# DevOps to Version Control

CIS 2235 Linux System Administration

## What is DevOps?

Good question….in order to define it, we need to understand what came first

## Historical view

Once upon a time….

Coders coded

Sys Admins stood up machines/infrastructure

- testing environments
- staging environments
- production environments

Code would be released to the testing environment and tested by a combination of dev and QA.

Code would then be released into the staging environment

## Historical view - 2

Once in staging, more testing would happen.

Then, the code would be released to production.

This is often a pretty lengthy cycle.

Agile attempted to speed up the development cycle but left the back-end pieces alone.

## Historical View - 3

Each of QA, System Admin/infrastructure and development teams had their silo.

Silos led to decreased communication, and lack of understanding of each other’s roles.

Each group had its own priorities/goals.

While agile helped with turnaround, it didn’t attack this silo issue.

## Enter the idea of DevOps…

Very inconsistently used term, but it is generally recognized as:

Improving collaboration between all stakeholders from planning to delivery and automation of the delivery process to:

- Improve deployment frequency
- Achieve faster time to market
- Lower failure rate of new releases
- Shorter lead time between fixes

## Sounds nice, but how?

As an organization adapts a devOps mindset, it generally move through some stages:

- continuous integration
- continuous delivery
- continuous deployment

## Continuous Integration

- CI from agile
  - The newly developed code is integrated during every check-in
- CI is essential to DevOps because you:
  - Check your code in, compile it, and run basic validation testing

## Continuous Delivery

The next step in moving to a DevOps mindset

- Adds more automation and testing
- Code moves from a developer's hands out to the testing environment
- Might go to the staging environment, too

- Almost deploys without human intervention

## Continuous Deployment

Code goes all the way into production without any human intervention.

- CD is not releasing untested code. The new code is run through automated testing
- Will only be pushed to a small number of users
- Monitoring for quality and stability is key. Automated rollback if there are issues

## Who does this?

Some companies have achieved this, most notably:

- Amazon
- Netflix
- Etsy
- Pinterest
- Google

## This all sounds like code?

Automated testing and deployments need to be "coded."

Developers know how to code, but….

- tend not to understand testing, system builds, security, or deployments

Moving to a DevOps mindset requires that each team has members who understand each piece.

## How?

Organizations need to look at where automation is required to make this all work,

- How to automate builds?
- How to have reproducible builds?
- Environments need to be able to be created

## Tools view

### Set of tools to make this happen:

#### Source code repository

- A place to check in code.
- Old idea, but now want to include information about infrastructure, too.

#### Infrastructure as code!

The idea is to use tools/scripts to build environments from scratch, and save these configurations into the source code repository.

## Tools - 2

### Build server

- An automation tool that compiles the code from the repository into executable code.
  - Jenkins.

### Configuration management

- CM tools allow you to define the configuration of a server or an environment.
  - Examples: Puppet, Chef

## Tools - 3

Virtual infrastructure

- More and more, infrastructure is being virtualized or containerized.

You have cloud providers like

Amazon Web Services

Microsoft Azure

which allows you to programmatically create new machines with CM tools like Puppet or Chef

## Tools - 4

- Pipeline automation
  - Tools like Ansible combine all this:
    - cloud provisioning
    - configuration management
    - application deployment
  - So everything can be deployed from a developer’s hands to an appropriate environment.

## Future?

The need for faster and faster yet accurate deployment of apps is here to stay.

While it is still unclear how DevOps will evolve, it is an increasingly important area.

## Version Control Systems

# Version Control Systems

## Purposes of VCS

- Keep track of all past revisions of code
- Allow multiple programmers to edit the code
- Store the changes in a repository
- Programmers edit/test in their own personal workspace
- Export any version of the software to a release directory

## Version Control Systems

Application for system administrators.

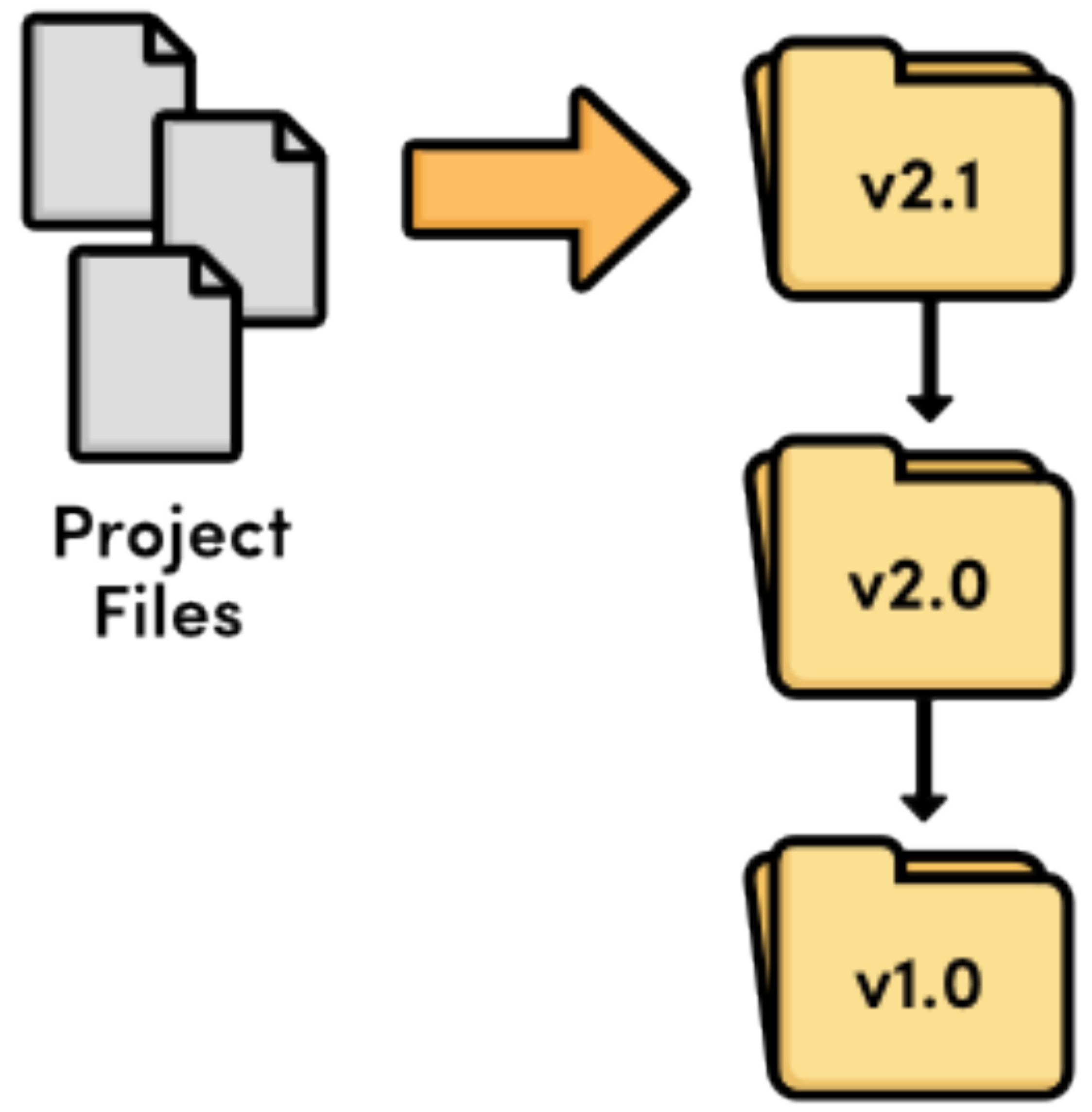
- Have a common repository that can be updated and accessed by an IT team
- All changes are documented and recorded
- Types of files: any text files
  - scripts (startup, monitor, performance, common tasks)
  - crons
  - etc config files

## History of version control

Copy all files into a new folder

Issues:

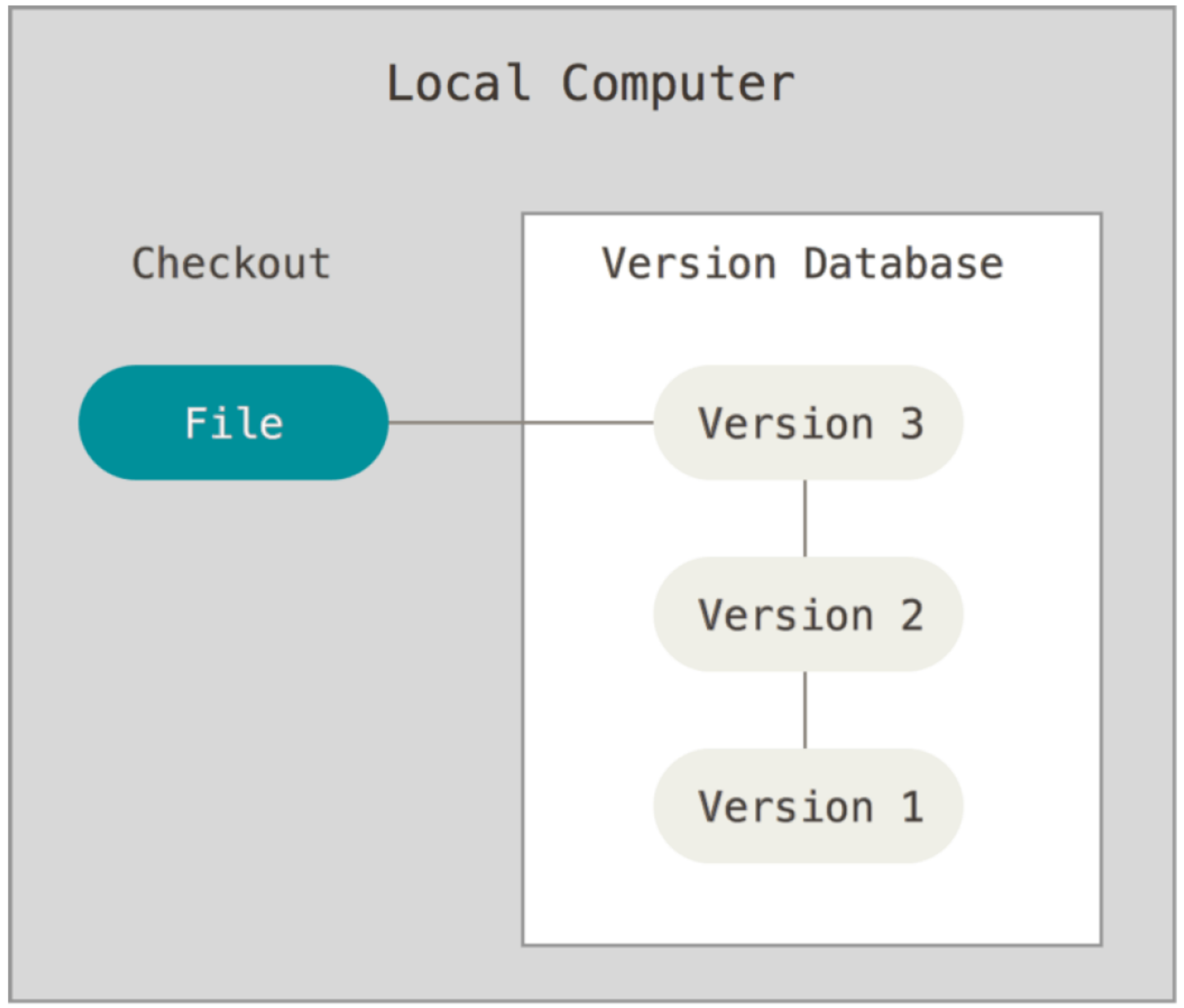
- No idea who was editing what
- Whoever saved last won!
- No way to compare versions

