Subversion and Git - a comparative study

Author: Daniel-Ioan Giuriciu, anul IV, AC-CTI Romana

Version Control: The Why?

What is “version control”, and why should you care? A version control system is a system that records changes to a file or set of files over time so that you can recall specific versions later.

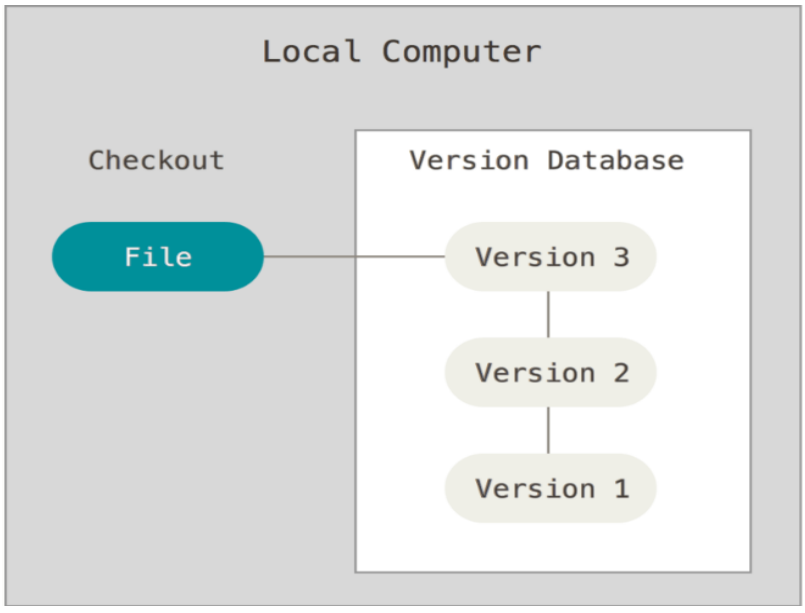
For us software developers and, really digital developers of any kind, storing different versions of our work not only sounds appealing but is essential. The safety net, of having a fuctioning version/implementation cuts our wings out free and allows us to experiment.

Forms of version control: The How?

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:

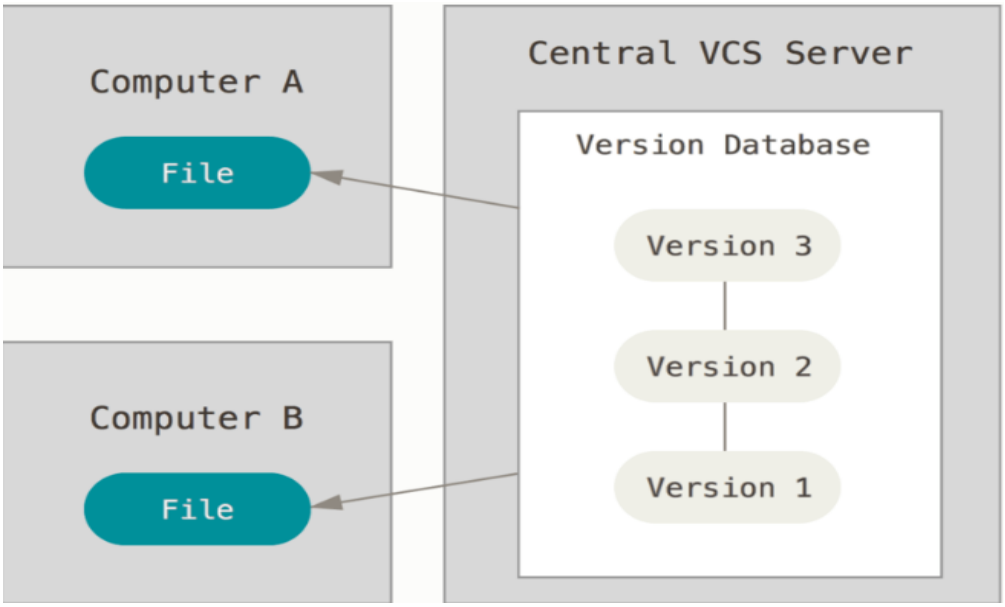
Local VCS: A not so bad attempt



This is all fine and dandy if you're working alone, which, aside for the select few, is highly unlikely in our line of work.

