# Introduction to Version Control with Git

Andreas Skielboe

Dark Cosmology Centre
Niels Bohr Institute

January 18, 2012

# License



See `http://progit.org/`

# Why Use Version Control?

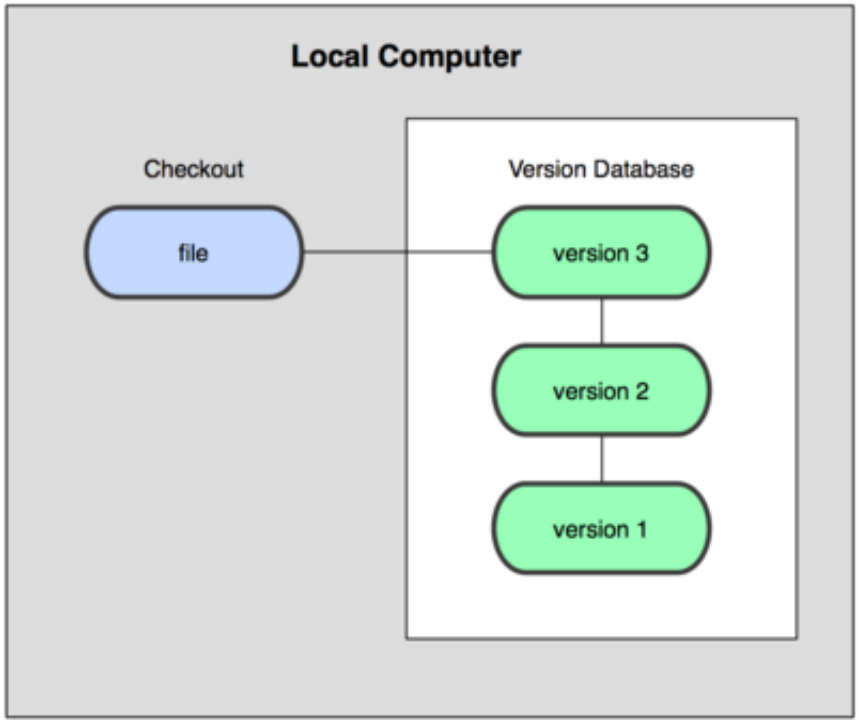
A Version Control System (VCS) is an integrated fool-proof framework for

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

... with minimal overhead.

# Local Version Control Systems

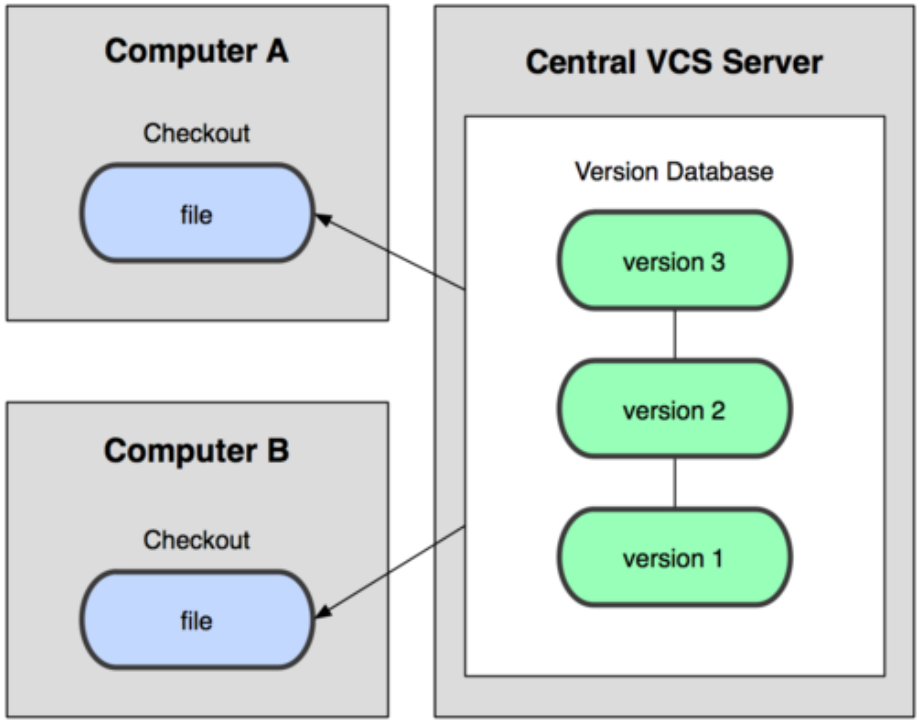
Conventional version control systems provides some of these features by making a local database with all changes made to files.




