

MODULE 2

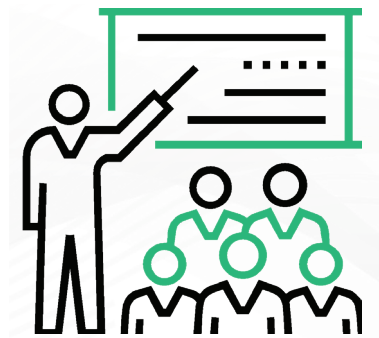
Source Code Management - History and Overview

You will learn about the 'Source Code Management - History and Overview' in this module.

Module Objectives

At the end of this module, you will be able to:

- Discuss the history of Version Control Systems (VCS)
- Explain basic operations that are done in a VCS
- Provide some examples for the different types of VCS
- Describe basics of Concurrent Versions System (CVS)
- Define Subversion, its features and limitations
- Explain Mercurial and its features
- Define Git
 - Overview and history
 - Advantages of Git



Module Topics

Let us take a quick look at the topics that we will cover in this module:

1. History of Version Control Systems (VCS)
2. Basic operations in a VCS
3. Examples of version control systems
4. Subversion (SVN)
 - Features and Limitations

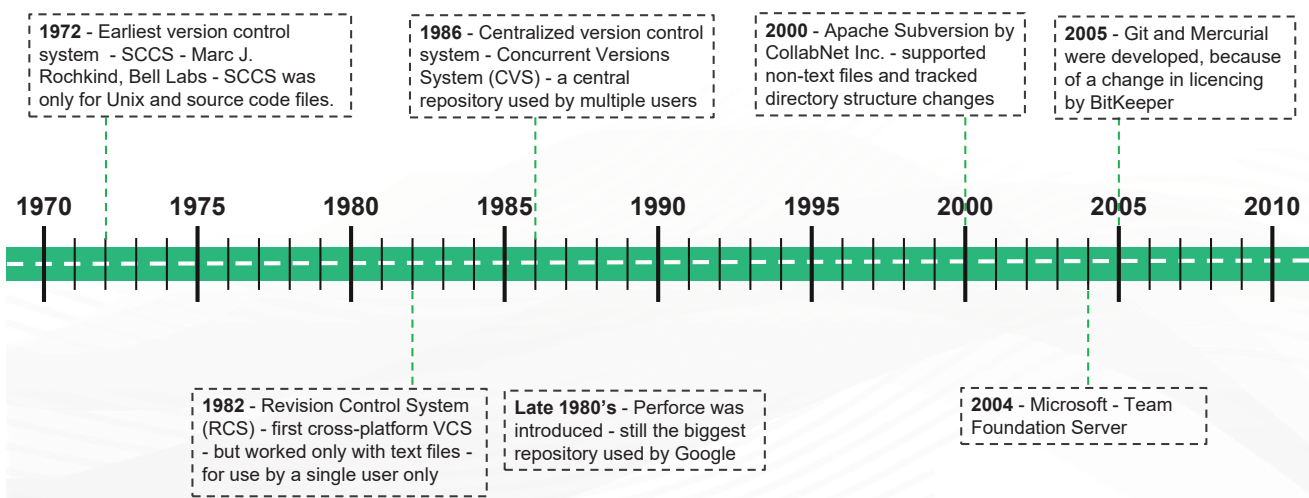


5. Mercurial

6. Git

- Overview
- History - Linux and Git by Linus Torvalds
- Advantages of Git

1.1 Brief History of Version Control Systems



The earliest of all version control systems was Source Code Control System (SCCS), written in 1972 by Marc J. Rochkind at Bell Labs. From it evolved most of the venerable open source version control systems, all still in use: RCS, CVS, and Subversion. SCCS was developed for the Unix operating system and it worked only with source code files. After 10 years, in 1982, Revision Control System (RCS), the first cross-platform VCS, was developed. But again, RCS worked only for text files. Both SCCS and RCS had limitations that they worked only for a single user and did not help in collaborative development.

Centralized version control systems, began to be developed in the year 1986. The initial release of CVS, Concurrent Versions System in 1990, was the first version control system with a centralized source code repository, which can be used by multiple users. One limitation that CVS had was that it still tracked the changes in the individual files, but not the entire directory.

Perforce came into existence in the late 1980's, which was the most widely used VCS during the .com era and is still the biggest repository used inside Google.

In 2000, Subversion was launched by CollabNet, which supported non-text files and it had the ability to track directory structure changes. The whole directory can be checked out, updated and checked in.

