**IT350 [SE] Assignment 3: A survey on Version Control systems.**

P Pavan

15IT224

**Version Control System**

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

For almost all software projects, the source code is like the crown jewels - a precious asset whose value must be protected. For most software teams, the source code is a repository of the invaluable knowledge and understanding about the problem domain that the developers have collected and refined through careful effort. Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team. Further, in all software development, any change can introduce new bugs on its own and new software can't be trusted until it's tested. So testing and development proceed together until a new version is ready.

Good version control software supports a developer's preferred workflow without imposing one particular way of working. Ideally it also works on any platform, rather than dictate what operating system or tool chain developers must use. Great version control systems facilitate a smooth and continuous flow of changes to the code rather than the frustrating and clumsy mechanism of file locking - giving the green light to one developer at the expense of blocking the progress of others.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

### **Local Version Control Systems**

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

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.



Figure 1. Local version control.

One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. 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**

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



Figure 2. Centralized version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it’s far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

### **Distributed Version Control Systems**

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.



Figure 3. Distributed version control.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

**1.Git**

Git is a version control system for tracking changes in computer files and coordinating work on

those files among multiple people. It is primarily used for source code management in software

development, but it can be used to keep track of changes in any set of files. Git was created by

Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers

contributing to its initial development. Its current maintainer since 2005 is Junio Hamano.As a

distributed revision control system it is aimed at speed,data integrity, and support for distributed,

non-linear workflows. As with most other distributed version control systems, and unlike most

client–server systems, every Git directory on every computer is a full-fledged repository with

complete history and full version tracking abilities, independent of network access or a central

server.

**Features**

**->Feature Branch Workflow**

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control-systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.

Feature branches provide an isolated environment for every change to your codebase. When a

developer wants to start working on something—no matter how big or small—they create a new

branch. This ensures that the master branch always contains production-quality code.

Using feature branches is not only more reliable than directly editing production code, but it also

provides organizational benefits. They let you represent development work at the same granularity as the your agile backlog.

**->Distributed Development**

Git,is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.

Having a full local history makes Git fast, since it means you don’t need a network connection to

create commits, inspect previous versions of a file, or perform diffs between commits.

Distributed development also makes it easier to scale your engineering team. If someone breaks the production branch in a centralized system, other developers can’t check in their changes until it’s fixed. With Git, this kind of blocking doesn’t exist. Everybody can continue going about their

business in their own local repositories. And, similar to feature branches, distributed development creates a more reliable environment.Even if a developer obliterates their own repository, they can simply clone someone else’s and start anew.

**->Pull Requests**

A pull request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

Since they’re essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren’t destroying the entire project by treating pull requests as a formal code review.

**->Community**

In many circles, Git has come to be the expected version control system for new projects. If your

team is using Git, odds are you won’t have to train new hires on your workflow, because they’ll

already be familiar with distributed development.

In addition, Git is very popular among open source projects. This means it’s easy to leverage 3rd-party libraries and encourage others to fork your own open source code.

**->Faster Release Cycle**

The ultimate result of feature branches, distributed development, pull requests, and a stable

community is a faster release cycle. These capabilities facilitate an agile workflow where

developers are encouraged to share smaller changes more frequently. In turn, changes can get

pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.

**Advantages**

**Distributed model:** This means your work is your own. You can let others see only what is

necessary. Not everything has to be public. There are other advantages to the distributed

model, such as the speed and possibility of working offline

**Branching and merging are easy:** Branching is a walk in the park. It feels like a natural

part of the workflow. They are cheap, fast and consume very little space so that you can

branch whenever you want. This means you can sandbox your features and ideas till they are

ready for the mainstream.

**Workflow is flexible:** Compared to Centralized VCS, git has the qualities that allow to

choose your own workflow. It can be as simple as a centralised workflow to as hierarchical

as the dictator-lieutenant workflow. Use the process that best fits you.

**Data integrity is assured:** Because git uses SHA1 trees, data corruption due to external

reasons can be easily detected.

**Disadvantages**

**Steep learning curve:** Many commands with many options, some commands are non-

intuitive and need a level of understanding the internals of git, commands and arguments are

inconsistent to some degree

**Binary files are a big no:** If your project has non-text files that are updated frequently

(images for websites or MS Office documents), then git becomes bloated and slow.