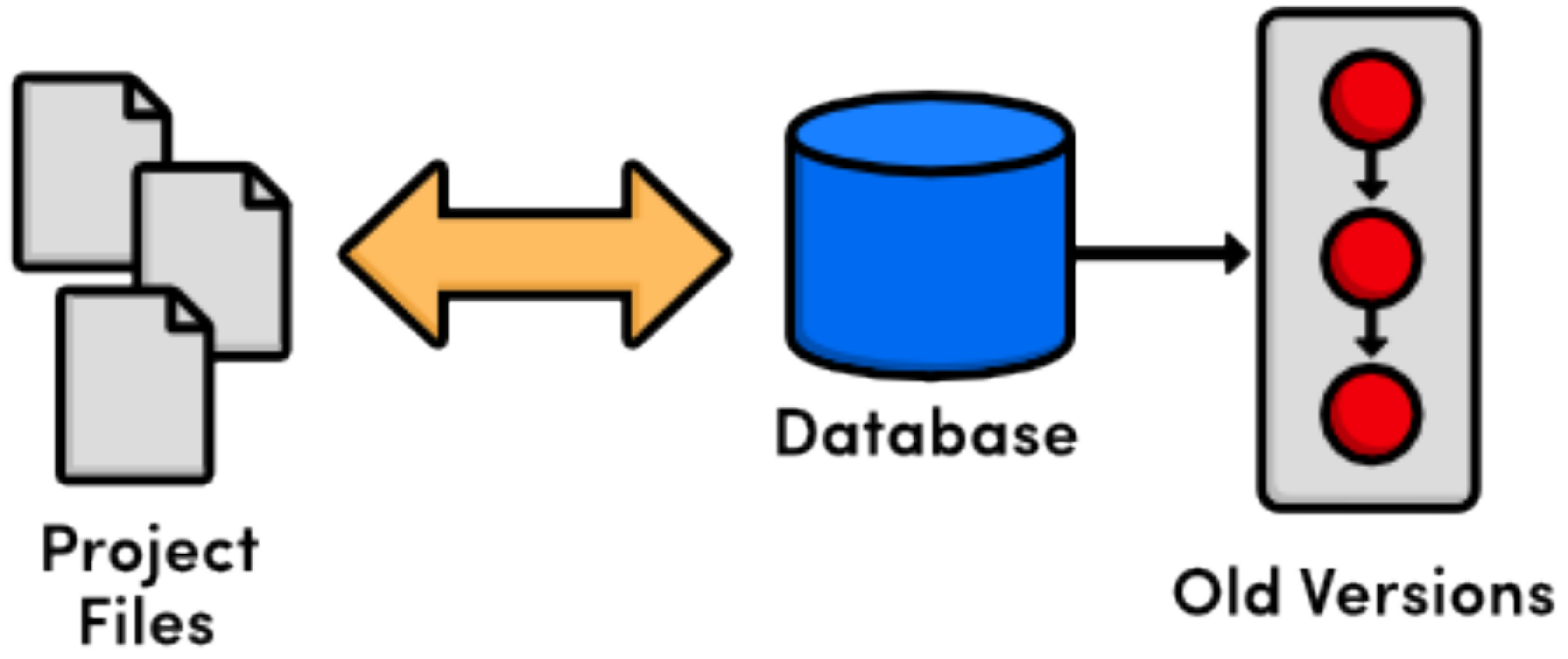


# local version control

**FIGURE 1-1**

*Local version control.*





## Multi-user VCS architecture

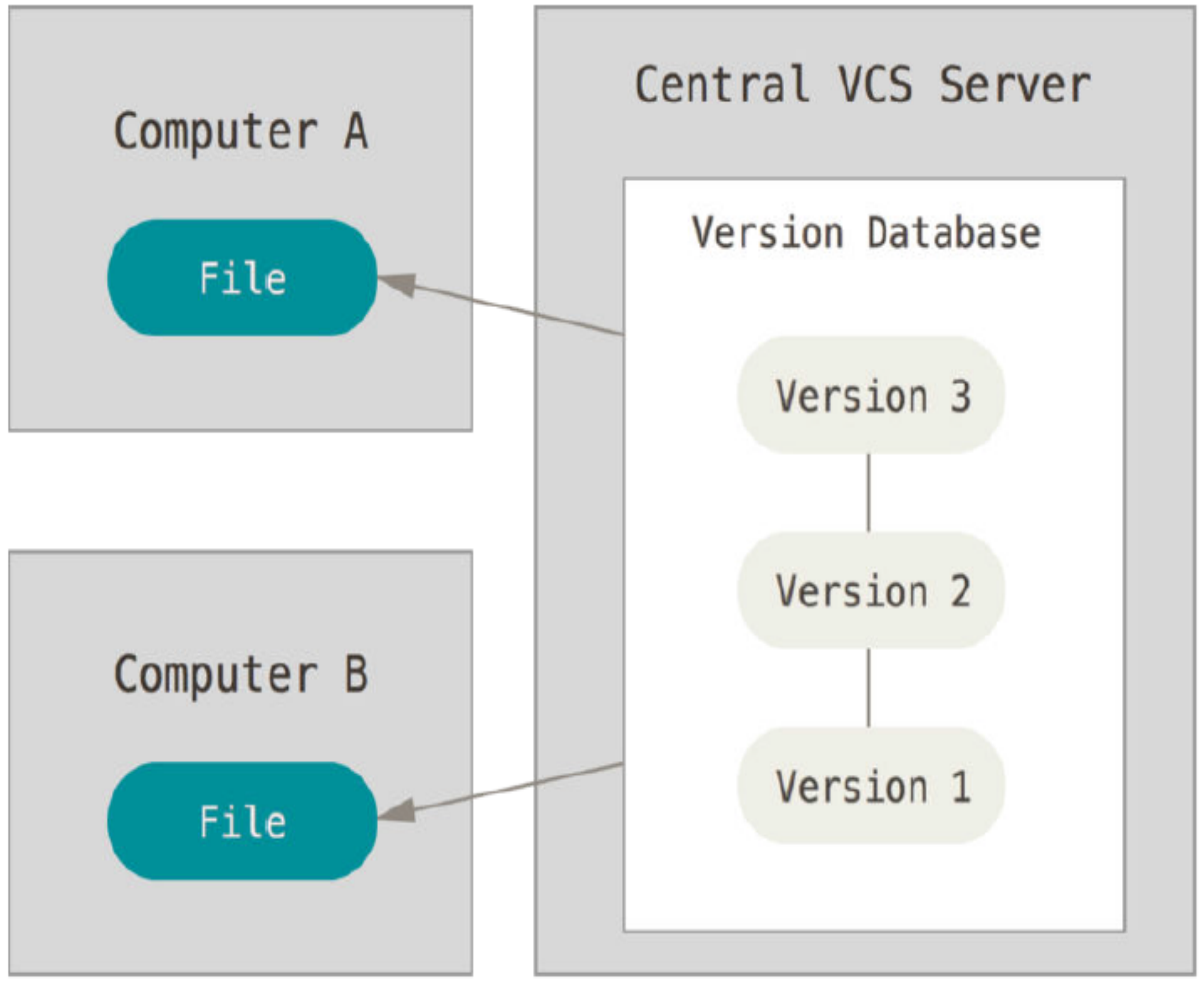
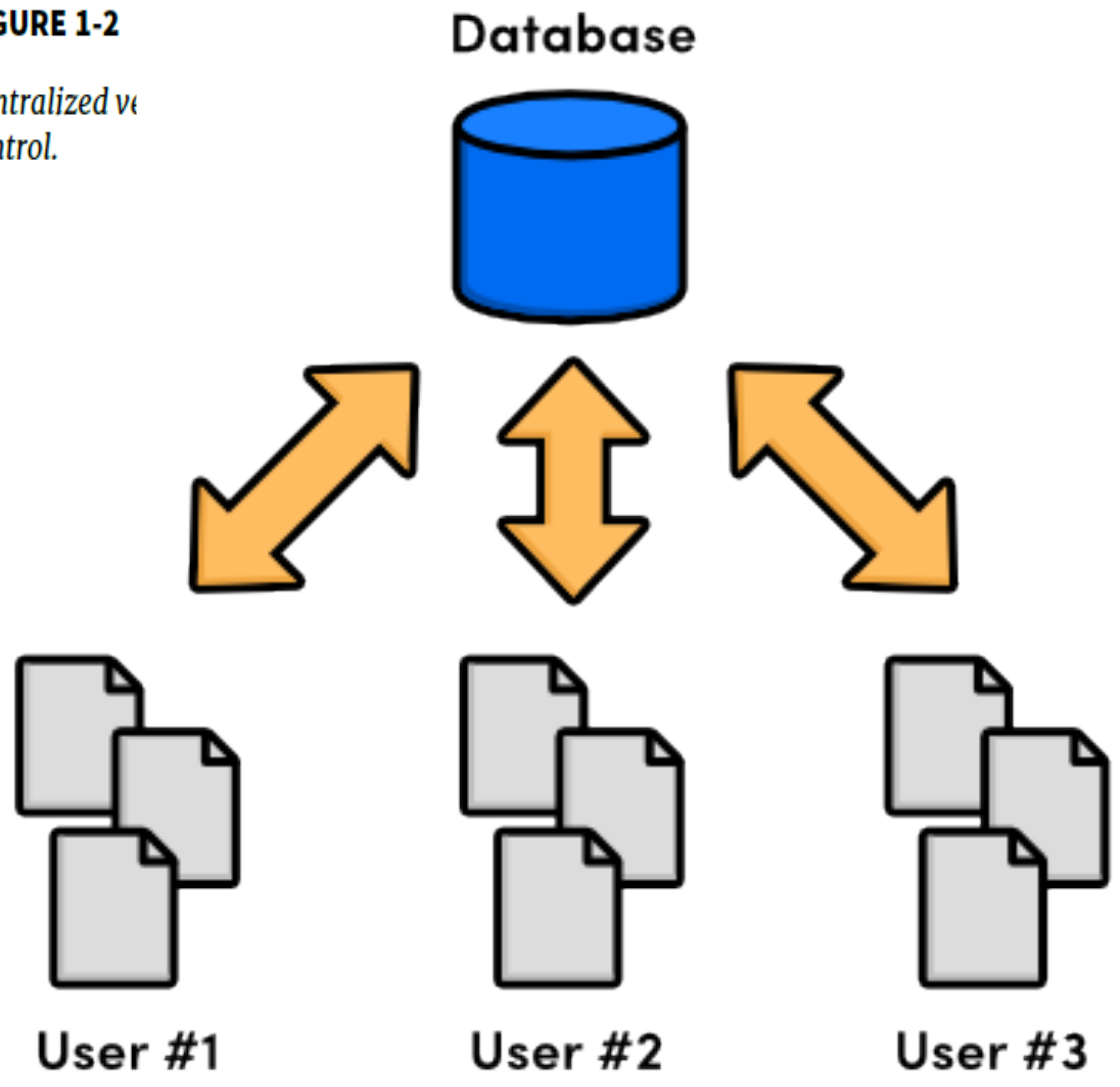