Microsoft introduced the Team Foundation Server (TFS) in 2004 to replace the Visual SourceSafe Version Control system. TFS was expensive, as it came only with MSDN subscription. The TFS Express edition is closely integrated with Visual Studio, so code can be checked in and out from there. TFS also has bug and issue tracking features.

The next era of VCS is the Distributed Version Control System. It enabled everyone to have their individual repositories and the changes can then be shared with the central server. In 2005, BitKeeper, a version control system provider, changed its licensing methods. The BitKeeper community edition was used by Linus Torvalds and the kernel group on Linux. The licensing issue made them create a separate version control system for themselves, and that's how Git was created. Git revolutionized the way of source code management. In the same year, a competing product Mercurial was also developed due to BitKeeper changes, which has been widely used in open source projects.

Three Generations of Versions Control

Generation	Networking	Operations	Concurrency	Example
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

What did You Grasp?



- Which of the following is the first cross-platform VCS?
 - RCS
 - SCCS
 - CVS
 - Git

2.1 Basic Operations in a VCS

There are a set of basic operations performed in a VCS. They are as follows:

→ Create	→ Status
→ Checkout	→ Diff
→ Commit	→ Revert
→ Update	→ Log
→ Add	→ Tag
→ Edit	→ Branch
→ Delete	→ Merge
→ Rename	→ Resolve
→ Move	→ Lock

The list of basic operations typically performed in a VCS is given above. Let's look at each of them in detail.

2.1.1 Create

Create operation here refers to creating a new repository. A repository is a database where all the edits or updates of the source code are stored.

Following are the key details of create:

- Repository keeps track of the tree, i.e., all the files and the layout of the directories in which they are stored.
- A repository is a three-dimensional entity that exists in a continuum defined by directories, file and time.
- While creating a new repository, we need to specify the name and the location in which it has to be created.

The Create operation is used to create a new empty repository. A repository, as we saw in the previous module, is a database that contains all the versions of and changes made to the source code. As opposed to a file system, which is two-dimensional, a repository is three dimensional that exists in a continuum defined by directories, files and time. The create operation is one of the first operations you will use, and after that, it gets used a lot less often. While creating a repository, the VCS would expect us to specify the name and the location in which it has to be created.

2.1.2 Checkout

Following are the key details of checkout:

- The checkout operation is used to make a new working copy of an existing repository.
- A working copy is a snapshot of the repository used by a developer as a place to make changes. Though the repository is shared by all the developers, the changes are made only to the working directory first, and then merged with the repository.
- A working copy is a private space to a developer and changes made to this working copy will not affect the rest of the team.
- The working copy is actually more than just a snapshot of the contents of the repository, but also contains some metadata so that it can keep careful track of the state of things.

The checkout operation is used to make a new working copy for a repository that already exists. A working copy is a snapshot of the repository used by a developer as a place to make changes. The repository is shared by the whole team, but people do not modify it directly. Rather, each individual developer works by updating the working copy.

With a version control tool, working on a multi-person team is much simpler. Each developer has a working copy to use as a private workspace. He can make changes to his own working copy without adversely affecting the rest of the team. The working copy is actually more than just a snapshot of the contents of the repository. It also contains some metadata so that it can keep careful track of the state of things.

2.1.3 Commit and Update

Following are the key details of commit and update operations:

- Commit operation is used to apply the modifications in the working copy to the repository as a new changeset.
- The commit operation takes the pending changeset and uses it to create a new version of the tree in the repository.
- The Update operation is used to update the working copy with respect to the repository.
- The Update is sort of like the mirror image of commit. Both operations move changes between the working copy and the repository.
- Commit goes from the working copy to the repository. Update goes in the other direction.

Commit: Commit is the operation that actually modifies the repository. Several other operations modify the working copy and add an operation to a list that is called the pending *changeset*, a place where changes wait to be committed. The commit operation takes the pending changeset and uses it to create a new version of the tree in the repository. All modern version control tools perform this operation atomically. In other words, no matter how many individual modifications are in your pending changeset, the repository will either end up with all of them (if the operation is successful), or none of them (if the operation fails). It is impossible for the repository to end up in a state with only half of the operation done. The integrity of the repository is assured. It is typical to provide a log message (or comment) when you commit, explaining the changes you have made. This log message becomes part of the history of the repository.

Update: Update makes the working copy up-to-date by applying changes from the repository, merging them with any changes that have been made to the working copy if necessary. When the working copy was first created, its contents exactly reflected a specific revision of the repository. The VCS remembers that revision, so that it can keep careful track of where you started making your changes. This revision is often referred to as the parent of the working copy, because if changes are committed from the working copy, that revision will be the parent of the new changeset.