A much more dynamic and team-supportive approached is what is calle the **Centralized VCS**

Centralized VCS: A valiant effort



The benefits of this **push-pull architecture** are more than obvious.

- Everyone knows the actual state of the project
- Much more easy to manage from an admin's point of view
- Security features are now available(e.g. Read/Write permissions)

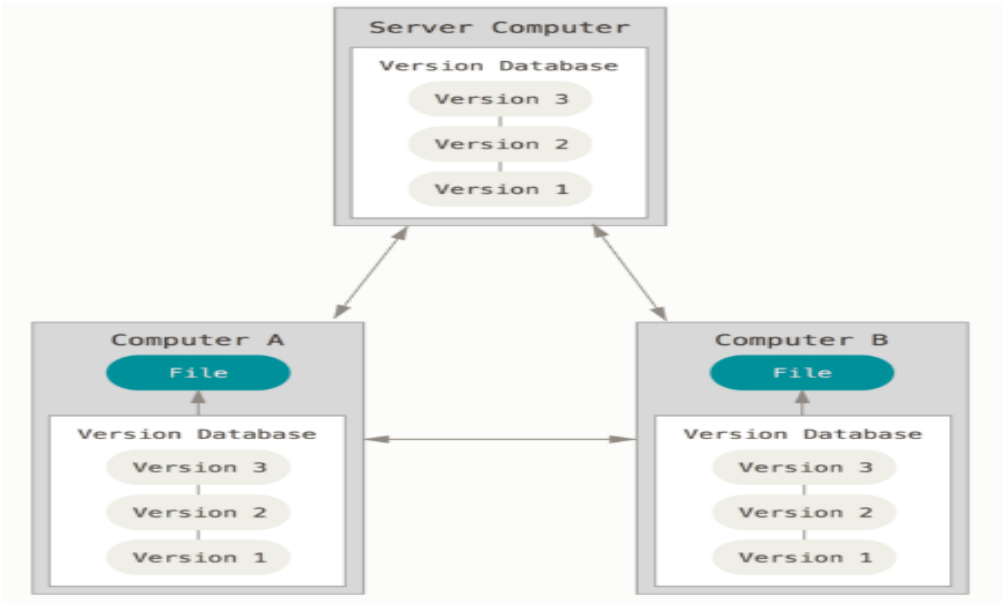
However, this setup also has some **serious** downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard 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

Distributed VCSs: The home run

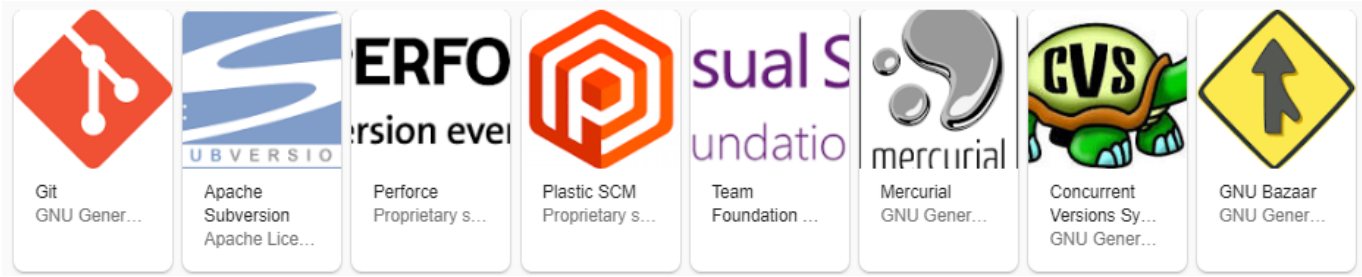
This is where distributed version control systems step in and save the day - in a distributed vcs you don't checkout the latest snapshot of a file, rather you mirror an entire working copy of the repository, history included.