FIGURE 1-2

*Centralized ve*
*control.*

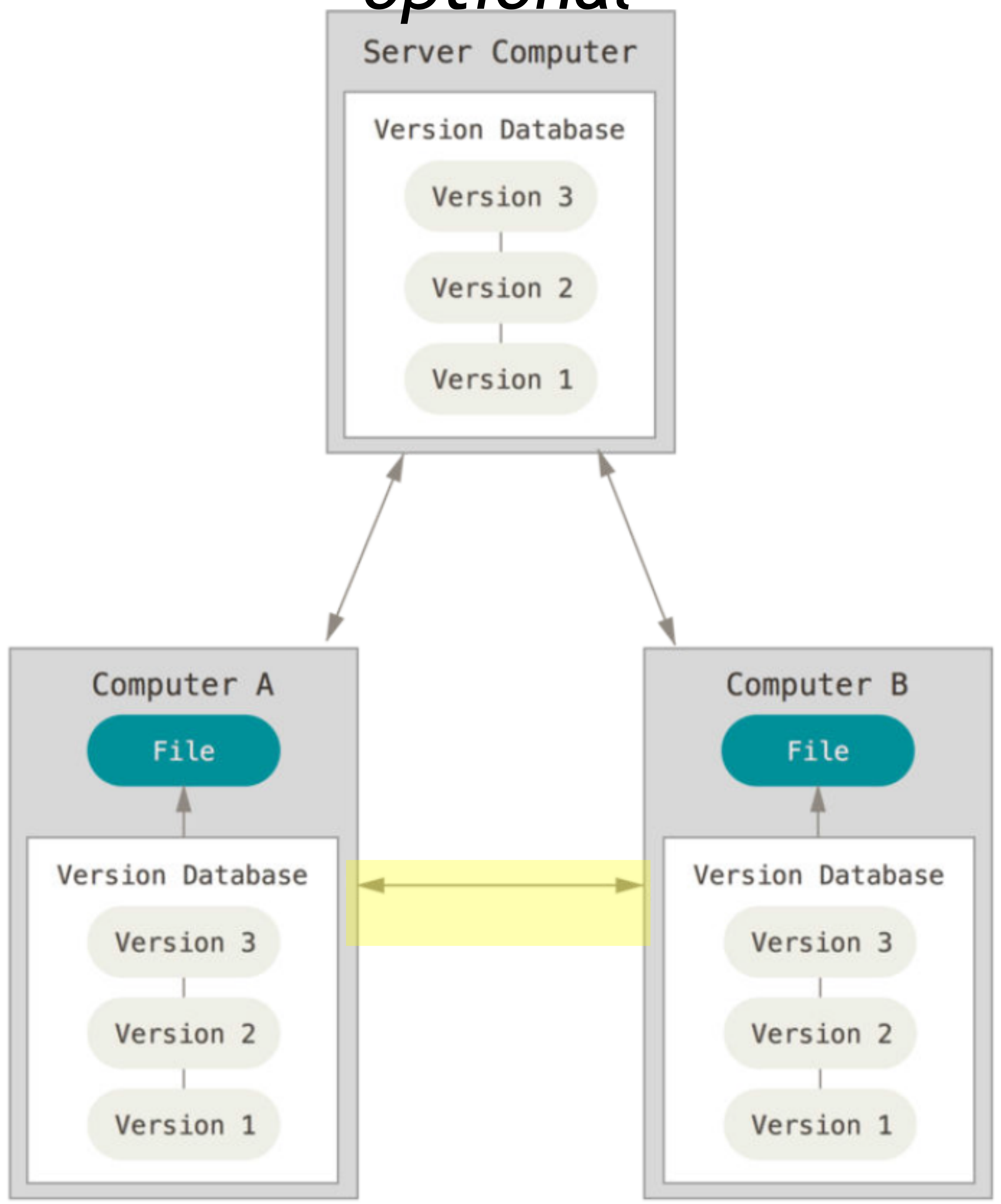


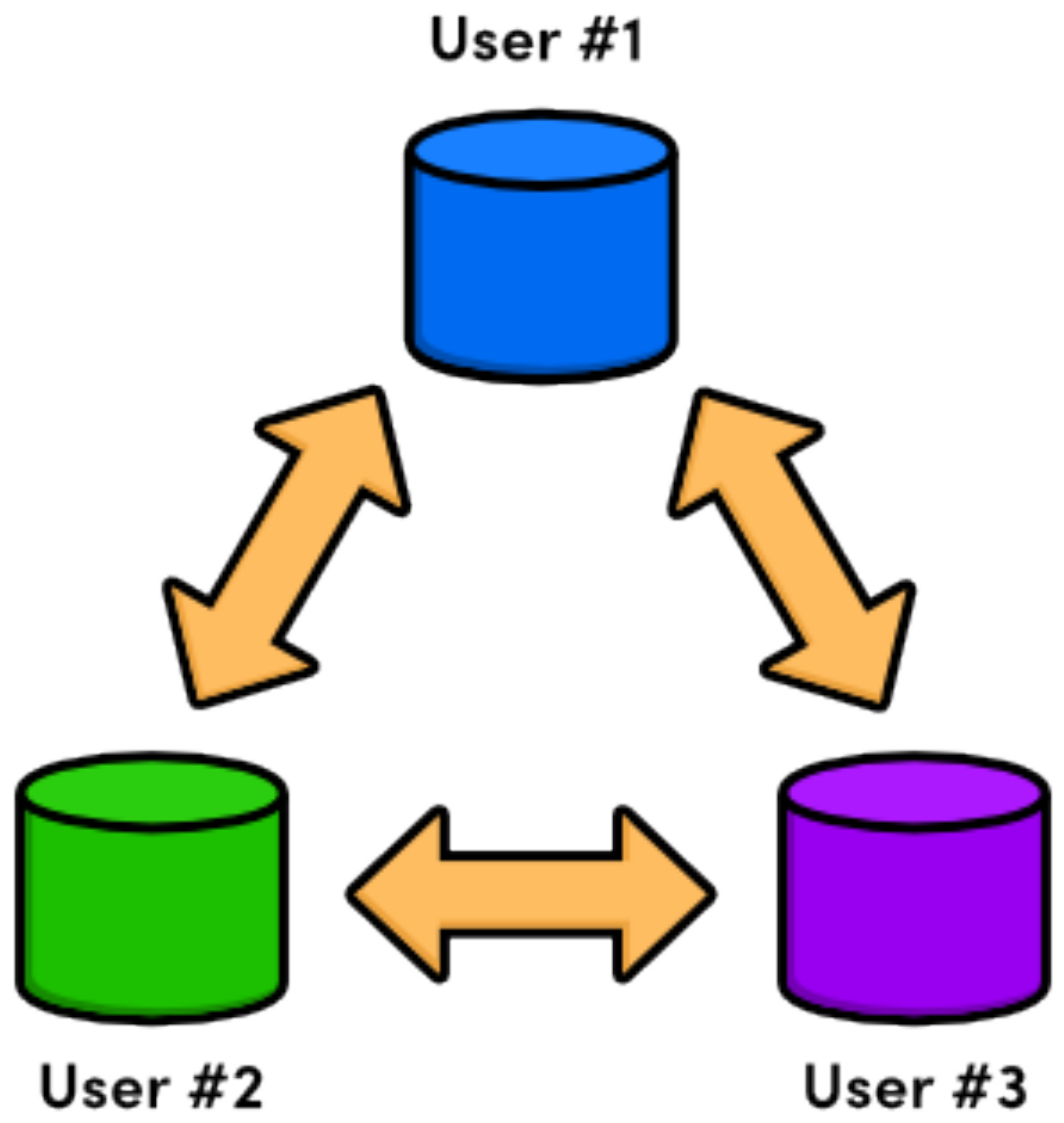
Examples: cvs,  subversion

## Multi-user VCS architecture

T

*optional*

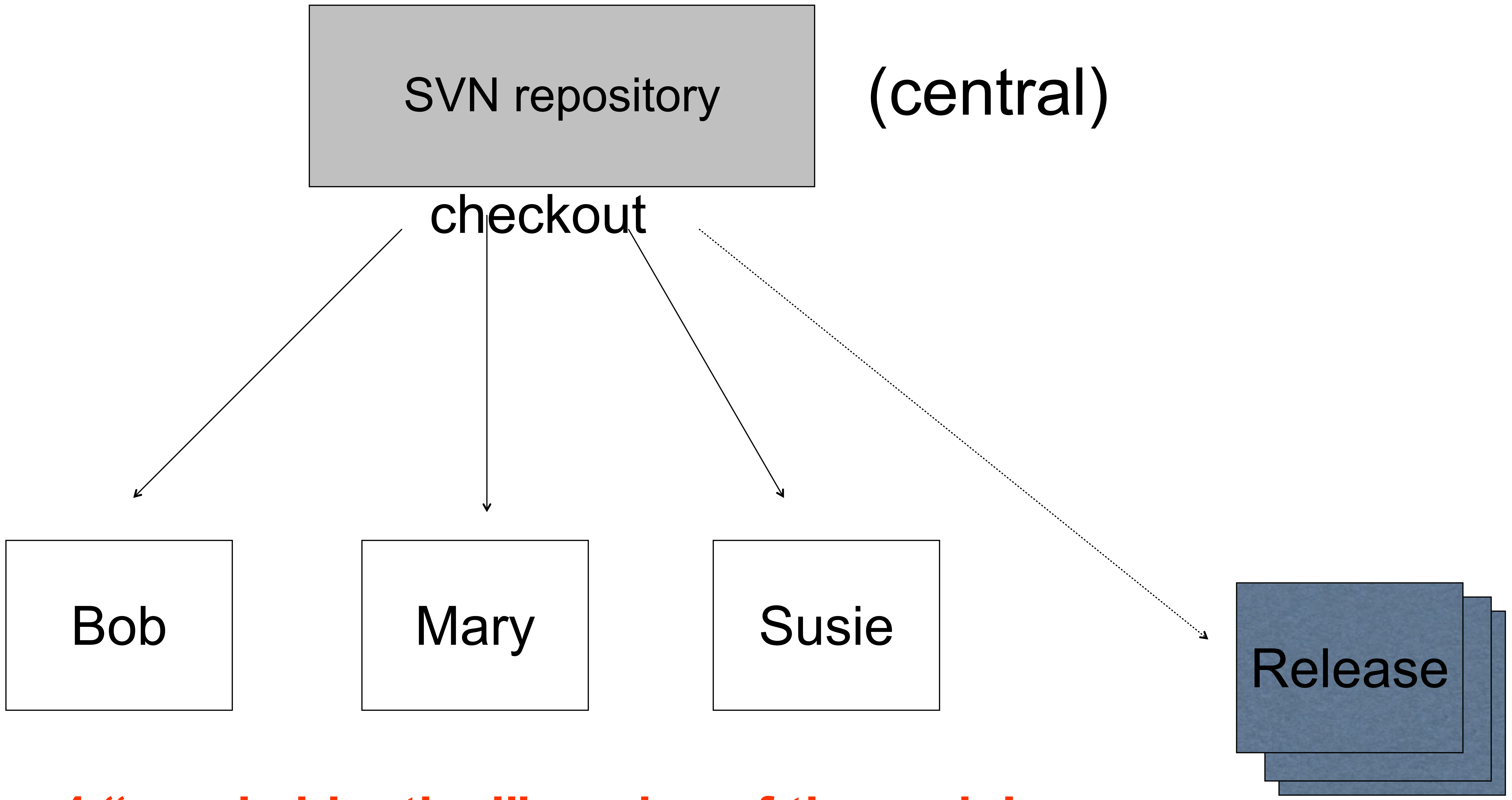




git, mercurial (Hg), bazaar (bzr)
"everyone has the repo"

## SVN overview

In a ‘special’ dir → `/var/svn/repo`    Contains all versions of code: 1.0, 1.1, 1.2