Any file can be recreated by getting changes from the database and patch them up.

# Centralized Version Control Systems

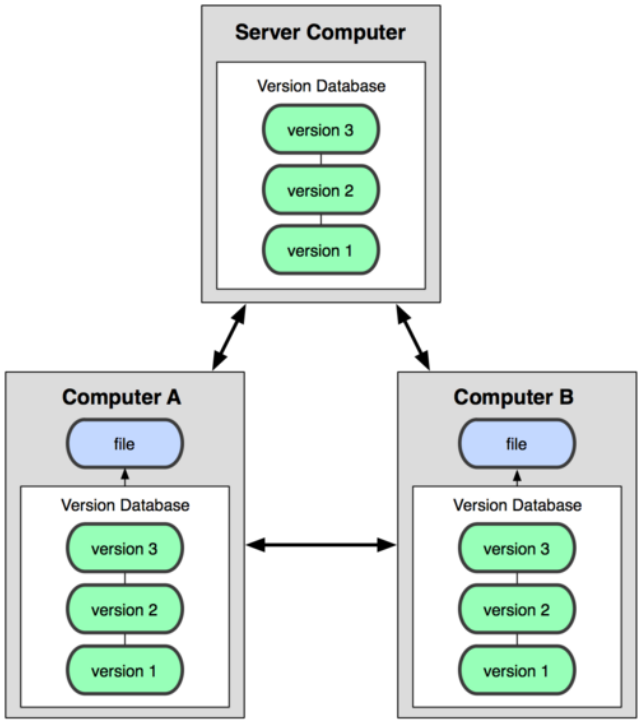
To enable synchronization and collaborative features the database is stored on a central VCS server, where everyone works in the same database.




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

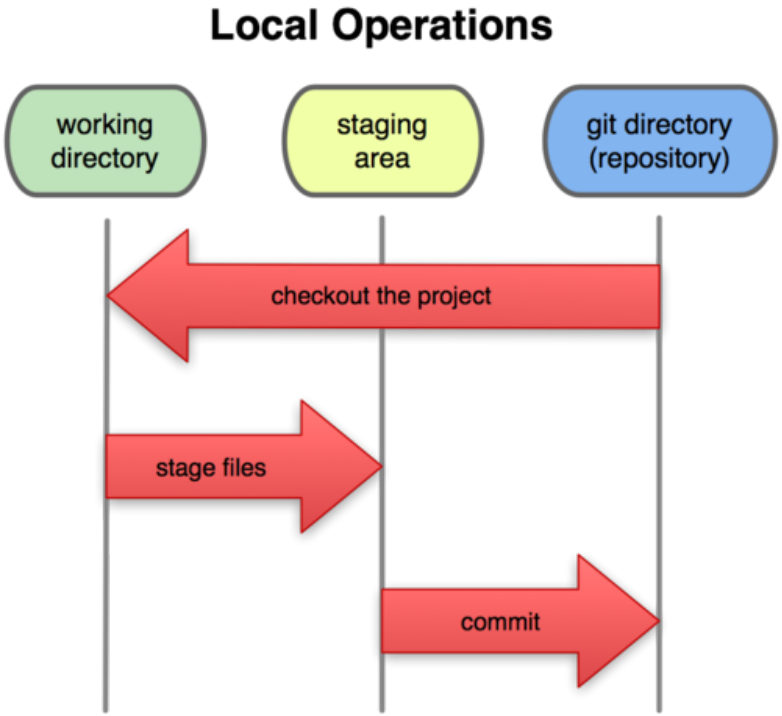
# Git Basics - The Git Workflow

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*.