2.1.4 Add, Edit and Delete

Following are the key details of add, edit and delete operations:

- Add operation is used to add a file or directory to the working copy that is not added to version control yet, which has to be added to the repository.
- Edit operation is used to modify a file.
- Edit doesn't directly involve the VCS, edits are actually made using the text editor or the development environment and the VCS will notice the change and make the modified file part of the pending changeset.
- Delete operation is used to delete a file or a directory.
- The delete operation will immediately delete the working copy of the file, but the actual deletion of the file in the repository is added to the pending changeset.

Add: The add operation is used when there is a file or directory in the working copy, that is not yet under version control and that has to be added to the repository. The item is not actually added immediately. Rather, the item becomes a part of the pending changeset, and is added to the repository when you commit.

Edit: Edit operation is used to modify a file and is the most common operation when using a version control system. During checkout, the working copy contains a bunch of files from the repository. The files are modified with an intention to make the changes a part of the repository. With most version control tools, the edit operation doesn't actually involve the VCS directly. The file is edited using a text editor or development environment and the VCS will notice the change and make the modified file part of the pending changeset.

Delete: The delete operation is used when a file or directory has to be removed from the repository. If you try to delete a file which has been modified in your working copy, your VCS might raise a warning. Typically, the delete operation will immediately delete the working copy of the file, but the actual deletion of the file in the repository is added to the pending changeset. In the repository the file is not really deleted. When a changeset containing a delete is committed, a new version of the tree is created, which does not contain the deleted file. The previous version of the tree is still in the repository, and that version still contains the file.

2.1.5 Rename and Move

Following are the key details of remove and move operations:

- Rename operation is used to change the name of a file or a directory.
- The operation is added to the pending changeset, but the item in the working copy typically gets renamed immediately.
- Rename detection usually works well in practice, but if a file has been both renamed and modified, there is a chance the VCS will do the wrong thing
- The Move operation is used to move a file or a directory from one location to the other.
- Some tools treat rename and move as the same operation, while others keep them separate.

Rename: To change the name of a file or a directory, the rename operation is used. The item in the working copy gets renamed immediately, even when the operation is added to the pending changeset. There is a lot of variety in how version control tools support rename. Some of the previous tools had no support for rename at all. Some tools (including Bazaar and Veracity) implement rename formally, requiring that they be notified explicitly when something has to be renamed. Such tools treat the name of a file or directory as simply one of its attributes, subject to change over time.

Tools like Git implement rename informally, detecting renames by observing changes rather than by keeping track of the identity of a file. Rename detection usually works well in practice, but if a file has been both renamed and modified, there is a chance the VCS will do the wrong thing.

Move: The move operation is used to move a file or directory from one place in the tree to another. The operation is added to the pending changeset, but the item in the working copy typically gets moved immediately. Some tools treat rename and move as the same operation (in the Unix tradition of treating the file's entire path as its name), while others keep them separate (by thinking of the file's name and its containing directory as separate attributes).

2.1.6 Status, Diff and Revert

Following are the key details of status, diff and revert operations:

- Status operation is used to list the modifications that have been made to the working copy.
- In other words, status shows what changes would be applied to the repository if it is committed.
- Diff shows the details of the modifications that have been made to the working copy (status shows the list alone, but not the details).
- A revert is used to undo the modifications that have been made to the working copy.
- A complete revert of the working copy will throw all your pending changes and return the working copy to the way it was just after the checkout.

Status: As changes are made in the working copy, each change is added to the pending changeset. The status operation is used to see the pending changeset. In other words, status shows what changes would be applied to the repository if it is committed.

Diff: The status provides a list of changes, but no details about them. To see exactly what changes have been made to the files, the diff operation is used. VCS implements diff in a number of different ways. For a command-line application, it may simply print out a diff to the console or a visual diff application may be launched.

Revert: A revert is used to Undo modifications that have been made to the working copy. A complete revert of the working copy will throw away all the pending changes and return the working copy to the way it was just after the checkout.

2.1.7 Log and Tag

Following are the key details of log and tag operations:

- The log operation shows the history of changes to the repository.
- Most version control tools present ways of slicing and dicing this information.
- Log operation displays information such as who made the change and when and the log message.
- Version control tools provide a way to mark a specific instant in the history of the repository with a meaningful name, called the tag.
- This is not altogether different from the descriptive and memorable names we use for variables and constants in our code.

Log: Log operation shows the history of changes to the repository. Repository keeps track of every version that has ever existed. The log operation is the way to see those records. It displays each changeset along with additional data such as:

- Who made the change?
- When was the change made?
- What was the log message?