**Basic Commands**

|  |  |  |
| --- | --- | --- |
| **Tell Git who you are** | Configure the author name and email address to be used with your commits.  Note that Gitstrips some characters (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |
| **Create a new local repository** |  | git init |
| **Checkout a repository** | Create a working copy of a local repository:  For a remote server, use: | git clone /path/to/repository  git clone username@host:/path/to/repository |
| **Add files** | Add one or more files to staging (index): | git add <filename>  git add \* |
| **Commit** | Commit changes to head (but not yet to the remote repository):  Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -m "Commit message"  git commit -a |
| **Push** | Send changes to the master branch of your remote reposi. | git push origin master |
| **Status** | List the files you've changed and those you still need to add or commit: | git status |
| **Connect to a remote repository** | If you haven't connected your local repository to a remote server, add the server to be able to push to it:  List all currently configured remote repositories: | git remote add origin <server>  git remote -v |
| **Branches** | Create a new branch and switch to it:  Switch from one branch to another:  List all the branches in your repo, and also tell you what branch you're currently in:  Delete the feature branch:  Push the branch to your remote repository, so others can use it:  Push all branches to your remote repository:  Delete a branch on your remote repository: | git checkout -b <branchname>  git checkout <branchname>  git branch  git branch -d <branchname>  git push origin <branchname>  git push --all origin  git push origin :<branchname> |
| **Update from remote repository** | Fetch and merge changes on the remote server to your working directory:  To merge a different branch into your active branch:  View all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging:  After you have manually resolved any conflicts, you mark the changed file: | git pull  git merge <branchname>  git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch>  git add <filename> |
| **Tags** | You can use tagging to mark a significant changeset, such as a release:  CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using:  Push all tags to remote repository: | git tag 1.0.0 <commitID>  git log  git push --tags origin |
| **Undo local changes** | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept.  Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git checkout -- <filename>  git fetch origin  git reset --hard origin/master |
| **Search** | Search the working directory for foo(): | git grep "foo()" |

**2. Mercurial**

Mercurial is a distributed revision-control tool for software developers. It is supported on Microsoft-Windows and Unix-like systems, such as FreeBSD, macOS and Linux. Matt Mackall originated Mercurial and serves as its lead developer. Mercurial's major design goals include high performance and scalability, decentralized, fully distributed collaborative development, robust handling of both plain text and binary files, and advanced branching and merging capabilities, while remaining conceptually simple. It includes an integrated web-interface.

**Advantages**

**Simplicity:**

The command line UI is \_very\_ predictable and easy to learn. The names of many basic commands have the same meaning as those of CVS and Subversion.

- This way people coming from those other VC systems can really expect "hg add" to add a file to

the repository and not do something surprising or even wrong.

- Easy command names and predictably, very uniform options for those commands means that it is often quite easy to ‘guess’ what a command.

**It is easy to use & write extensions:**

The existing extensions cover a lot of the functionality a new user might require. Writing new ones

is easy and the Wiki has nice documentation describing how to do it. The code of the existing

extensions is a nice read too; seeing how other people have hooked into Hg and extended its core

functionality, using a wonderful language like Python helps a lot when one is trying to create new

extensions from scratch or to extend one of the current extensions.

**Disadvantage**

**Lack of partial checkout or partial clone support:**

Working with several-hundred-MB or multi-GB workspaces is difficult if you have to use MQ or

something else that requires frequent "update" or "status" commands. Being able toclone only

parts of a tree or to check out only parts of tree from a full clone would be nice.