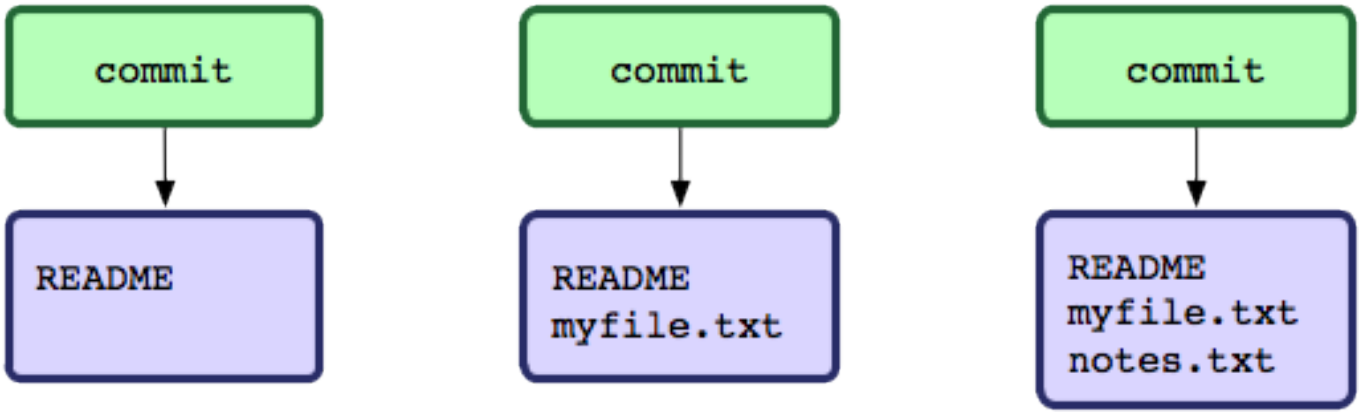
# Git Basics - The Three States

The three basic states of files in your Git repository:



# Git Basics - Commits

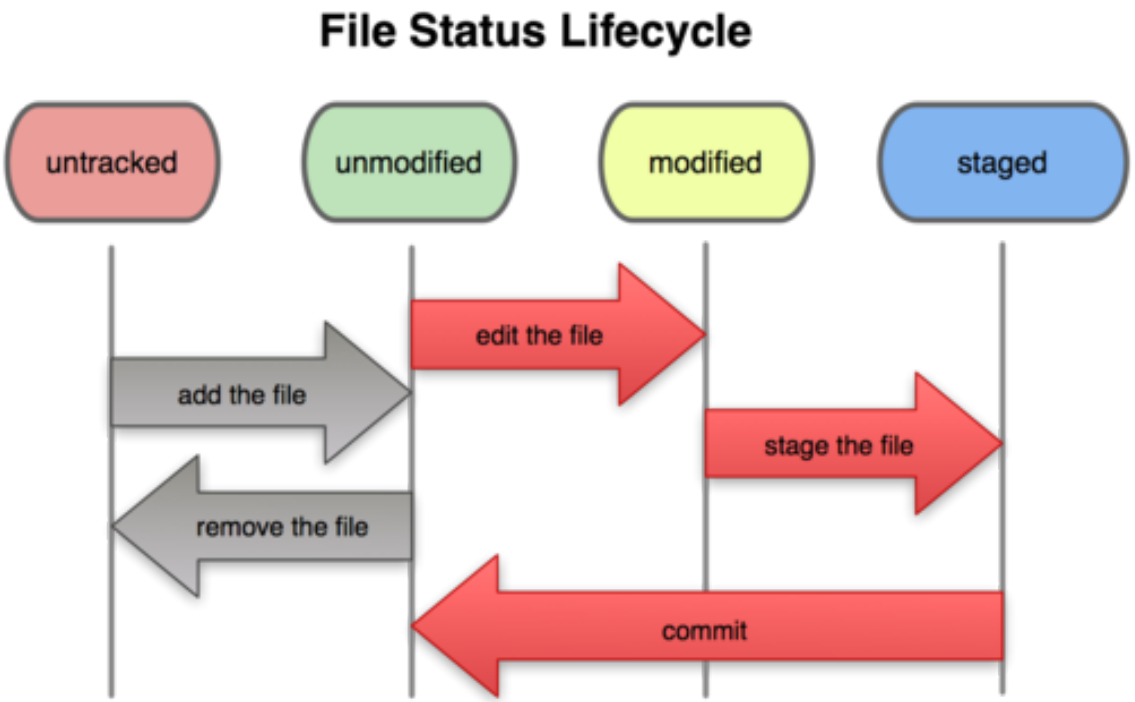
Each commit in the git directory holds a snapshot of the files that were staged and thus went into that commit, along with author information.