Most version control tools present ways of slicing and dicing this information. For example, you can ask log to list all the changesets made by a particular user, or all the changesets made during April 2018.

Tag: Tag is used to associate a meaningful name with a specific version in the repository. Version control tools provide a way to mark a specific instant in the history of the repository with a meaningful name. This is not altogether different from the descriptive and memorable names we use for variables and constants in our code.

2.1.8 Branch and Merge

Following are the key details of branch and merge operations:

- Branch operation is used to create another line of development.
- Branch operation is especially useful when a new version of the software is developed and still the older version has to be kept separate.
- Merge operation is used to apply changes from one branch to another.
- Typically, merge operation is helpful when a branch is used to enable the development to diverge, which later has to converge again, at least partially.
- Changes can be made in two branches manually, but merge operation makes the process simple by automating this as much as possible.

Branch: Branch is used to create another line of development. The branch operation is used when the development process has to diverge into two different paths. This is essentially useful during different versions of software release. Branching is to ensure that the development of one version doesn't impact the previous stable version. Some of the dimensions of branching are as follows:

- **Physical:** In this form, physical configuration of the system is branched. Components that are branched are files, components, and subsystems.
- **Functional:** In this form, functional configuration of the system is branched. Components that are branched are features, logical changes, both bug fixes and enhancements, and other significant units of deliverable functionality (e.g., patches, releases, and products).
- **Environmental:** In this form, operating environment of the system is branched. Here, various aspects of the build and runtime platforms like compilers, windowing systems, libraries, hardware, operating systems, etc. and/or for the entire platform is branched.

- **Organizational:** The team's work effort is branched. Here, branches are created for activities/tasks, subprojects, roles, and groups.
- **Procedural:** The team's work behaviors are branched. In this form, branches are created to support various policies, processes, and states.

Merge: Merge is used to apply changes from one branch to another. Typically, when you have used branch to enable your development to diverge, you later want it to converge again, at least partially. For example, if you created a branch for 3.0.x bug-fixes, you probably want those bug fixes to happen in the main line of development as well. Without the merge operation, you could still achieve this by manually doing the bug-fixes in both branches. Merge makes this operation simpler by automating things as much as possible.

2.1.9 Resolve and Lock

Following are the key details of resolve and lock operations:


- Resolve operation is used to handle conflicts that arise from a merge operation.
- Merge automatically deals with everything that can be done safely. Everything else is considered a conflict.
- The resolve operation is used to help the user figure things out and to inform the VCS how the conflict should be handled.
- The lock operation is used to get exclusive rights to modify a file.
- Not all version control tools include this feature. In some cases, it is provided, but is intended to be rarely used.

Resolve: In some cases, the merge operation requires human intervention. Merge automatically deals with everything that can be done safely. Everything else is considered a conflict. For example, what if the file `foo.js` was modified in one branch and deleted in the other? This kind of situation requires a person to make the decisions. The resolve operation is used to help the user figure things out and to inform the VCS how the conflict should be handled.

Lock: The lock operation is used to get exclusive rights to modify a file. Not all version control tools include this feature. In some cases, it is provided, but is intended to be rarely used. For any files that are in a format based on plain text (source code, XML, etc.), it is usually best to just let the VCS handle the concurrency issues. But for binary files which cannot be automatically merged, it can be handy to grab a lock on a file.

So far, we have seen the basic operations done using a VCS. Let's now look at some examples for VCS. Knowledge of basic operations is important to understand the features of different VCS.

What did You Grasp?



1. Which of the following operations is used to show the list of modifications made to the working copy?
A) Log
B) Status
C) Diff
D) Tag
2. Add operation is used to modify a file.
A) True
B) False

3.1 Examples of Version Control Systems

The following table summarizes the types of version control systems and the examples for each of the categories.

Local Data Model		Centralized (client-server) Model		Distributed Model	
Open Source	Proprietary	Open Source	Proprietary	Open Source	Proprietary
<ul style="list-style-type: none"> → Source Code Control System (SCCS) → Revision Control System (RCS) 	<ul style="list-style-type: none"> → None 	<ul style="list-style-type: none"> → Concurrent Versions System (CVS) → Subversion (SVN) → Vesta 	<ul style="list-style-type: none"> → Team Foundation Server (TFS) → Visual SourceSafe → Visual Studio Team Services (VSTS) → AccuRev 	<ul style="list-style-type: none"> → Git → Mercurial → Veracity → GNU arch → Bazaar → BitKeeper 	<ul style="list-style-type: none"> → Visual Studio Team Services → Plastic SCM

We learnt about the types of VCS in the previous module. Go through the table above and know the examples for VCS. The list is not comprehensive and there are a lot more systems available. The table presents the most popular systems.