**4 “nearly identical” copies of the code!**

## SVN repository references

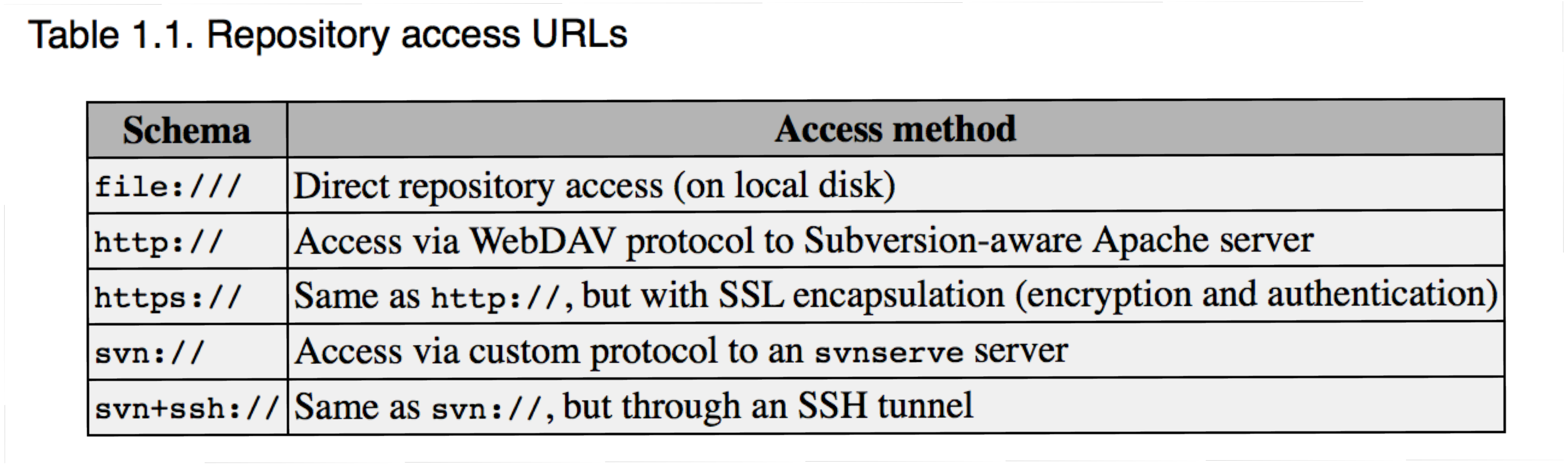


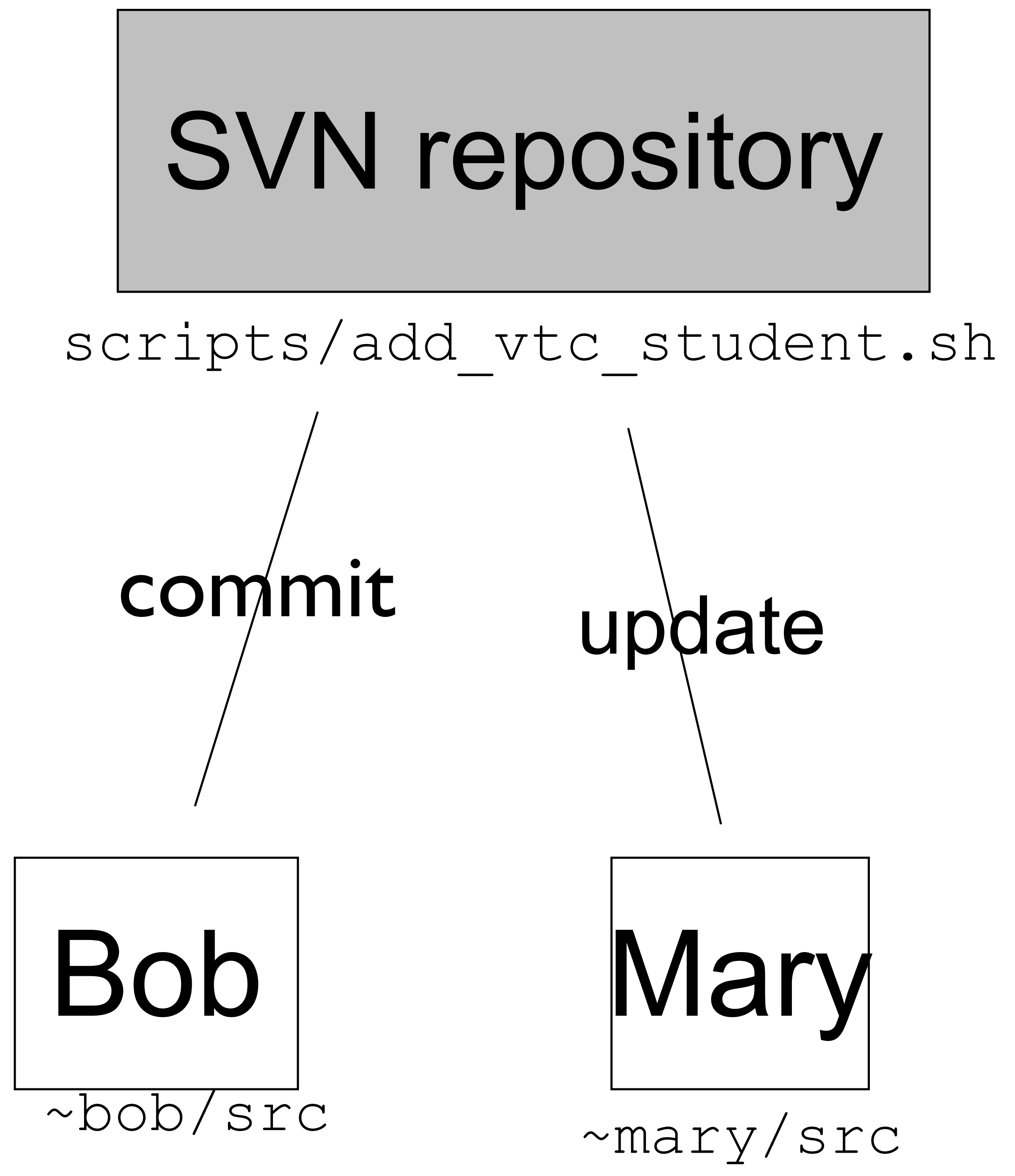
Table 1.1. Repository access URLs

| Schema | Access method |
|---|---|
| `file:///` | Direct repository access (on local disk) |
| `http://` | Access via WebDAV protocol to Subversion-aware Apache server |
| `https://` | Same as `http://`, but with SSL encapsulation (encryption and authentication) |
| `svn://` | Access via custom protocol to an `svnserve` server |
| `svn+ssh://` | Same as `svn://`, but through an SSH tunnel |

Example:

svn checkout svn+ssh://svn.example.com/svn/project

## SVN flow



1. Bob makes a change to `~bob/src/scripts/add_vtc_student.sh`
2. He tests it in his own `src` dir and it works
3. Bob checks into the repository

   ```
   svn commit -m "Bob's cool changes"
   ```

   Shell script goes from v1.3 to v1.4

   Bob and the repository have the changes

4. Mary (other programmers) need to get Bob's new code.

   ```
   svn update
   ```

   Bob's changes are put into her copy of the code

   Any conflicts (if she changed the same line Bob did) are hi-lighted

## Examples

Commit:

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
$
```

Update:

```
$ svn update
Updating '.':
U    button.c
Updated to revision 57.
```

## Examples

The commit will fail if any of the files you update are “out of date” in the repository, meaning someone has changed the file and committed it. You need to do an SVN update to merge their changes into your file and then commit the merged changes.

Update will tell you if the file was:

U - Updated
G - merGed
C - Conflicted

## Conflicts

What if two people edit the same code and make different changes?

- This is called a conflict.

SVN marks the conflict and forces the programmer to 'resolve' it before changes can be checked in.

Example:

- Bob changes line 5 and checks in first.
- Susie also changes line 5 and does an update
- SVN tells Susie that there is a conflict: "C"
- Susie edits the files
  - conflicts are marked with "<<<", "===",">>>"
- Susie makes changes
- Then she updates and checks in
- Bob updates and gets Susie's changes

## SVN summary

Purpose:

- track code changes (revisions)
- allow multiple programmers
- merge changes from multiple sources
- handle conflicts
- be able to pull any past revision

Key commands:

- `svn checkout` - check out a SVN project to my local directory
- `svn update` - make my local version like the repository
- `svn commit` - check in my local changes to the repository

# SVN central repository init

The import is the trickiest part.

1. Setup the directories:

```
$ sudo mkdir /home/svn
$ sudo mkdir /home/svn/myproject
```

2. Create the repository