**Basic Commands**

|  |  |
| --- | --- |
| **Task** | Mercurial (Hg) commands |
| Create a **new local directory:** | hg init |
| **Connect** your local repository to a remote  server: | hg push <server\_URL> |
| **Copy** a remote repository to your local system: | hg clone <URL\_to\_repository> |
| **Add** a specific file to staging (Git) or after a new file is created (Hg): | hg add <filename> |
| **Add** all changes to staging (Git) or all new files  (Hg): | hg add |
| **Commit** changes locally: | hg commit -m "<message>" |
| **Push** changes to your remote repository: | hg push |
| **List** the status of the files you've changed and  those you still need to add or commit: | hg status |
| **Create** a new branch (Git) or bookmark (Hg)  and switch to it: | hg bookmark <bookmark\_name> |
| **Switch** from one branch or bookmark to another: | hg bookmark <bookmark\_name> |
| **List** all the branches/bookmarks in your repo  with an indication of the one you are on: | hg bookmarks |
| **Delete** the feature branch/bookmark: | hg bookmark -d <bookmark\_name> |
| **Push** the branch/bookmark to your remote  repository: | hg push -B <bookmark\_name> |
| **Fetch and merge** changes on the remote server to your working directory: | hg pull -u |
| **Merge** two different revisions into one: | hg merge |
| **Show all changes** made since the last commit: | hg diff |

**3. SVN(Subversion)**

Apache Subversion (often abbreviated SVN, after its command name svn) is a software versioning and revision control system distributed as open source under the Apache License. Software developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation.

Subversion was created by CollabNet Inc. in 2000, and is now a top-level Apache project being

built and used by a global community of contributors.

**Features**

**Directories are versioned.**

Subversion versions directories as first-class objects, just like files.

**Copying, deleting, and renaming are versioned.**

Copying and deleting are versioned operations. Renaming is also a versioned operation, albeit with some quirks.

**Atomic commits.**

No part of a commit takes effect until the entire commit has succeeded. Revision numbers are per-commit, not per-file, and commit's log message is attached to its revision, not stored redundantly in all the files affected by that commit.

**Branching and tagging are cheap (constant time) operations.**

There is no reason for these operations to be expensive, so they aren't.

Branches and tags are both implemented in terms of an underlying "copy" operation. A copy takes

up a small, constant amount of space. Any copy is a tag; and if you start committing on a copy, then it's a branch as well.

**Merge tracking.**

Subversion 1.5 introduces merge tracking: automated assistance with managing the flow of changes between lines of development, and with the merging of branches back into their sources. The 1.5 release of merge tracking has basic support for common scenarios; we will be extending the feature in upcoming releases.

**File locking.**

Subversion supports (but does not require) locking files so that users can be warned when multiple people try to edit the same file. A file can be marked as requiring a lock before being edited, in which case Subversion will present the file in read-only mode until a lock is acquired.

**Parseable output.**

All output of the Subversion command-line client is carefully designed to be both human readable

and automatically parseable; scriptability is a high priority.

**Natively client/server, layered library design with clean APIs.**

Subversion is designed to be client/server from the beginning; the code is structured as a set of

modules with well-defined interfaces, designed to be called by other applications.

**Binary files handled efficiently.**

Subversion is equally efficient on binary as on text files, because it uses a binary diffing algorithm

to transmit and store successive revisions.

**Costs are proportional to change size, not data size.**

In general, the time required for a Subversion operation is proportional to the size of the changes

resulting from that operation, not to the absolute size of the project in which the changes are taking place.

**Bindings to programming languages.**

The Subversion APIs come with bindings for many programming languages, such as Python, Perl, Java, and Ruby. (Subversion itself is written in C.)

**Advantages**

1. Because of its centralized nature, Subversion offers better management of binary files out-of-the-box, as only the working tree is on client machines.
2. Subversion is easier to grasp for non-technical users
3. Subversion has more safe history, e.g. it would require someone to have access to the centralized server to change the history.

**Disadvantages**

1. Good only if a single developer is involved in the projects
2. Even if the repository is readily available to everyone via network share, this is a bad idea.It removes any layers of protection between the users and the repository
3. Users can accidentally (or intentionally) corrupt the repository database, it becomes hard to take the repository offline for inspection or upgrade.

**Basic Commands**

1. SVN Checkout – Create working copy

2. SVN Commit – Save changes to the repository

3. SVN List – Lists directory entries

4. SVN Add – Add a new file to SVN repository

5. SVN Delete – Removing a file from repository

6. SVN Diff – Display the difference

7. SVN Status – Status of the working copy

8. SVN Log – Display log message

9. SVN Move – Rename file or directory

10. SVN Update – Update the working copy.

**For the Employee Management System Software project, Git will be used as the version control system.**