How to git

A quick intro

Georg Stadler
stadler@cims.nyu.edu

March 31, 2022

Git history and overview

- Git is a version control systems (keeps all previous versions), others include: subversion, cvs, mercurial
- Useful for code, but also any other text-based projects: papers, LaTeX, letters, class material, ...
- Git has similarities to Dropbox, Google Drive, but is superior for cooperations (also with yourself)
- Git is open-source, started in 2005, developed by developers of Linux kernel (Linus Torvalds) after disagreement with different commertial repo
- Github, Bitbucket, Gitlab etc. mostly provide storage and web platform for git, but have nothing to do with the software itself
- Git is local, you can (and should!) use it by yourself

Why Use Version Control?

Slides adapted from Andreas Skielboe

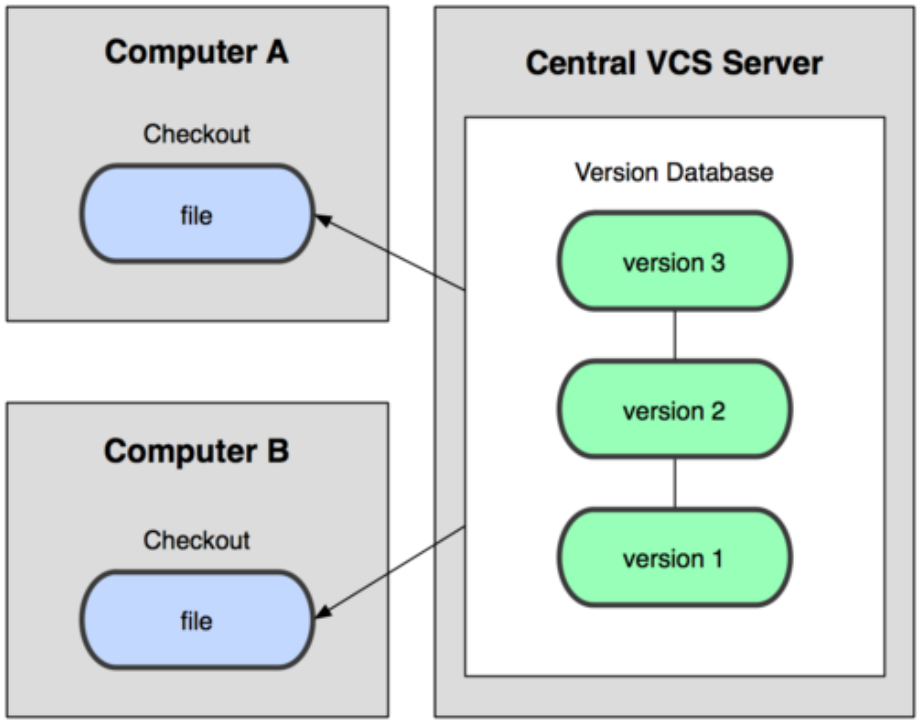
A Version Control System (VCS) is an integrated fool-proof framework for

- Backup and Restore
- Short and long-term undo
- Tracking changes
- Synchronization
- Collaborating
- Sandboxing

... with minimal overhead.

Centralized Version Control Systems

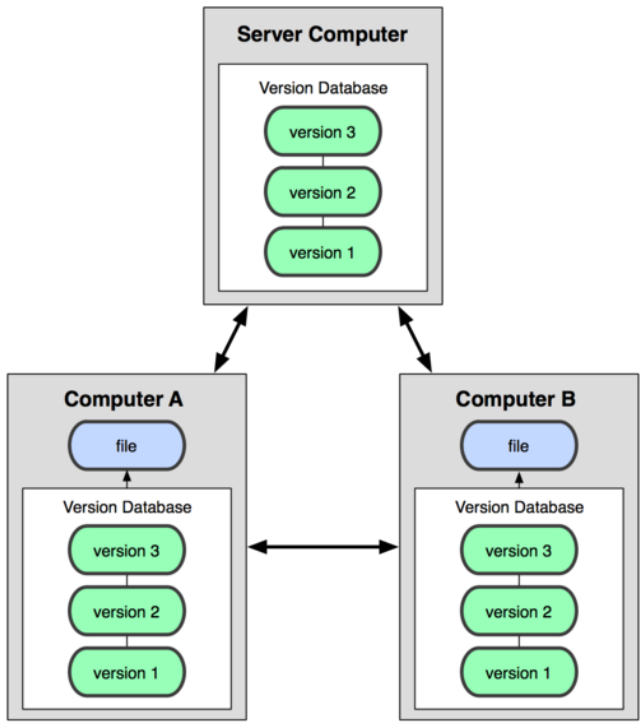
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.



Introduces problems: single point of failure, inability to work offline.

Distributed Version Control Systems

To overcome problems related to centralization, distributed VCSs (DVCSs) were invented. Keeping a complete copy of database in every working directory.



Actually the most **simple** and most **powerful** implementation of any VCS.