We'll learn about a few version control systems in the upcoming sections. SCCS and RCS are so rarely used these days and so we'll start with centralized version control systems like SVN. For better understanding of SVN, we'll explain CVS first.

What did You Grasp?



1. Which among the following is an open source VCS?
 - A) TFS
 - B) Plastic SCM
 - C) Visual SourceSafe
 - D) SCCS

4.1 Concurrent Versions System (CVS)

Following are the key details of Concurrent Versions System (CVS):

- CVS is an open source layer implemented on top of RCS, since RCS was restricted to local file systems like SCCS.
- CVS works on a client-server model.
- CVS was meant to provide powerful branching and tagging functionalities.
- Originally written in 1984–1985 by Dick Grune and made publicly available in 1986 as a set of shell scripts, and later ported to C in 1988 by Brian Berliner.
- Primary innovation in CVS is that files are not locked to individual's use, hence concurrency can be maintained.
- Concurrency here refers that multiple users can work on the same repository at the same time.
- CVS has also inherited some of the problems from RCS, despite the innovations it provided to the developer's world.

CVS stands for Concurrent Versions System. 'Concurrent' in this context means that multiple developers can work at the same time on the same repository. CVS is an open source wrapper implemented on top of RCS, which provides extra features such as a client-server architecture and more powerful branching and tagging facilities. Originally written in 1984–1985 by Dick Grune and made publicly available in 1986 as a set of shell scripts, it was ported to C in 1988 by Brian Berliner. For many years, CVS was the best known and most popular version control system in the world, mainly because it was the only free VCS. CVS brought a number of innovations both to versioning and to the software development process. Probably the most important of these is that the default behavior of CVS is not to lock files (hence 'concurrent')—in fact, this was the principal motivation for CVS' development.

Despite its innovations, CVS has many problems, some of which are due to its inheriting a per-file change tracking system from RCS.

- Branching in CVS involves copying every file into a new copy of the repository. This can take a long time and use a lot of disk space if you have a big repository.
- Since branches are copies, merging from one branch into another can give you lots of phantom conflicts, and does not automatically merge newly added files from one branch into another. There are workarounds, but they are time-consuming, error-prone, and altogether thoroughly unpleasant.
- Tagging in CVS involves touching every file in the repository, another time-consuming process in large repositories.
- Check-ins to CVS are not atomic. This means that if your check-in process gets interrupted, your repository will be left in an intermediate state. Similarly, if two people try to check in at the same time, the changes from both sources may be interleaved. This makes it hard to see who changed what, or to roll back one set of changes.
- Renaming a file is not a first-class operation: You have to delete the old file and add a new one, losing the revision history in the process.
- Setting up and maintaining a repository is hard work.
- Binary files are just blobs in CVS. It makes no attempt to manage changes to binary files, so disk usage is inefficient.

4.2 Subversion (SVN)

Following are the key details of Subversion (SVN):

- Subversion (SVN) was developed to be a better alternative to CVS, that addresses many of the limitations posed by CVS.
- The command structure of SVN is very similar to CVS and this helped many CVS users quickly adapt to SVN.
- SVN changed the unit of versioning from files in SCCS and RCS to revision, i.e., you need not have a file for every file checked in, rather a set of changes to the files in a set of directories.
- A revision is the snapshot of all the files in the repository at that point in time.
- In addition to describing changes to files, deltas can include instructions for copying and deleting files.
- In SVN, every commit applies all changes atomically and creates a new revision.

Subversion was developed in 2000 by CollabNet Inc., as part of the Apache Software Foundation. It is an open source, centralized version control system.

Subversion (SVN) was designed to fix many of CVS' problems, and in general can be used as a superior replacement to CVS in any situation. It was designed to be familiar to users of CVS, and retains essentially the same command structure. This familiarity has helped SVN rapidly replace CVS in application software development.

Many of the good qualities of SVN derive from abandoning the format common to SCCS, RCS, and their derivatives. In SCCS and RCS, files are the unit of versioning: There is a file in the repository for every file checked in. In SVN, the unit of versioning is revision, which comprises a set of changes to the files in a set of directories. You can think of each revision as containing a snapshot of all the files in the repository at that time. In addition to describing changes to files, deltas can include instructions for copying and deleting files. In SVN, every commit applies all changes atomically and creates a new revision.

4.2.1 Features of SVN

Following are the features of SVN:

- In SVN, revision numbers apply globally to the repository, compared to individual files.
- Subversion treats directories, file attributes, and metadata the same way it treats files, i.e., changes to any of these can be versioned in the same way as changes to files.
- There are three subdirectories in each Subversion repository: trunk, tags and branches.
- SVN is an improvisation to CVS, that it keeps a local copy of the version of every file.
- More recent versions of Subversion have features such as merge tracking, that makes it comparable to commercial VCS.