> How does this solve the aforementioned problem?

In the case of the root node failing, because each and every client node has a virtually entire, working copy of the project, it can then restore the root node, updatind it with the necessary changes once it gets back online. Even better, if configured so, any client node can become a secondary root node, thus, taking up the responsibilities of the root node in case of failure.



VCS Options: The Who?



The market is not scarce when it comes to the pool of available version control systems, but, we'll focus on the two most *popular* version control systems: **Subversion** and **Git**

Subversion

SVN represents the most popular **centralized** version control system on the market. Remember, with a centralized system, all files and historical data are stored on a **one and only** central server. Developers commit their changes directly to that central server repository.

Work is comprised of three parts:

- Trunk: The trunk is the hub of your current, stable code and product. It only includes or it **SHOULD** only include: tested, unbroken code.
- Branches: Here is where you house new code and features. Using a copy of the trunk code, team members conduct research and development in the branch. Doing so allows each team member to work on the enhanced features without disrupting each other's progress.
- Tags: Consider tags a duplicate of a branch at a given point in time. Tags aren't used during development, but rather during deployment after the branch's code is finished. Marking your code with tags make it easy to review and, if necessary, revert your code.

A workflow example: to create a new feature you first branch the code from the trunk, i.e. take an exact copy of the trunk at a give point in time and place it into a new folder within the branches area. Then you work on your feature. When you're done, you merge your changes back into the trunk.

The benefit of branching is the ability to make commits into the branch without breaking the trunk. You only merge into the trunk when your code is error-free. This keeps your trunk stable. And users generally appreciate how easy it is to use and understand SVN.

However, working on one central server means there is a single point of failure. If there is an error, it can destroy all builds. Limited offline access is also a frequent point of complaint.