```
$ svnadmin create /home/svn/myproject
```

3. setup file permissions

```
$ cd /home/svn
$ sudo chown -R www-data:subversion myproject
$ sudo chmod -R g+rws myproject
```

## Distributed VCS → git

Why? A Centralized VCS broke down.

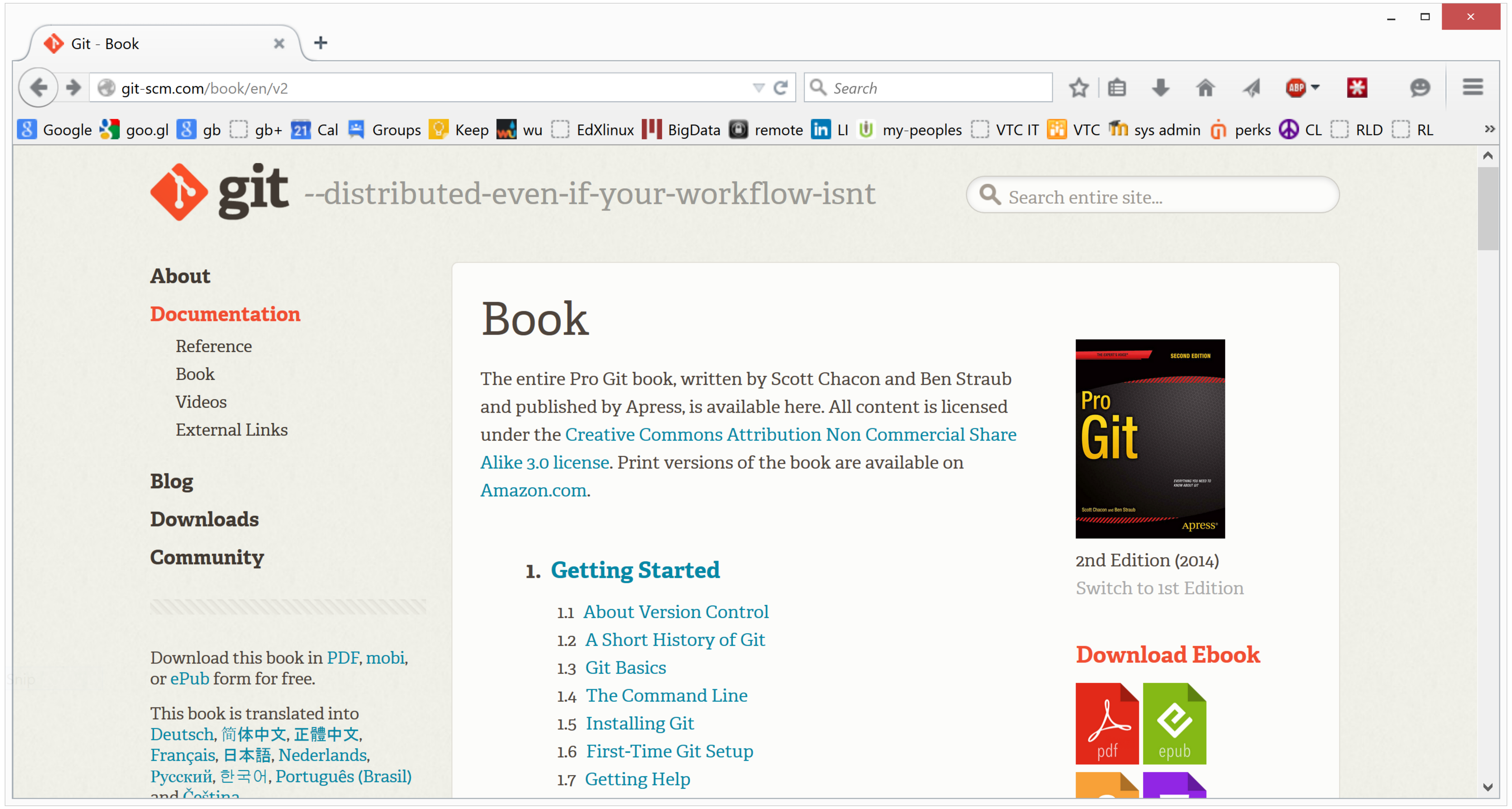
*Every* working dir has a copy of the repository

Four file 'states'

1. Untracked → a new file which git is not even keeping track of
2. Modified → a tracked file that has been changed
3. Staged → a modified file ready to go into the next commit snapshot
4. Committed → stored in local repo, ready to be pushed out to everyone

# Free online book #1

https://git-scm.com/book/en/v2



Git - Book
git-scm.com/book/en/v2
Search
Google goo.gl gb gb+ Cal Groups Keep wu EdXlinux BigData remote LI my-peoples VTC IT VTC sys admin perks CL RLD RL
git --distributed-even-if-your-workflow-isnt
Search entire site...
About
Documentation
Reference
Book
Videos
External Links
Blog
Downloads
Community
Download Ebook
pdf
epub

Book

The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. Print versions of the book are available on Amazon.com.

2nd Edition (2014)
Switch to 1st Edition

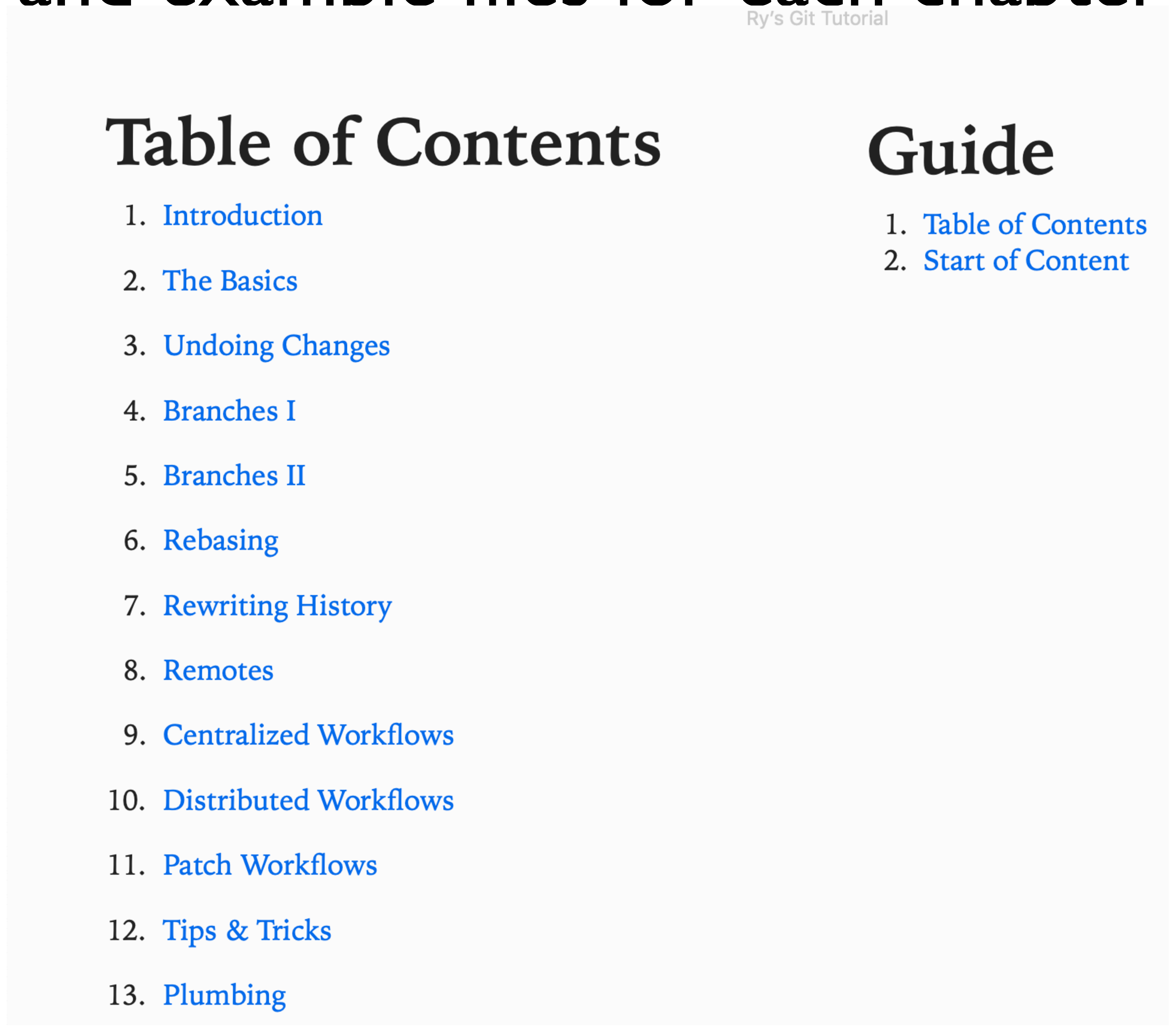
1. Getting Started
   - 1.1 About Version Control
   - 1.2 A Short History of Git
   - 1.3 Git Basics
   - 1.4 The Command Line
   - 1.5 Installing Git
   - 1.6 First-Time Git Setup
