sdfsdfdsfdsf

VCS or the Version Control System (Revision Control System)

- is a Repository of files with monitored Access

- it tracks, all the changes made in your repository.

- Who made the changes (author)

- Why they made it (message)

- References to problem fixed

- Enhancements introduced.

**Difference between Centralized and Distributed.**

**Centralized**

- Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy.

**Distributed**

- These systems do not necessarily rely on a central server to store all the versions of a project’s files. Instead, every developer “clones” a copy of a repository and has the full history of the project on their own hard drive. This copy (or “clone”) has all of the metadata of the original.

-This method may sound wasteful, but in practice, it’s not a problem. Most programming projects consist mostly of plain text files (and maybe a few images), and disk space is so cheap that storing many copies of a file doesn’t create a noticeable dent in a hard drive’s free space. Modern systems also compress the files to use even less space.

dfgfdg

**Git** is a Distributed Version control type that records changes to a file or set of files over time so that you can recall specific versions later.

**Git** - One of the most common version control tool used by a lot of developers.

**Git Diagram,**

- Here is Distributed Version Control Diagram.

- Every developer is pulling and pushing in it.

- As you can see in the diagram, there is a one server with 3 PC/workstation connected, and each of them has their own repository. So for example, for PC #1, I have used the command "pull" to make a working copy of files from the server. So these files can be anything like source codes, resource file, etc. And if you want to make your changes/updates reflect from the server, your will use the commit and push command.

**Git Overview**

- Snapshot

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes.

- Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like

at that moment and stores a reference to that snapshot.

- To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

- Nearly Every Operation is Local

- Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network. If you’re used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

- Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it.

- Generally Only Adds New Data

- When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.

- The Three States

- Committed means that the data is safely stored in your local database.

- Modified means that you have changed the file but have not committed it to your database yet.

- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

**Benefits of Git**

* + Speed
    - Example: With the use of Branching, all developers can make a copy of working repository on their workstation. Unlike SVN, we only have 1 centralize repository.
    - No Locking
  + Simple design
    - Simple design and very easy to use.
    - Design like the Git Bash Terminal
  + Strong support for non-linear development (thousands of parallel branches)
  + Fully distributed
    - Git is an open source program
  + Able to handle large projects like the Linux kernel efficiently (speed and data size)
  + Controlled Branching
    - Branches are controlled by the individual developers.
  + Better Synchronization
    - Better synchronization

**Git Limitations**

* + Usage Complexity
    - Git has a complex information model and it doesn’t really abstract that from the user. Git’s model is comprised of directed acyclic graphs, commits, trees, blobs, branches, tags, and remotes. Git has a staging area, a stash, and a reflog. All in all, Git has 145 commands, but there’s no git undo.
  + Repository Permissions
    - **Git** doesn’t concern itself with access control. It defers that job to the file system or ssh. Because of this, the ability to restrict access to Git repositories is severely limited.
    - **Read Access** - You cannot restrict read access to specific files, directories, or branches within a single repository with Git. You can either clone the entire repository or none of it. You need to arrange things such that setting read permissions per-repository is sufficient.
    - **Write** **Access** - Options for restricting write access aren’t quite as limited. While Git doesn’t support write access control out of the box, it does provide hooks that can reject pushes. So, third party tools can add write access control to Git. gitolite, for example, can restrict write access to files, directories, or branches. It can also restrict force pushes.
  + Unmodifiable History
  + Locks
    - Git does not support locking files.
  + Updating Very Large Repositories
    - Git stores snapshots of the entire history of a repository locally when you clone. Disk space is cheap and still getting cheaper. More likely, the bottleneck is network speed. If you need to clone a large repository over the network, it’s going to take a while. Megabytes, no problem. Gigabytes, manageable. Terabytes, don’t bother (in 2013). This is made more troublesome by the lack of partial clones.
  + No Revision Numbers
    - The key thing to understand is that git cannot have revision numbers - think about the decentralized nature. If users A and B are both committing to their local repositories, how can git reasonably assign a sequential revision number? A has no knowledge of B before they push/pull each other's changes. The key thing to understand is that git cannot have revision numbers - think about the decentralized nature. If users A and B are both committing to their local repositories, how can git reasonably assign a sequential revision number? A has no knowledge of B before they push/pull each other's changes.
    - A single, monotonically increasing revision number only really makes sense for a centralized version control system, where all revisions flow to a single place that can track and assign numbers. Once you get into the DVCS world, where numerous copies of the repository exist and changes are being pulled from and pushed to them in arbitrary workflows, the concept just doesn't apply. (For example, there's no one place to assign revision numbers - if I fork your repository and you decide a year later to pull my changes, how could a system ensure that our revision numbers don't conflict?)

SSH (Secure Shell) Access Key – provides more secure way of logging into virtual private server with SSH than using a password.

**When do you undo?**

1. committed too early
2. forgot to add some files,
3. messed up your commit message

git remote add name url

ex. Git remote add backup /sample/backup.git

git remote –v

name url(fetch)

name url(push)

origin <https://github.com/sample/sample> (fetch)

origin <https://github.com/sample/sample> (push)

git push –all push all branch

**Tracked** files are the one handled (version controlled) by Git, that were once added and committed. Untracked files are most of the time files you don't want to be controlled, because for example they are generated by your compiler.

You add **untracked** files to the .gitignore file, so that Git don't ask you if you want to track them.