One of the most important characteristics of SVN's repository model is that revision numbers apply globally to the repository rather than to individual

files. SVN treats directories, file attributes, and metadata the same way it treats files, which means that changes to these objects can be versioned in the same way as changes to files.

Branching and tagging in SVN are also much improved. Instead of updating each individual file, SVN leverages the speed and simplicity of its copy-on-write repository. There are three subdirectories in every SVN repository: trunk, tags, and branches. To create a branch, we can simply create a directory with the branch name under the branches directory, and copy the contents of trunk at the revision that has to be branched from, to the new branch directory that is just created. The branch that is just created is thus simply a pointer to the same set of objects that the trunk points to, until the branch and trunk begin to diverge. As a result, branching in SVN is an almost constant-time operation.

Tags are handled in exactly the same way as branches, except they are stored under a directory called tags. SVN does not distinguish between tags and branches, so the difference is simply a convention. A tagged revision can be treated as a branch in SVN.

SVN is also an improvisation to CVS by keeping a local copy of the version of every file as it existed during the last check out from the central repository. This means that many operations can be

performed locally, making them much faster than in CVS. They can even be done when the central repository is not available, which makes it possible to continue working while disconnected from the network.

4.2.2 Limitations of SVN

Limitations of SVN arise out of the client-server model in which it is developed. These limitations are as follows:

- Changes can only be committed while online.
- The data that SVN uses to track changes on local clients is stored in `.svn` directories in each folder in the repository. This might lead to confusions.
- While server operations are atomic, client-side operations are not. If a client-side update is interrupted, the working copy can end up in an inconsistent state.
- Revision numbers are unique in a given repository, but not globally unique across different repositories.
- SVN repositories cannot support some features of distributed version control systems.

The limitations of SVN are mainly due to the fact that it follows a client-server model. Some of the limitations of SVN are listed below:

- Changes can only be committed while online. This might sound obvious, but one of the main advantages of distributed version control systems lies in the fact that checking in is an operation separate from sending the changes to another repository.
- The data that SVN uses to track changes on local clients is stored in `.svn` directories in each folder in the repository. It is possible to update different directories on the local system to different revisions, and even to different tags or branches. While this may be desirable, in some cases it can lead to confusion and even errors.
- While server operations are atomic, client-side operations are not. If a client-side update is interrupted, the working copy can end up in an inconsistent state. Generally, this is fairly easy to fix, but in some cases it is necessary to delete whole subtrees and check them out again.
- Revision numbers are unique in a given repository, but not globally unique across different repositories. This means, for example, that if a repository is broken into smaller repositories for some reason, the revision numbers in the new repositories will not bear any relationship to the old ones.
- SVN repositories cannot support some features of distributed version control systems.

We'll now move on to Mercurial, one of the distributed version control systems.

What did You Grasp?



1. Revision numbers in SVN are applied globally.
A) True
B) False
2. Which of the following statements about SVN is/are true?
A) Tagged revisions can be considered as a branch in SVN
B) The unit of versioning in SVN is file
C) Trunk is one of the subdirectories in SVN
D) None of the above

5.1 Mercurial

Following are the key details of Mercurial:

- Mercurial is an open source VCS, originally designed for large-scale projects, written in Python and developed as an alternative to BitKeeper.
- Mercurial was created with Matt Mackall in 2005, who served as its lead developer until late 2016.
- Mercurial is decentralized and aims to be fast, lightweight, portable and easy-to-use.
- Mercurial is a truly distributed VCS, that offers each developer a local copy of the entire development history.
- Mercurial has been adopted by many organizations like Facebook, W3C and Mozilla.



Like Git, Mercurial was also developed as an alternative to BitKeeper, because of the licensing issues that arose in 2005. Mercurial was created by Matt Mackall in 2005, as a free, open source version control system. It is originally designed for use in large-scale projects.

Unlike traditional systems like SVN, which built as client-server architecture model, Mercurial is truly distributed VCS, where a local copy of the entire development history is given to each developer.

Mercurial is adopted by many organizations like Facebook, W3C, Mozilla, etc. Some of the open source projects that use Mercurial are Adblock Plus, Mozilla, RhodeCode, NetBeans, OpenJDK, etc.