Each and every commit can always be looked at and retrieved.

# Git Basics - File Status Lifecycle

Files in your working directory can be in four different states in relation to the current commit.



# Git Basics - Working with remotes

In Git **all remotes are equal**.

A *remote* in Git is nothing more than a link to another git directory.

# 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

Hands-on with Git
(here be examples)

# Hands-on - First-Time Git Setup

Before using Git for the first time:

## Pick your identity

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

## Check your settings

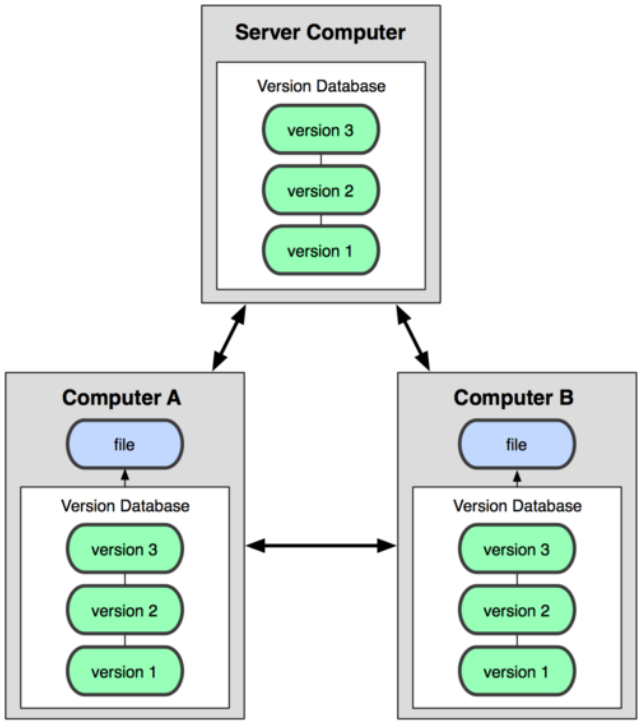
```
$ git config --list
```

## Get help

```
$ git help <verb>
```

# Hands-on - Getting started with a bare remote server

Using a Git server (ie. no working directory / *bare* repository) is the analogue to a regular centralized VCS in Git.



# 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**.

# Hands-on - Summary

Summary of a minimal Git workflow:

- `clone` remote repository
- `add` you changes to the staging area
- `commit` those changes to the git directory
- `push` your changes to the remote repository

- `pull` remote changes to your local working directory

# More advanced topics

Git is a powerful and flexible DVCS. Some very useful, but a bit more advanced features include:

- Branching
- Merging
- Tagging
- Rebasing

# Checkout these slides

The LaTeX-source of these slides is freely available on GitHub.

## GitHub

```
$ git clone git://github.com/askielboe/into-to-git-slides.git
```

Have fun using Git!

# References

Some good Git sources for information:

- Git Community Book - `http://book.git-scm.com/`
- Pro Git - `http://progit.org/`
- Git Reference - `http://gitref.org/`
- GitHub - `http://github.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/`

# Applications

GUIs for Git:

- GitX (MacOS) - `http://gitx.frim.nl/`
- Giggle (Linux) - `http://live.gnome.org/giggle`