   - 1.7 Getting Help

Download this book in PDF, mobi, or ePub form for free.

This book is translated into Deutsch, 简体中文, 正體中文, Français, 日本語, Nederlands, Русский, 한국어, Português (Brasil) and Čeština

# Free online book #2

Ry's Git Tutorial - this page contains links to the epub file and example files for each chapter



Ry's Git Tutorial

## Table of Contents

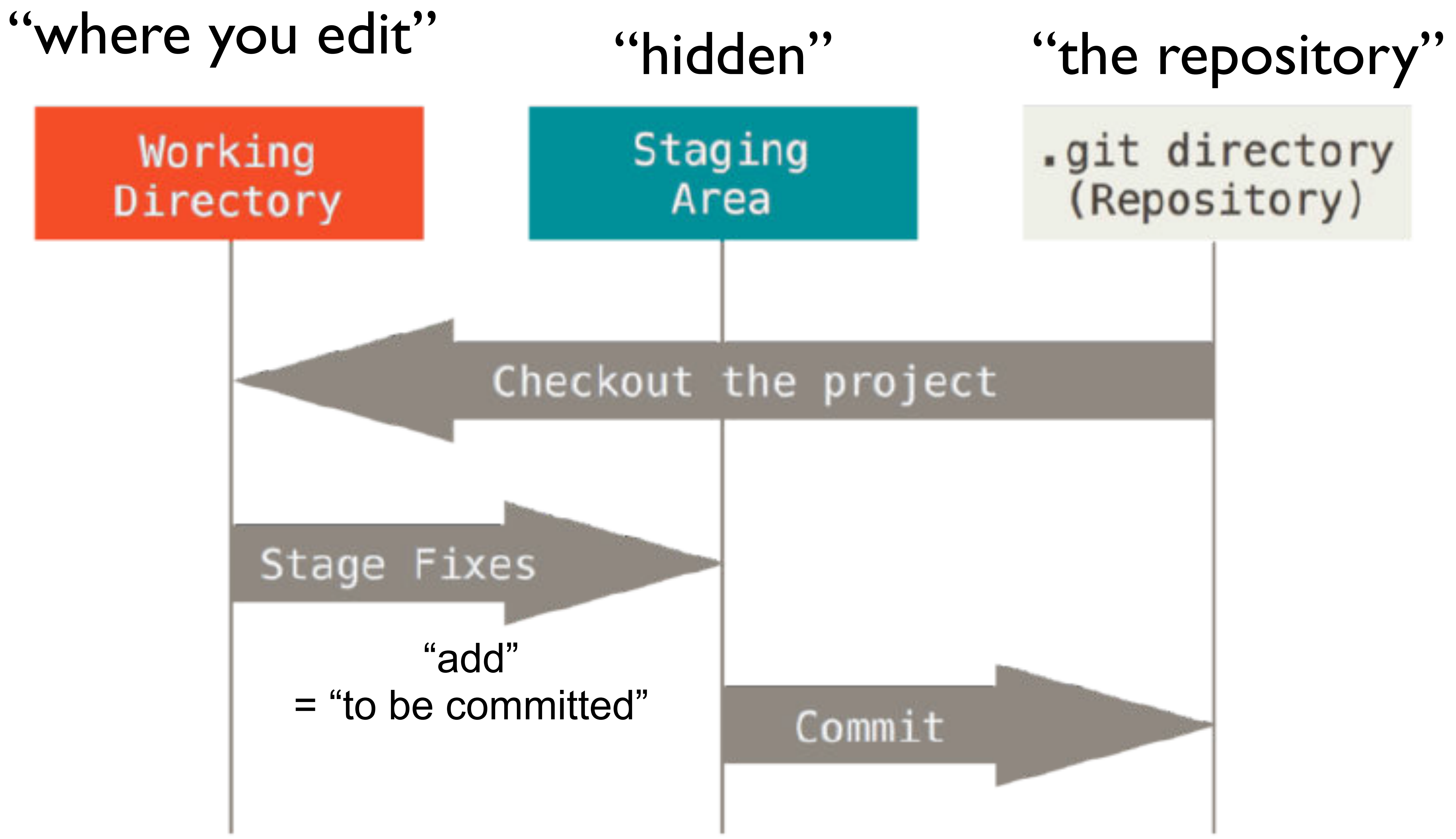
## Guide

## Branching tutorial

Or, try this interactive tutorial on branching:

[https://learngitbranching.js.org](https://learngitbranching.js.org)

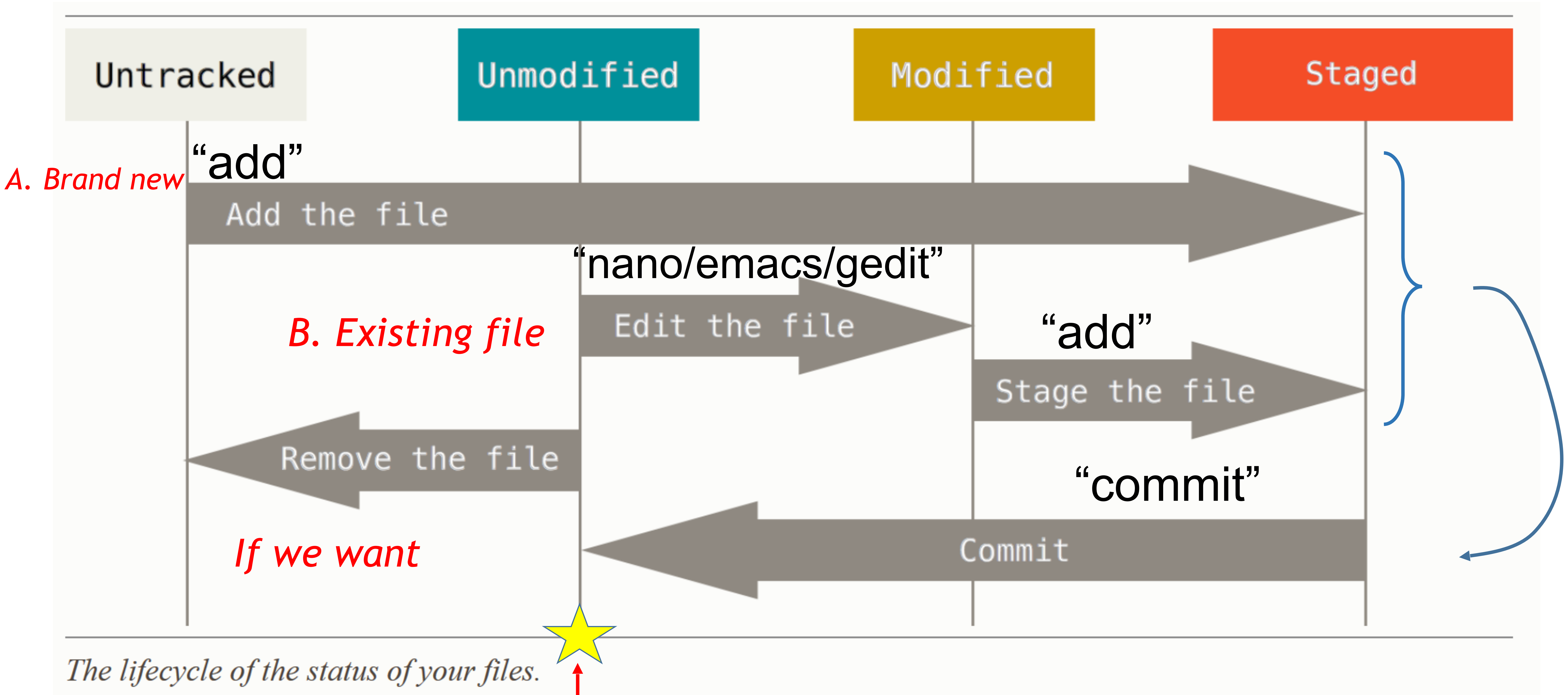
# git local workflow



Typical:

- Edit several files, "adding" them as you edit them.
- When they are <u>all</u> ready, `commit` to make a new 'snapshot'

# *File* states within git

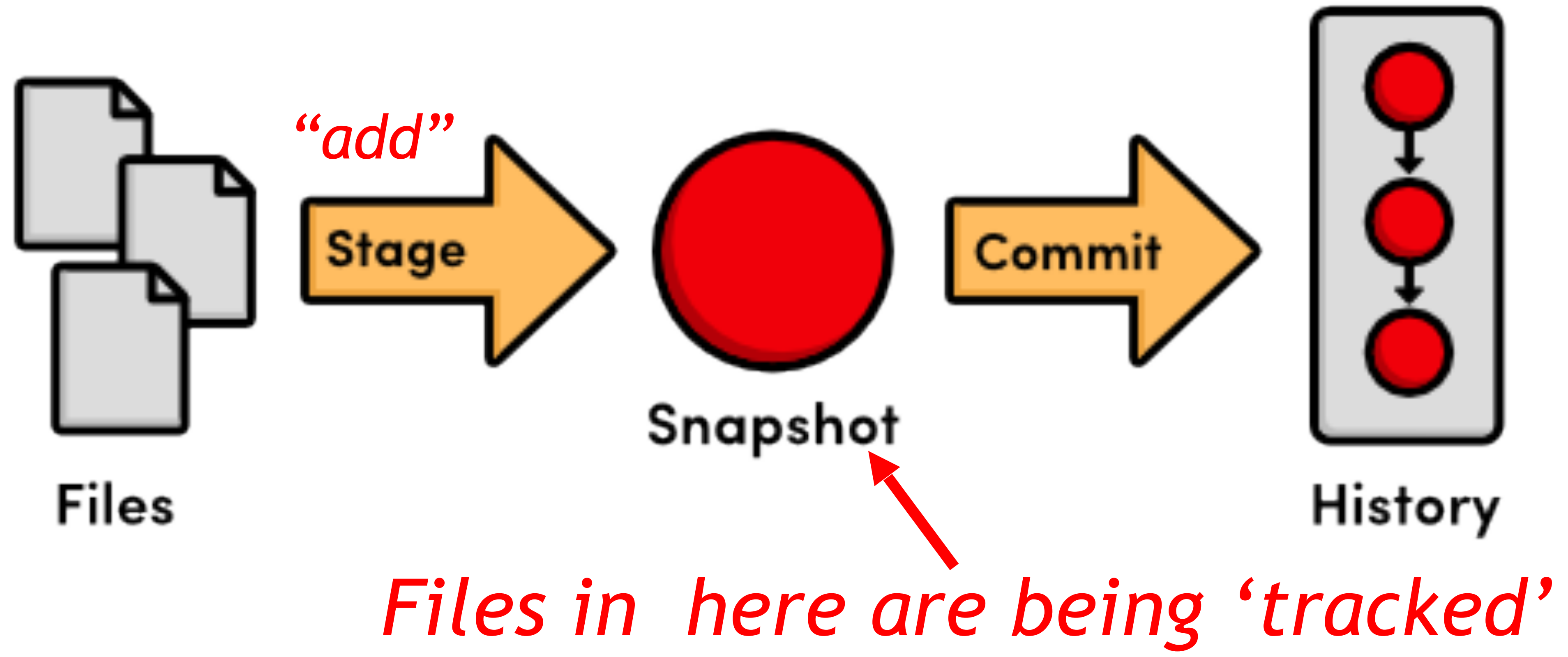


The lifecycle of the status of your files.

*"At rest" - "steady state"*

# Git term review

**Init** – start a new repository (.git)
**Stage** – tell git which files to include in the next snapshot
**Add** – tell git that this file (new or existing) is to be included in staging for the next snapshot (commit)
**Commit** – record a new snapshot from all the files in staging
**Track** – a file that is in staging is said to be 'tracked'



## User setup

Git requires some 'global info' before you can go any further adding files to your new repo (`~/.gitconfig`)