5.1.1 Features of Mercurial

Following are the features of Mercurial:

- Implementation of Mercurial and its data structures are designed to perform fast.
- Mercurial is platform-independent with most parts of it written in Python, and a portion of it in portable C.
- The command line interface (CLI) of Mercurial is similar to SVN, this makes the transition between the tools smoother.
- History is always preserved in Mercurial, with hg rollback it is possible to undo the last pull or commit.
- There are many third-party GUIs available that make working with Mercurial a lot easier.
- The functionalities of Mercurial can be increased with extensions, some are and shipped with the official.

The implementation and data structures of Mercurial are designed to perform fast. Committing, branching and merging are fast and cost-effective. Being decentralized, it aims to be fast, lightweight, portable and easy-to-use, hence suitable for large-scale projects. It is platform-independent, with most parts of it written in Python, and a portion in portable C.

The command line interface of Mercurial is full-featured, stable and elegant. Mercurial documentation is very concise and using `hg help`, it is very easy to find what we are looking for. Being similar to SVN's CLI, Mercurial allows an easy transition between these tools. Mercurial anticipates some of the most common aliases for commands and makes them work.

Mercurial preserves the history and we can generate diffs between revisions, or jump back in time within seconds. Many programmers are inclined towards using a graphical interface to their version control system compared to a command line. Several third-party GUIs are available to make working with Mercurial easier. Mercurial is also highly extensible, i.e., there are some official ones shipped along with Mercurial and third-party extensions are also supported.

The command set is consistent, which is the desired feature for many developers. Mercurial is backwards-compatible, which is a very important feature to consider when using for a long-term.

What did You Grasp?



1. State True or False.
Mercurial is a centralized VCS.
A) True
B) False

6.1 Git

- Git is one of the free, open source distributed version control systems, created by Linus Torvalds in 2005, as an alternative to BitKeeper.
- Git was originally written in C, but has also been re-implemented in other languages like Java, Ruby, Python, etc.
- Git has been the VCS of choice in many open source projects like Android, Eclipse, etc., and also in most of the commercial organizations.

Some of the goals behind the development of Git were as follows:

Speed

Simple design

Strong support for non-linear development (thousands of parallel branches)

Fully distributed

Able to handle large projects like the Linux kernel efficiently (speed and data size)

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Git was born out of a licensing issue that the Linux kernel development team faced with BitKeeper, that abruptly invalidated the free licenses. The history of Git is explained in the next section. Git was originally written in C, but it has been implemented in many other languages, including Java, Ruby, Python, etc.

Some of the goals behind the development of Git were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Git has been used in many open source projects as well as commercial organizations across the world.

6.2 The Making - History of Linux and Git

- 1 No VCS was used by Linus Torvalds initially. Patches developed by Kernel contributors were manually applied by Linus.
- 2 Linus was not satisfied with any of the VCS available at that time, but kernel developers insisted to have one.
- 3 In 2002, Linus opted BitKeeper, a closed source commercial system developed by BitMover company.
- 4 BitMover imposed certain restrictions on the Linux community in exchange for the free license and demanded control over some metadata.
- 5 In 2005, due to a contract breach, BitMover revoked all the licenses issued to the Linux community and they were forced to have an alternative.
- 6 Linus began writing Git in 2005, and in June 2005, Linus' git revision control system had become fully self-hosting.

We have already seen that Git originated because of the licensing issue that the Linux community faced with BitKeeper. Let's have a detailed look at how the Linux community managed their source code earlier and how Linus Torvalds ended up creating Git on his own.

Initially there was no version control at the Linux community and Linus Torvalds, the creator of Linux, was manually applying the patches to the source tree, whenever the contributors submitted their patches. Though open source VCS like CVS were around that time, there were many limitations to it. Particularly, the way CVS tracked the changes and conflict management were troublesome. Linus was not a great fan of SVN as well.

Not having a proper revision control system made the kernel developer community unhappy and Linus was forced to choose a version control system. To everyone's shock, Linus, one of the greatest advocates and practitioners of open source, selected BitKeeper, a closed-source commercial VCS provided by BitMover company. This created a lot of controversy among the kernel development community, but that did not change Linus' standpoint. BitKeeper's major claim was that it offered a distributed system, that no provider offered at that time. Linus was particular about using BitKeeper mainly because of this reason.

The major drawback was that BitKeeper imposed certain restrictions on the kernel development community in exchange for the free license. One major condition was that Linux developers should not work on any competing revision control projects while using BitKeeper. Second was that BitMover, in order to monitor any abuse to the free license, would control certain metadata of the kernel project. Kernel developers could not compare previous kernel versions without the metadata. Despite these conditions, Linus continued to use BitKeeper for years.