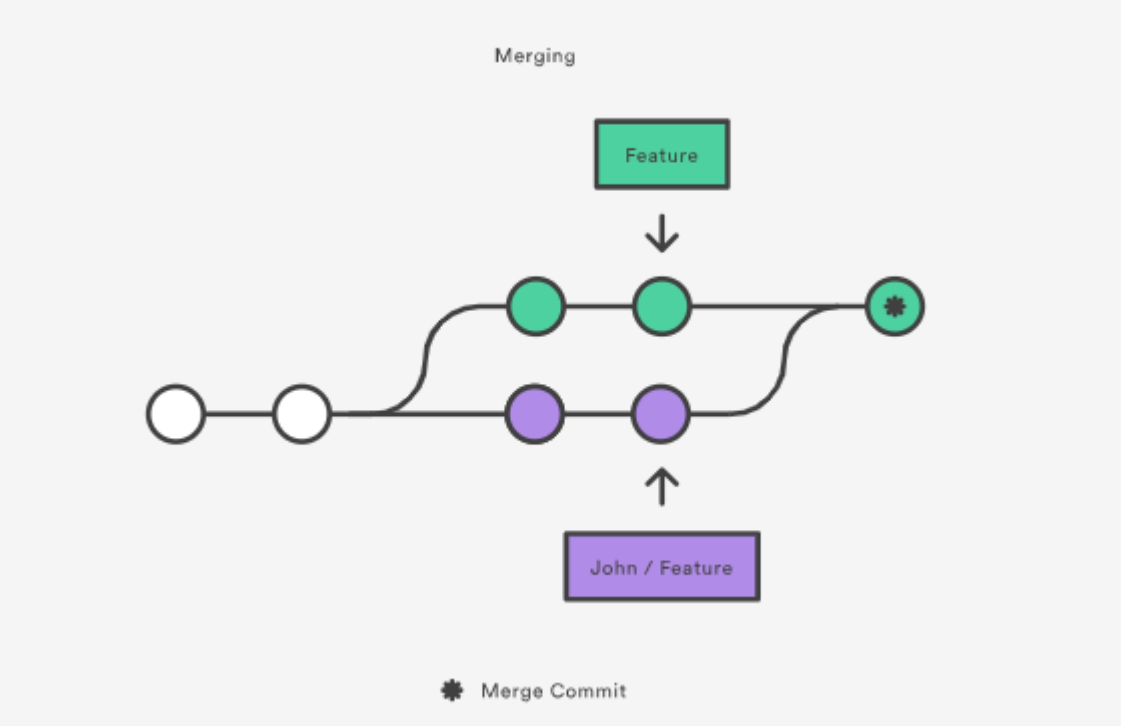
Git

Unlike SVN, Git utilizes multiple repositories, it's a **distributed version control system**: a central repository and a series of local repositories. Local repositories are exact copies of the central repository complete with the entire history of changes.

The Git workflow is similar to SVN, but with few extra steps:

> Let's assume we get co-opted into an already existing project, which is a very usual user story for a developer

- Checkout: Firstly you would get a copy of the repository that you would have to work upon, again, this is a fully working copy of it, history include, not just a snapshot of the/some files.
- Branch off: Branch off of your local master
- Develop your features
- Stage changes: An extra step that is not present in Subversion, where, once you commit, it's gone, forever. Staging changes allows for some last minute checking and a more selective commit
- Commit changes: The changes that were staged, get commited to the local repository
- Push to remote: Finally, push out the your local/branch to the remote/branch and merge the changes.



Many people prefer Git for version control for a few reasons:

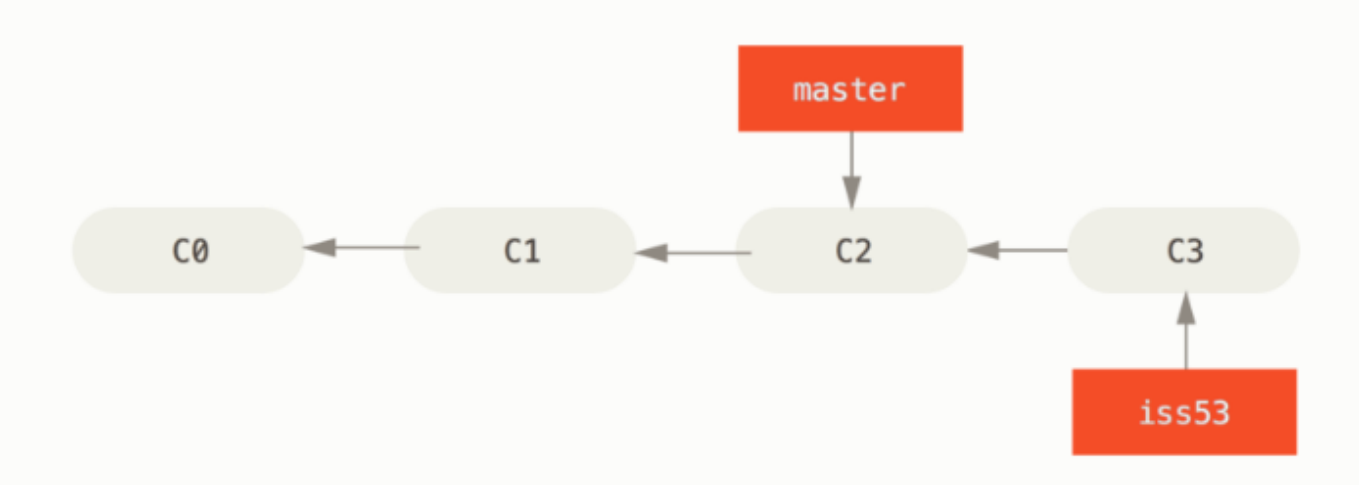
- It’s faster to commit. Because you commit to the central repository more often in SVN, network traffic slows everyone down. Whereas with Git, you’re working mostly on your local repository and only committing to the central repository every so often.
- No more single point of failure. With SVN, if the central repository goes down or some code breaks the build then no other developers can commit their code until the repository is fixed. With Git, each developer has the own repository, so it doesn’t matter if the central repository is broken. Developers can continue to commit code locally until the central repository has been fixed, and then they can push their changes.
- It’s available offline. Unlike SVN, Git can work offline, allowing your team to continue working without losing features if they lose connection.
- The extra steps whichi initially may seem as overhead, actually significantly decrease bad or trigger happy commits.