```
$ git config --global user.email “ldamon@vtc.edu"
$ git config --global user.name  “Leslie Damon"
```

Alternatively, you can also tell Git your name and email address using the GIT_AUTHOR_NAME and GIT_AUTHOR_EMAIL environment variables.

Check your git settings with:

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

## Create a new git repo

cd into the desired directory of the new repo

```
$ git init
```

Verify that a new `.git` directory exists

The current directory "is" your local, working instance of the repo

The `.git` directory is your distributed copy of the full repo

**Note**: if you don't want a dir under git version control → just delete `.git`. *That's it.*

```
$ mkdir myGitRepository
$ cd myGitRepository/
$ git init
Initialized empty Git repository in /home/
ldamon/myGitRepository/.git/
$ ls -a
.   ..   .git
```

## Git project status

View the status in git repo

```
$ git status
```

Why is index.html “untracked”?

```
$ vi index.html
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
   index.html

Nothing added to commit, but untracked files present (use "git add" to track
```

## Staging a snapshot

“staging a snapshot” means adding/editing/removing files to the project snapshot.

- New files in the dir are called 'untracked'.
- Edited files are called ‘modified’

To include *new* files (or edited ones) into this next snapshot is called 'tracking' them.

```
$ git add index.html
```

Now they are 'staged‘ (*i.e.* “to be committed”)

## Staging a snapshot

```
$ git add index.html
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
	new file:   index.html
```

## Checking the status

`$ git status -s` is the short version

A – added

N – new

M - modified

```
$ git status -s
A  index.html
```

## Commit snapshot to history

commit makes 'snapshot' of the staged files in the project

  ensure that they are saved

Files are moved from 'staged' to 'unmodified' status

Note:  If there are multiple staged staged files all are committed

Comment required.  -m is easiest method:

```
$ git commit -m “new comment”
```

# Commit snapshot to history

```
$ git status -s
A  index.html
$ git commit -m "add index.html"

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got ‘ldamon@ubuntults.(none)')\

ldamon@ubuntults:~/myGitRepository$ git commit -m "add index.html"
[master (root-commit) 20965e4] add index.html
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

## View history

Check the log of revisions

```
$ git log
```

```
$ git log
commit 81653aa6f06b56fe2fde9d14ded93aba0c763fac (HEAD -> master)
Author: Leslie Damon <ldamon@vtc.edu>
Date:   Mon Apr 26 18:49:40 2021 -0400

    fixed typo in index.html

commit 20965e447c8bfe90913e721b5083c1d9acdd0989
Author: Leslie Damon <ldamon@vtc.edu>
Date:   Mon Apr 26 18:42:05 2021 -0400

    add index.html
```