The concept of distributed version control system may sound familiar now, but in 2002 it was a completely new idea. Linus was inclined towards BitKeeper, since sub-groups of kernel developers could collaborate independently with the benefit of revision control and then feed their changes up to Linus when they were ready. This way, a huge portion of the work done by Linus (applying the patches manually) could be distributed to any other group. Many features could be developed independently and merged to the Linux source tree. None of the open source VCS available at that time provided these advantages.

In 2005, Andrew Tridgell, the creator of Samba, tried to reverse engineer BitKeeper to create an open source alternative. This was against the contract and BitKeeper revoked the licenses issued to the Linux community and there arose a sudden sense of uncertainty.

To get over this, Linus stopped working on the Linux kernel, the first such incident since its inception in 1991, and started writing his own version control system. And that's how Git was born.

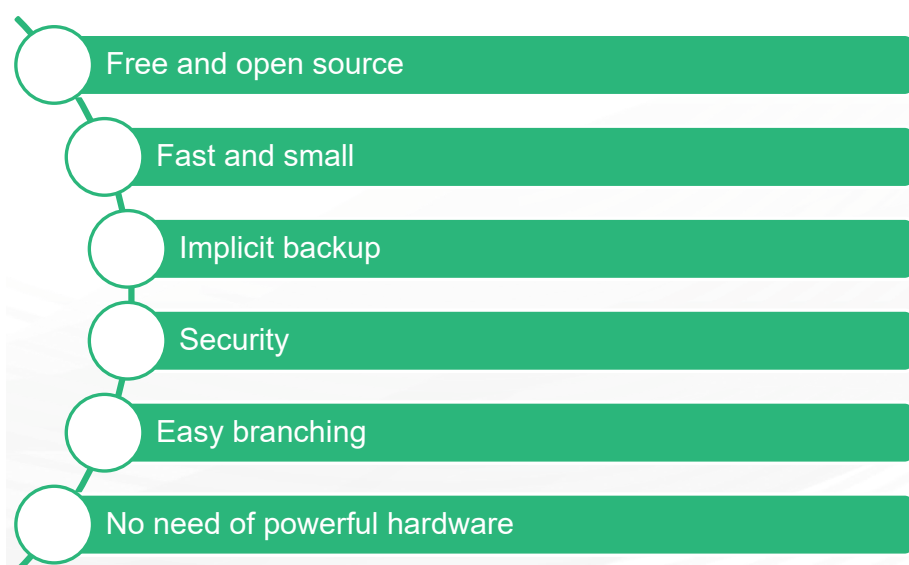
From the outset, Torvalds had one philosophical goal for Git—to be the anti-CVS—plus three usability design goals:

- Support distributed workflows similar to those enabled by BitKeeper
- Offer safeguards against content corruption
- Offer high performance

Within days of development, in June 2005, Linus' git revision control system had become fully self-hosting and within a few weeks it was ready to host Linux kernel development. Within a few months there was a huge participation from the community for further development of Git. And now, it has become an essential product in every developer's life.

6.3 Advantages of Git

Git has become essential in the life of many developers, because of the features it offers. The following diagram shows the primary set of advantages it offers.



Git has become so popular version control system among a vast majority of developers because of the array of advantages it offers. Some of the advantages of Git are listed below:

Free and open source: Git is an open source distributed version control system. It is available freely over the internet. Git can be used to manage any number of software projects without paying a single penny. Being open source, its source code can be downloaded and changes can also be made to it according to the requirements.

Fast and compact: Most of the operations in Git are performed locally, it is super fast to work with. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. Git is primarily written in C, because of which the runtime overheads associated with other high-level languages are avoided. Git mirrors entire source code repository, but the amount of data on the client side is very less. This clearly shows the efficiency of Git at compressing and storing data on the client side.


Implicit backup: When there are multiple copies of data, chances are very less that the data is lost. Data on the client side mirrors the entire repository, hence it can be used in the event of a crash or disk corruption.

Security: Git makes use of a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

Easier branching: CVCS uses copy mechanism. If we create a new branch, it will copy the entire code to the new branch, which is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

No need of powerful hardware: In centralized VCS, the central server needs to be highly efficient and powerful to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, hardware limitations of the server cause a severe impact on performance. In case of distributed VCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

What did You Grasp?



- Fill in the blank.
Git is originally in written in _____.
 A) Python
 B) C
 C) Ruby
 D) Java

In a nutshell, we learnt:



1. History of version control systems
2. Basic operations that are done in a VCS
3. Some examples for the different types of VCS
4. Basics of Concurrent Versions System (CVS)
5. Overview of Subversion, its features and limitations
6. Mercurial and its features
7. Overview of Git and the history of Git
8. Advantages of Git