Merging

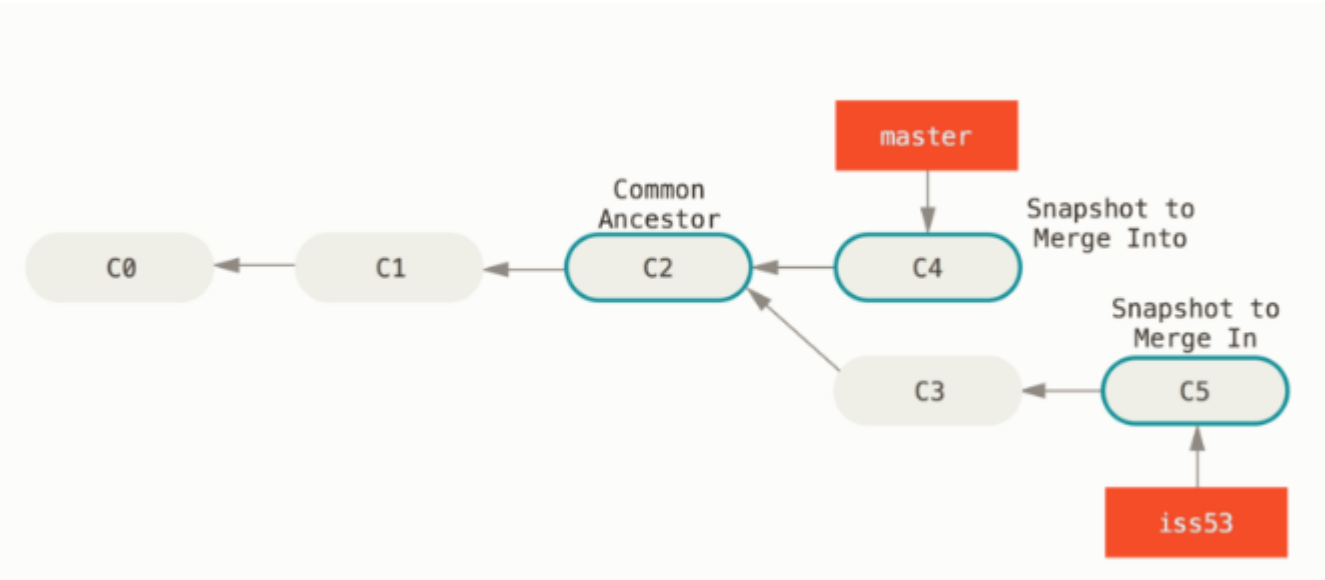
Another particularly important aspect is merging and how it is facilitated(asisted) by the VCS.

By far, Subversion's largest complaint amongst all has been the difficulties encountered when merging. Let's find out why, but first, a bried reminder about what merging actually is:

