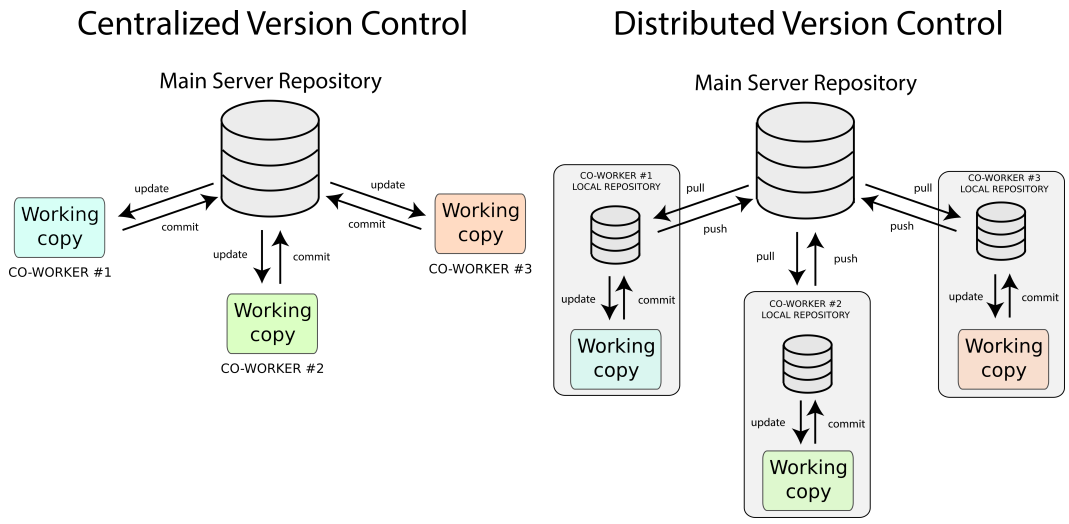

Figure 1.1: Centralized vs Distributed.

1.0.1 Versioning Benefits

It is possible to sum up the main benefits of a version control system in 3 points:

- A **long-term variation history** of each file. Such history includes author, date, and purpose of every created, deleted and altered file present in the repository. Having a so detailed chronology enables the user to turn back the clock anytime a bug occurs, identifying the root of the issue with a clear procedure, even in case of very complex projects. To be noted that different versioning software manage file differently.
- Team members can **work concurrently** on different code streams. In fact two or more branches can be developed in parallel and successively merged. This allows to have several workflows that can eventually converge. Whenever a merge is done, the software is in charge of detecting conflicts by relieving the user of this task.

- **Code traceability**. Being able to track any code changes along with their annotations and comments provides users with a powerful mean able to locate bugs. In long term program, such annotated history helps developers, even the new ones involved in the project, to easily understand the code and to take the most suitable move according to the intended design plan.

1.0.2 GIT

GIT is a distributed control version used by many big companies. In GIT many operations are local, in the sense that they need just local files to be performed. In fact since it is distributed, each user has its repository and the related history. Such an important feature allows users to work even when they are offline or they have no access to the server. GIT is based on four main stages in which files can reside: **untracked**, **modified**, **staged** and **committed**.

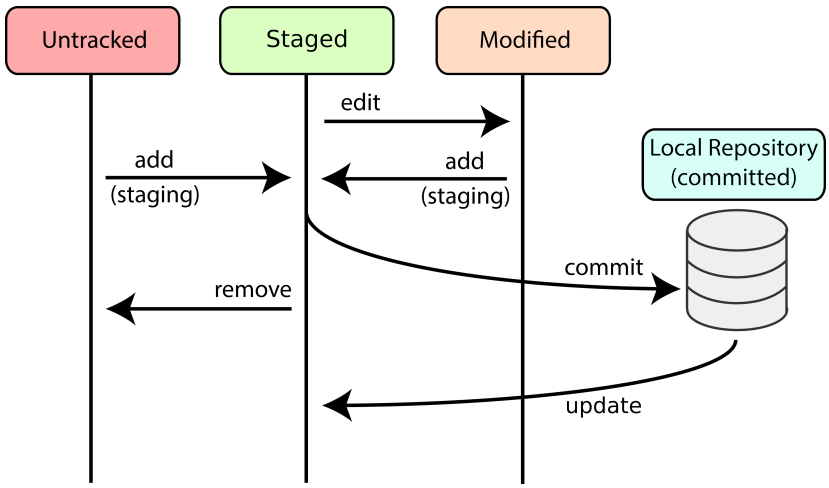

Figure 1.2: GIT stages.

Files labeled as untracked are part of the working directory, but the versioning system does not care about them, so their history is not tracked. When they are added to the repo, they are labeled as staged and the VCS starts to track their movements. Whenever a file is modified, it is labeled as modified and in order to be committed on the local repo, must be staged before. Files that have been committed can be successively pushed on the remote repository.
The working directory is a checkout of one version of the project. Such directory contains pulled files from the GIT compressed directory. Here the user modifies, deletes or creates files. The staging area is represented by a file, contained in the GIT directory that stores info about those files that will go into the next commit. GIT directory stores GIT metadata and represents the project database. Whenever a user calls for a clone, here is where files are moved.
In the following some basic commands are listed in order to get going with GIT.

Local repository

- **git init** [**project-name**] creates a new local repository in a sandbox folder;
- **git add** [**file**] adds (stages) [file] in preparation for versioning;
- **git add -u** adds all files already present in the repository in preparation for versioning;
- **git reset HEAD** [**file**] to unstage a file;
- **git checkout** –[**file**] discard changes in [file] and restore it as the last committed version;

- **git commit** records all staged files permanently in version history;
- **git commit -m "[message]"**;
- **git commit –amend** correct the last commit;
- **git status** describes the current situation of the repository;
- **git log** shows history of the current repository;
- **git diff** shows the differences between the files in the repository and last commit;
- **git diff [commit1] [commit2]** shows differences between [commit1] and [commit2];
- **git rm –cached [file]** removes file from version control (does not delete the file);
- **git rm [file]** deletes the file (almost definitively);
- **git reset [commit]** undoes all commits after [commit], local files are left unchanged;
- **git reset –hard [commit]** restores everything to the specified [commit].

Remote repository:

- **git clone [remote]** gets all data from the remote repository and assigns to it the shortname origin;
- **git init –bare** in a remote folder, creates a remote repository;
- **git remote add [shortname] [user@address:path to folder]** from the sandbox directory, links to the remote repository [user@address:path to folder] and assigns optional [shortname];
- **git pull [remote] [branch]** downloads most recent missing data from the remote to the local repository and automatically merges changes. [remote] can be a whole address or a shortname;
- **git fetch [remote]** downloads most recent missing data from the remote to the local repository. [remote] can be a whole address or a shortname;
- **git merge** manually merges files;
- **git push [remote] [branch]** example: git push [origin] [master] puts your changes in the remote repository. You must be up to date.

Branches:

- **git branch** lists all existing branches;
- **git branch [branch name]** begins new branch;
- **git checkout [branch name]** moves HEAD to point to the [branch name] branch;
- **git checkout -b [branch name]** creates branch [branch name] and moves HEAD to it;
- **git merge [branch name]** if you are on another branch (e.g. master) it merges [branch name] and the master branches;
- **git branch -d [branch name]** deletes branch [branch name];
- **git fetch [remote]** fetches everything from [remote] and sets remote-tracking branch [remote]/[RemoteBranch];

- **git checkout -b [NewBranch] [remote]/[RemoteBranch]** creates [NewBranch] from [RemoteBranch] and moves head to it.

For further details please visit the website `https://rogerdudler.github.io/git-guide/index.html`