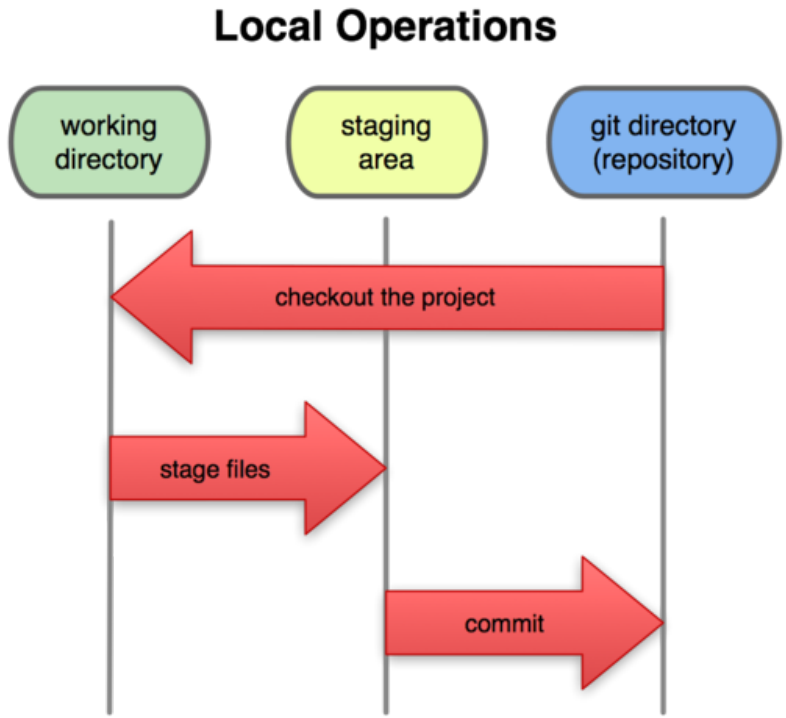
Git Basics - The Git Workflow

The simplest use of Git:

- **Modify** files in your *working directory*.
- **Stage** the files, adding snapshots of them to your *staging area*.
- **Commit**, takes files in the staging area and stores that snapshot permanently to your *Git directory*.

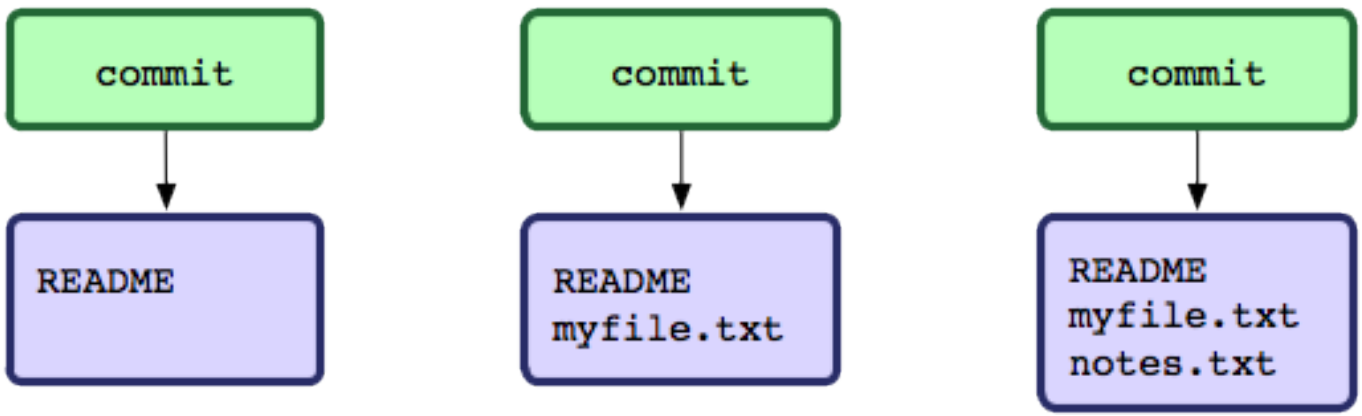
Git Basics - The Three States

The three basic states of files in your Git repository:



Git Basics - Commits

Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.



Each and every commit can always be looked at and retrieved.
In Git **all remotes are equal**.

Git Basics - Working with remotes

The easiest commands to get started working with a remote are

- *clone*: Cloning a remote will make a complete local copy.
- *pull*: Getting changes from a remote.
- *push*: Sending changes to a remote.

Fear not! We are starting to get into more advanced topics. So lets look at some examples.

Git Basics - Advantages

Basic advantages of using Git:

- Nearly every operation is local.
- Committed snapshots are always kept.
- Strong support for non-linear development.

Hands-on - Getting started with remote server

When the remote server is set up with an initialized Git directory you can simply *clone* the repository:

Cloning a remote repository

```
$ git clone <repository>
```

You will then get a complete local copy of that repository, which you can edit.

Hands-on - Getting started with remote server

With your local working copy you can make any changes to the files in your working directory as you like. When satisfied with your changes you add any modified or new files to the staging area using *add*:

Adding files to the staging area

```
$ git add <filepattern>
```

Hands-on - Getting started with remote server

Finally to commit the files in the staging area you run *commit* supplying a *commit message*.

Committing staging area to the repository

```
$ git commit -m <msg>
```

Note that so far **everything is happening locally** in your working directory.

Hands-on - Getting started with remote server

To **share your commits** with the remote you invoke the *push* command:

Pushing local commits to the remote

```
$ git push
```

To recieve changes that other people have pushed to the remote server you can use the *pull* command:

Pulling remote commits to the local working directory

```
$ git pull
```

And **thats it**.

References

Some good Git sources for information:

- Git Community Book - http://book.git-scm.com/
- GitHub - http://github.com/
- Bitbucket - http://bitbucket.com
- Gitlab - http://gitlab.com
- Git from the bottom up - http://ftp.newartisans.com/pub/git.from.bottom.up.pdf
- Understanding Git Conceptually - http://www.eecs.harvard.edu/~cduan/technical/git/
- Git Immersion - http://gitimmersion.com/

What should (not) be added to a repository?

Git tracks only updated files and uses compression to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc. NO! (unless unavoidable)

My rule of thumb: Files in the repository are permanent, only the best should make it in there (it's not your trash can!) They should compile (code/Latex), be (more or less) cleaned up, unless it's avoidable only source/text files.

Some git wisedom/opinion

Should I have a few large repositories or many small ones?

- I recommend many small ones
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

. . . any others??