Index

* Getting Started - About Version Control
* Local Version Control Systems
* Centralized Version Control Systems
* Distributed Version Control Systems
* Centralized vs Distributed Version Control: Which One Should We Choose?

# GIT

# What is a GIT ?

# **Characteristics of GIT**

# How GIT Works

# What is a GIT Repository?

# Synchronizing with Remote Repositories

1.1 Getting Started - About Version Control

This chapter will be about getting started with Git.

We will be learning some background on version control tools, then move on to how to install Git running on your system and finally how to get it set up to start working with.

At the end of this chapter you should understand why Git is around, why you should use it and you should be all set up to do so.

* **Benefits Version Control**

1. Enhances the project development speed by providing efficient collaboration,
2. Leverages the productivity, expedite product delivery, and skills of the employees through better communication and assistance,
3. Reduce possibilities of errors and conflicts meanwhile project development through traceability to every small change,
4. Employees or contributor of the project can contribute from anywhere irrespective of the different geographical locations through this **VCS,**
5. For each different contributor of the project a different working copy is maintained and not merged to the main file unless the working copy is validated. A most popular example is **Git, Helix core, Microsoft TFS,**
6. Helps in recovery in case of any disaster or contingent situation,
7. Informs us about Who, What, When, Why changes have been made.

* **Purpose of Version Control:**
* Multiple people **can work simultaneously on a single project**. Everyone works on and edits their own copy of the files and it is up to them when they wish to share the changes made by them with the rest of the team.
* It also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
* It integrates the work that is done simultaneously by different members of the team. In some rare case, when conflicting edits are made by two people to the same line of a file, then human assistance is requested by the version control system in deciding what should be done.
* Version control provides access to **the historical versions of a project**. This **is insurance against computer crashes or data loss**. If any mistake is made, you can easily roll back to a previous version. It is also possible to undo specific edits that too without losing the work done in the meanwhile. It can be easily known when, why, and by whom any part of a file was edited.
* **Local Version Control Systems**
* It is one of the simplest forms and has a **database** that kept all the changes to files under revision control. RCS is one of the most common VCS tools. It keeps patch sets (differences between files) in a special format on disk. By adding up all the patches it can then re-create what any file looked like at any point in time.
* To deal with this issue, programmers long ago **developed local VCSs** that had a simple database that kept all the changes to files under **revision control.**
* One of the most popular VCS tools was a **system called RCS**, which is still distributed with many computers today. [RCS](https://www.gnu.org/software/rcs/) works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.
* **Centralized Version Control Systems**
* Centralized version control systems contain just **one repository** and each user gets their **own working copy**. You need to commit to reflecting your changes in the repository. It is possible for others to see your changes by updating.
* The **benefit** of CVCS (Centralized Version Control Systems) makes collaboration amongst developers along with providing an insight to a certain extent on what everyone else is doing on the project. It allows administrators to fine-grained control over who can do what.
* It has some **downsides** as well which led to the development of DVS. The most obvious is the single point of failure that the centralized repository represents if it goes down during that period collaboration and saving versioned changes is not possible. What if the hard disk of the central database becomes corrupted, and proper backups haven’t been kept? You lose absolutely everything.

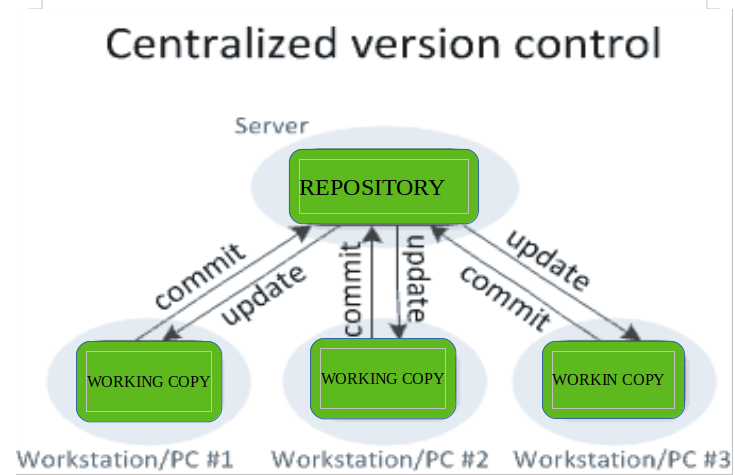


Figure. Centralized version control

* **Distributed Version Control Systems**
* Distributed version control systems contain. Each user has their own repository and working copy. Just committing your changes will not give others access to your changes. This is because commit will reflect those changes in your local repository and you need to push them in order to make them visible on the central repository. Similarly, when you update, you do not get other’s changes unless you have first pulled those changes into your repository.
* To make your changes visible to others, 4 things are required:
* You commit
* You push
* They pull
* They update
* The most popular distributed version control systems are **Git,** **Mercurial**. They help us overcome the problem of single point of failure.

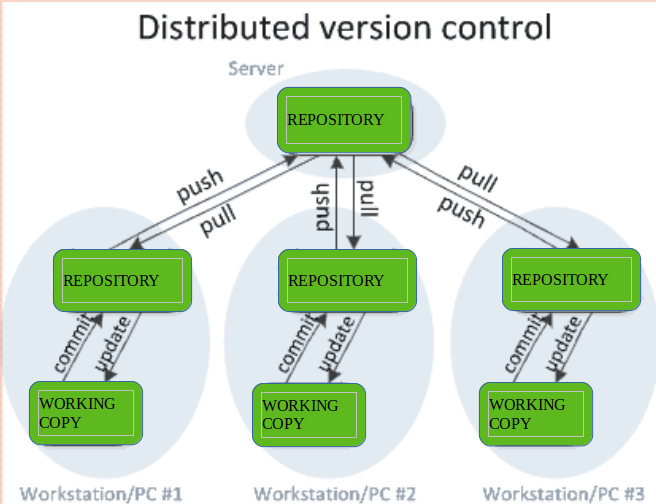


Figure. Distributed version control

# Centralized vs Distributed Version Control: Which One Should We Choose?

# In centralized source control, there is a server and a client. The server is the master repository that contains all of the versions of the code. To work on any project, firstly user or client needs to get the code from the master repository or server. So the client communicates with the server and pulls all the code or current version of the code from the server to their local machine.