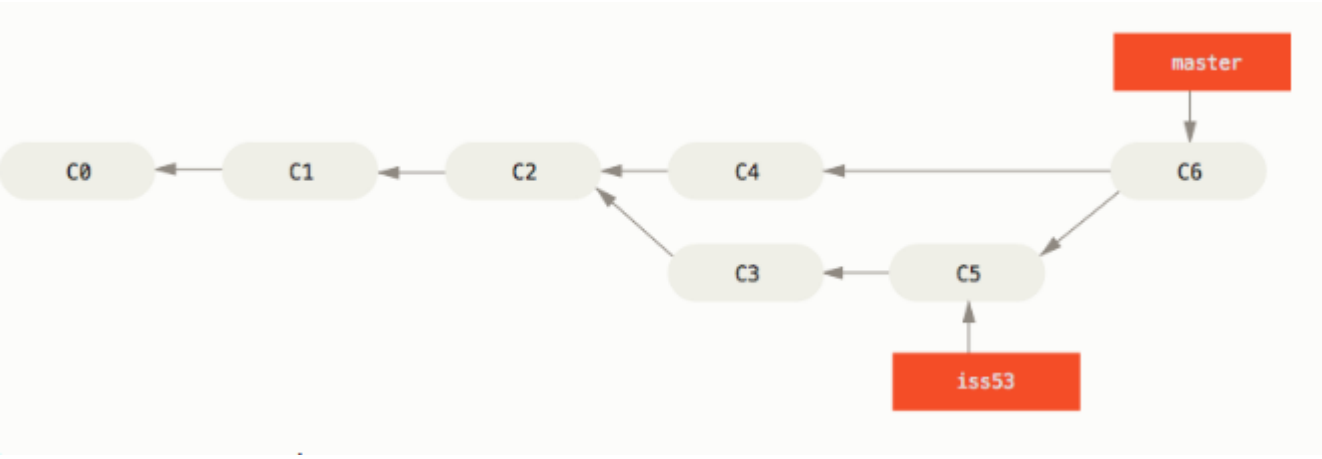
Let's assume that you branch off master/trunk and start doing some work:



Presumably some error is found on master that doesn't really affect you, so a hotfix is due



Now, once you are finished with your work and ready to share it the rest of the team you push it back to master, and merge it with the central, root repository.



Here, a special merge commit is made in case any **conflicts** appear.

Conflicts

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, the VCS won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix branch, you'll get a merge conflict:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

It is up to you to decide how to handle the changes, which to keep, yours, theirs, both? Specialized tools are usually used for this, highlithing and offering quick navigation between conflicts makes our lives so much more easier.

Aside from some overhead, this seems pretty manageable.

Merging in Subversion

So why does everybody complain about conflicts and the general merge workflow in Subversion?

Ponder this example:

```
      1   2   4     6     8
trunk o-->o-->o---->o---->o
       \
        \   3     5     7
b1       +->o---->o---->o
```

When we want to merge b1's changes into the trunk we'd issue the following command, while standing on a folder that has trunk checked out:

```
svn merge -r 2:7 {link to branch b1}
```

... which will attempt to merge the changes from b1 into your local working directory. And then you commit the changes after you resolve any conflicts and tested the result. When you commit the revision tree would look like this:

```
      1   2   4     6     8   9
trunk o-->o-->o---->o---->o-->o      "the merge commit is at r9"
       \
        \   3     5     7
b1       +->o---->o---->o
```

However this way of specifying ranges of revisions gets quickly out of hand when the version tree grows as subversion didn't have any meta data on when and what revisions got merged together. Ponder on what happens later:

```
           12        14
trunk  …-->o-------->o
                                    "Okay, so when did we merge last time?"
              13        15
b1     …----->o-------->o
```

This is largely an issue by the repository design that Subversion has, in order to create a branch you need to create a new virtual directory in the repository which will house a copy of the trunk but it doesn't store any information regarding when and what things got merged back in. That will lead to nasty merge conflicts at times.

What was even worse is that Subversion used two-way merging by default, which has some crippling limitations in automatic merging when two branch heads are not compared with their common ancestor.