# In other terms we can say, you need to take an update from the master repository and then you get the local copy of the code in your system. So once you get the latest version of the code, you start making your own changes in the code and after that, you simply need to commit those changes straight forward into the master repository. Committing a change simply means merging your own code into the master repository or making a new version of the source code. So everything is centralized in this model.

* There will be just one repository and that will contain **all the history or version of the code** and different branches of the code. So the basic workflow involves in the centralized source control is getting the latest version of the code from a central repository that will contain other people’s code as well, making your own changes in the code, and then committing or merging those changes into the central repository.
* In distributed version control most of the mechanism or model applies the same as centralized. The **only major difference you will find here is, instead of one single repository which is the server, here every single developer or client has their own server and they will have a copy of the entire history or version of the code and all of its branches in their local server or machine.**

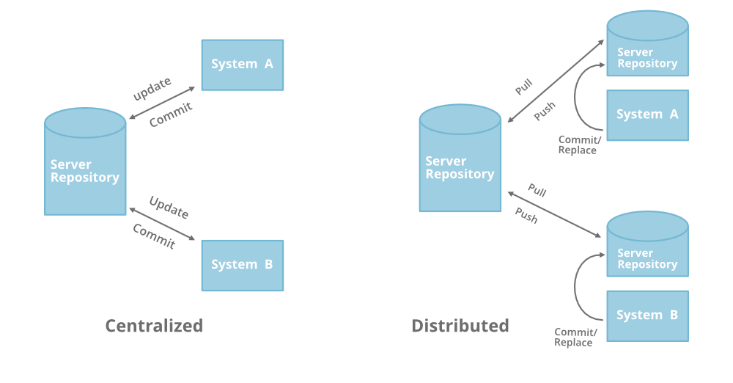


Figure. CVS vs DVS

# What is a GIT ?

* Git is a [free and open source](http://www.git-scm.com/about/free-and-open-source) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
* Git relies on the basis of distributed development of a software where more than one developer may have access to the source code of a specific application and can modify changes to it which may be seen by other developers...
* Initially designed and developed by [Linus Torvalds](https://en.wikipedia.org/wiki/Linus_Torvalds) for [Linux kernel](https://www.kernel.org/) development in 2005.
* Every git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.
* Git allows a team of people to work together, all using the same files. And it helps the team cope up with the confusion that tends to happen when multiple people are editing the same files.
* **Characteristics of GIT :**

1. Strong support for non-linear development
2. Distributed development
3. Compatibility with existing systems/protocol
4. Data Assurance
5. Automatic Garbage Collection
6. Periodic explicit object packing

* **How GIT Works**

1. A Git repository is a **key-value** object store where all objects are indexed by their SHA-1 hash value.
2. All commits, files, tags and file system tree nodes are different types of objects living in this repository.
3. A Git **repository is a large**[hash **table**](https://en.wikipedia.org/wiki/Hash_table) with **no provision**made for **hash collisions**.
4. Git specifically works by taking “**snapshots”** of files

# What is a GIT Repository?

# Repositories in [GIT](https://www.geeksforgeeks.org/git-lets-get-into-it/) contain a collection of files of various different versions of a Project. These files are imported from the repository into the local server of the user for further updating and modifications in the content of the file.

# A VCS or the [Version Control System](https://www.geeksforgeeks.org/version-control-systems/) is used to create these versions and store them in a specific place termed as a repository.

# The process of copying the content from an existing Git Repository with the help of various Git Tools is termed as ****cloning****. Once the cloning process is done, the user gets the complete repository on his local machine. Git by default assumes the work to be done on the repository is as a user, once the cloning is done.

# Users can also [create a new repository](https://www.geeksforgeeks.org/creating-repository-in-github/) or delete an existing repository. To delete a repository, the simpler way is to just delete the folder containing the repository.

## Working with a Repository

* A [GIT](https://www.geeksforgeeks.org/git-lets-get-into-it/) repository allows performing various operations on it to create different versions of a project. These operations include the addition of files, **creating new repositories, committing an action, deleting a repository,** etc. These modifications will result in **the creation of different versions** of a project.

### Synchronizing with Remote Repositories

* Git allows the users to perform operations on the Repositories by cloning them on the local machine. This will result in the creation of various different copies of the project.
* These copies are stored on the local machine and hence, the users will not be able to sync their changes with other developers. To overcome this problem, Git allows performing syncing of these local repositories with the remote repositories.

This synchronization can be done by the use of two commands in the Git.

* push
* pull
* **Push:** This command is **used to push all the commits of the current repository** to the tracked remote repository. This command can be used to push your repository to multiple repositories at once.
* **Syntax:**

$ git push -u origin master

To push all the contents of our local repository that belong to the master branch to the server(Global repository).

* **Pull:**

 Pull command is **used to fetch the commits** from a remote repository and stores them in the remote branches. There might be a case when other users perform changes on their copy of repositories and upload them with other remote repositories. But in that case, your copy of the repository will become out of date. Hence, to re-synchronize your copy of the repository with the remote repository, the user has to just use the

**git pull** command to fetch the content of the remote repository.

* **Syntax:**

$ git pull

**-Courtesy:**

**@GeeksForGeeks**