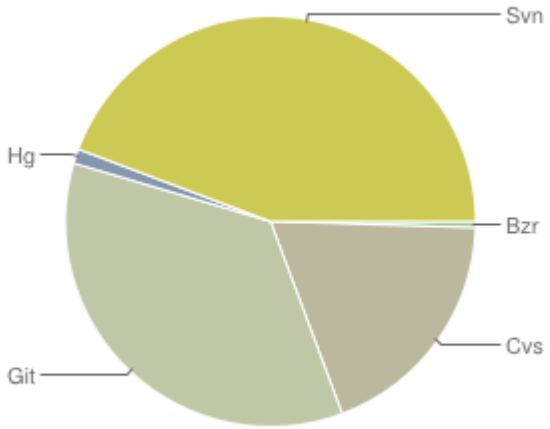
Which one?

As always, there is no **one perfect tool** for a job, however there are two things worthy of consideration when thinking about which one to go in depth on, and which to just have basic abilities; we cannot really ignore one completely, due to their popularity. The chances to bump into either git/svn are very high.

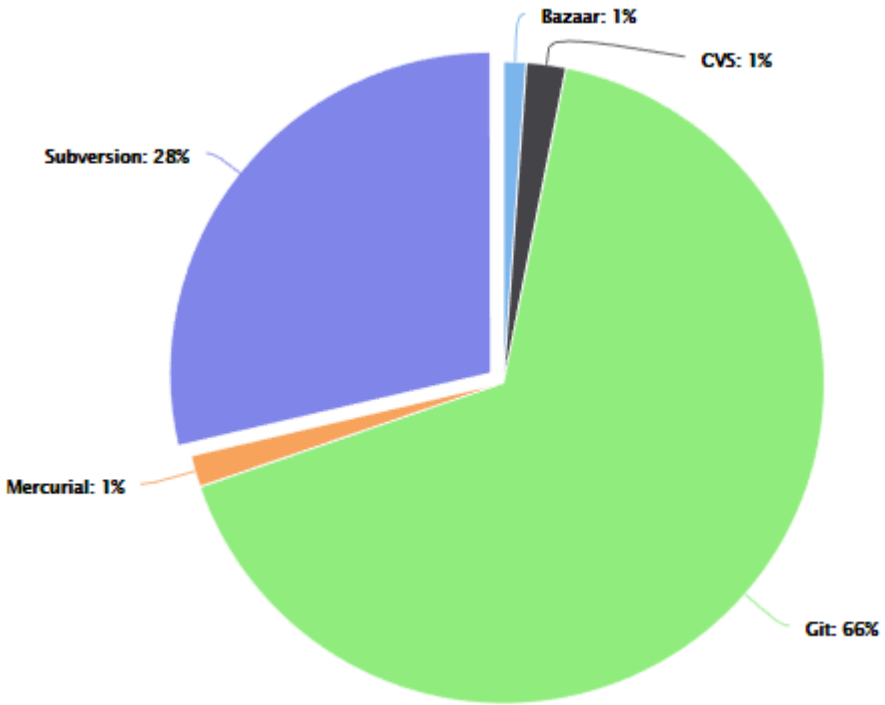
Usage/Market Share

It's best to be an expert in the most popular tool, because usually that's the highest one in demand. Evolution of the market share has shifted drastically in the last few years:

Here is the market share for early 2012:



And then the market share for late 2018:



It is quite evident that **Git** has gained the upper hand and cemented itself as the VCS system to chose.

Because of it's high pased acquisition of the repository market various tools such as

- Visual Studio
- Visual Studio Code
- IDEA Integrated Development Environments

Now provide fully integrated support for git operations, so you don't have to shift to the terminal to manage the work you just did.

Subversion has extensions and plugins of the sort, but they are not yet as seamlessly integrated into the respective systems.

What it does, have, is a nice wrapper/graphical user interface in Toirtoise SVN which helps a lot by greatly simplifying your day to day workflow with SVN.

Conclusion

As stated before there is no one perfect tool, however, based on:

- The evolution of the market
- Ease of use and general user experience
- Integration with popular tool
- Robustness and security
- Performance

my conclusion is as follows:

Master Git and know subsitence level **Subversion**.

Bibliography

- Git docs
- The Git book(second edition)
- backlog blogpost
- mentormate article
- stack overflow question