## View history

### Short version:

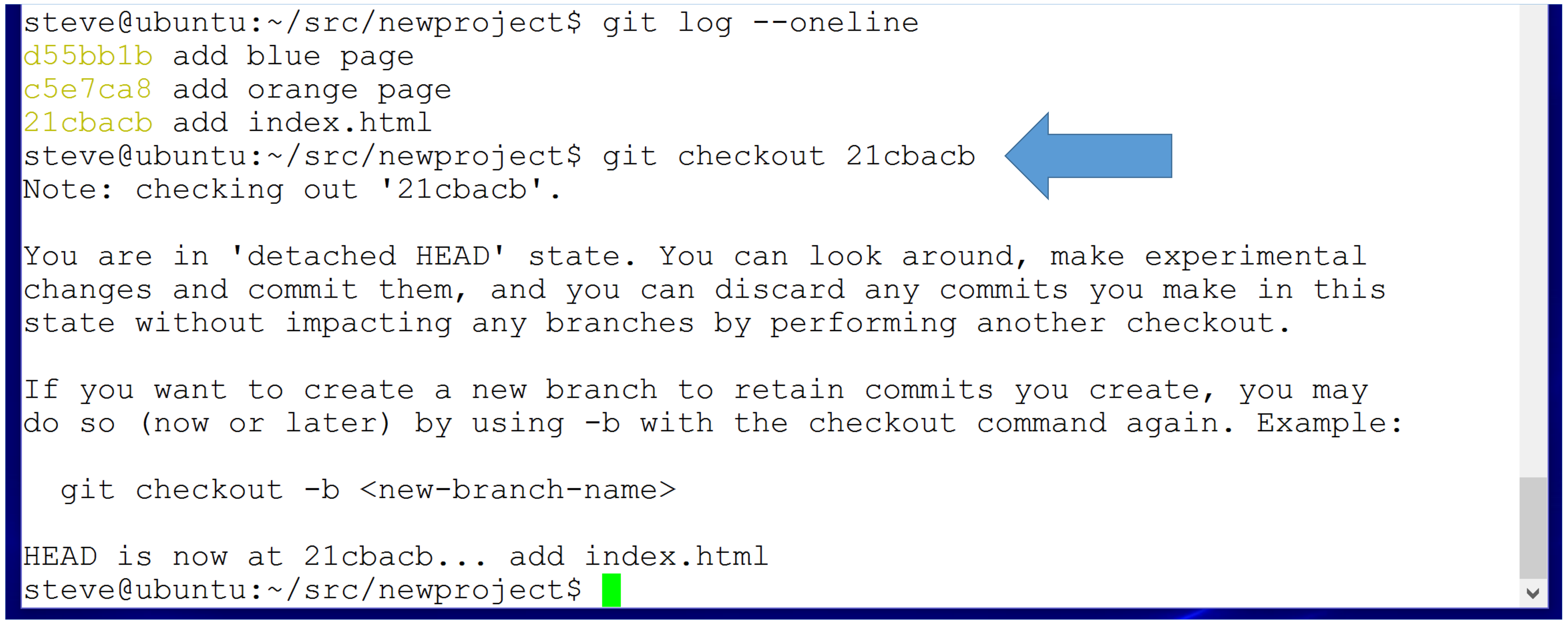
```
$ git log --oneline
```

```
$ git log --oneline
81653aa (HEAD -> master) fixed typo in index.html
20965e4 add index.html
```

# Going back in time

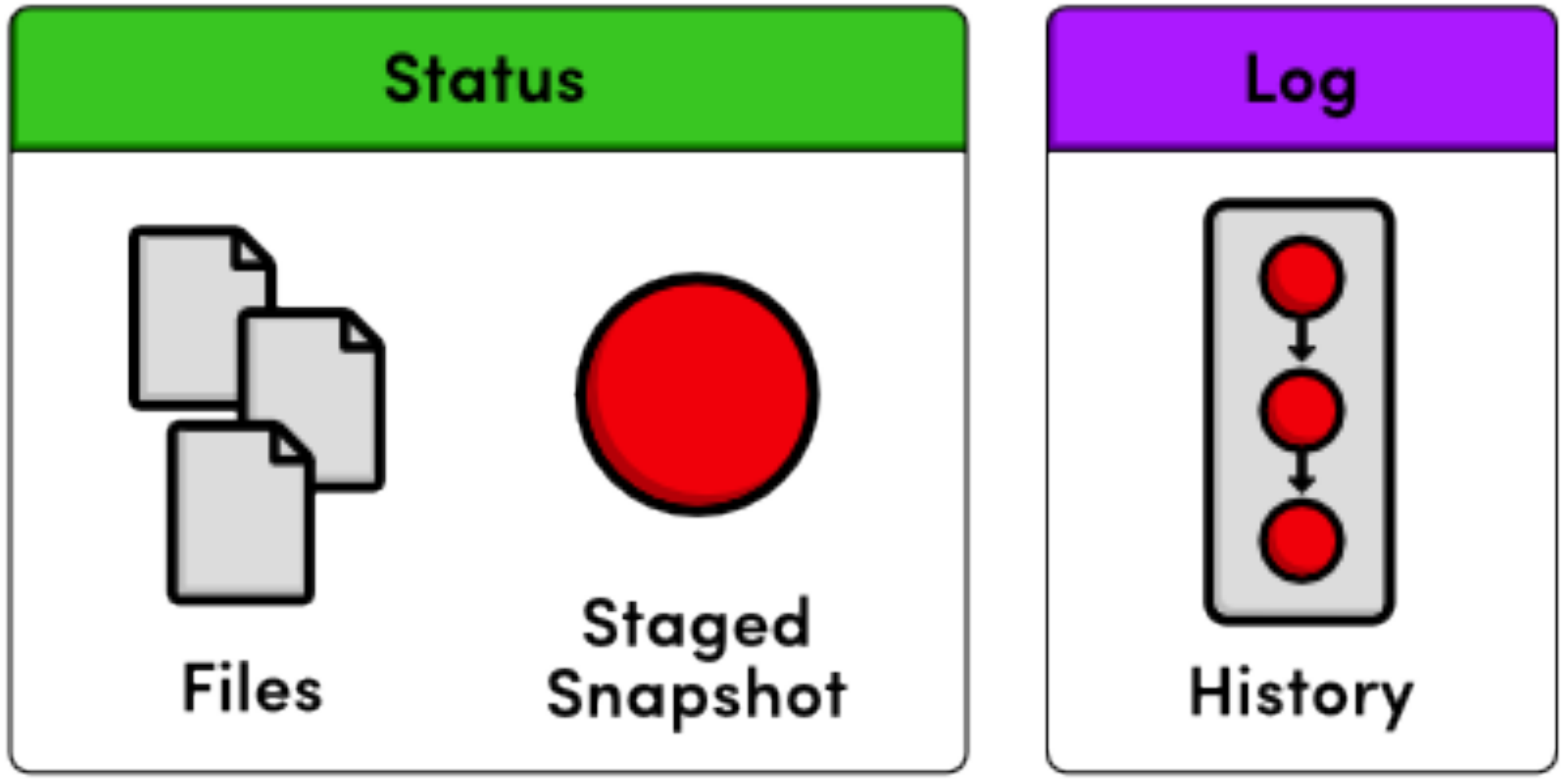
Use `$ git checkout <hash>` to change the working dir to a <u>previous</u> *snapshot*

Use `$ git checkout master` to *reset* to most recent version.



```
steve@ubuntu:~/src/newproject$ git log --oneline
d55bb1b add blue page
c5e7ca8 add orange page
21cbacb add index.html
steve@ubuntu:~/src/newproject$ git checkout 21cbacb
Note: checking out '21cbacb'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 21cbacb... add index.html
steve@ubuntu:~/src/newproject$
```

# Status vs log



“current”

“past”

## Remote repositories

One major point is to share repositories with remote submitters

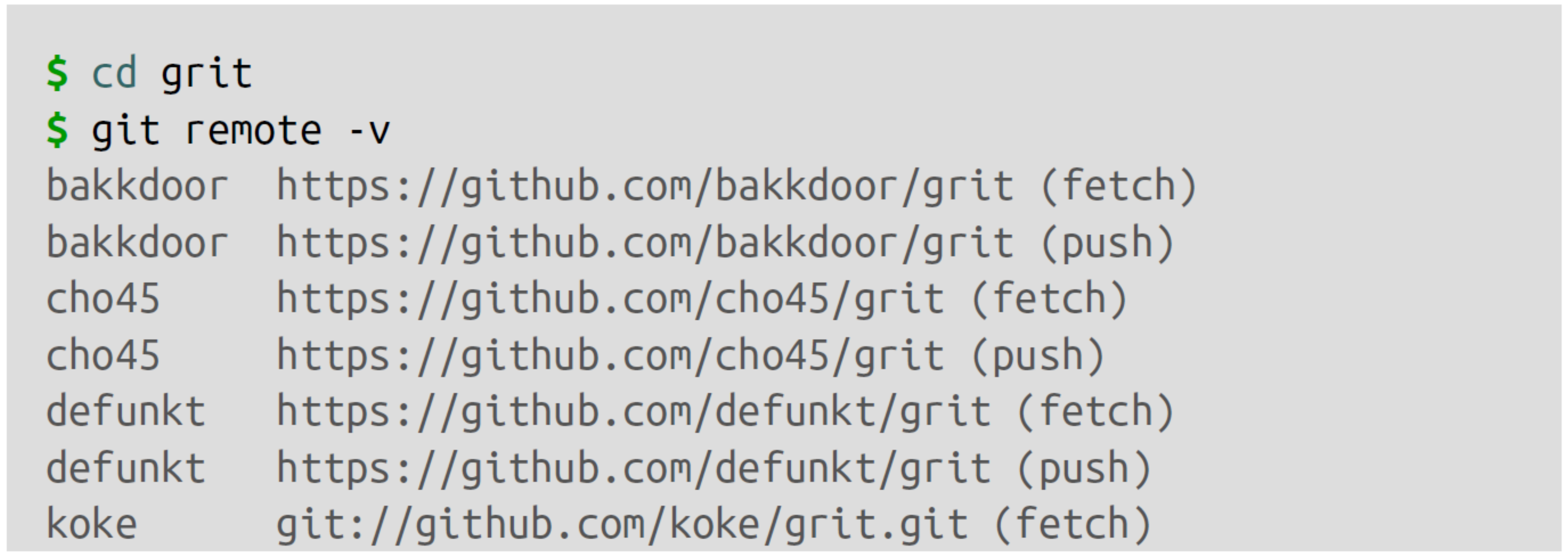
Check for remote repositories with:

```
$ git remote
```

Could be multiple remotes

Point to a remote with the `$ git clone` command:

```
$ git clone https://github.com/schacon/ticgit
```



```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
```

## Remote git server

Rather than point to many different repos, a git project is often used with a "central" remote server.

The multiple users all clone from that central repository.

Four transfer protocols:

- local ("/")
- ssh
- http
- git

To start a *new* git project:

```
$ git clone /opt/git/project.git
```

To point an *existing* git project to a remote:

```
$ git remote add local_proj /opt/git/project.git
```

## Remote repositories sync

When ready to make local changes available to everyone, the coder "pushes" out the committed changes.

```
$ git push [remote] [branch]
$ git push origin master
```

To get the changes of others to your local version, you "pull"

```
$ git pull
```

## example

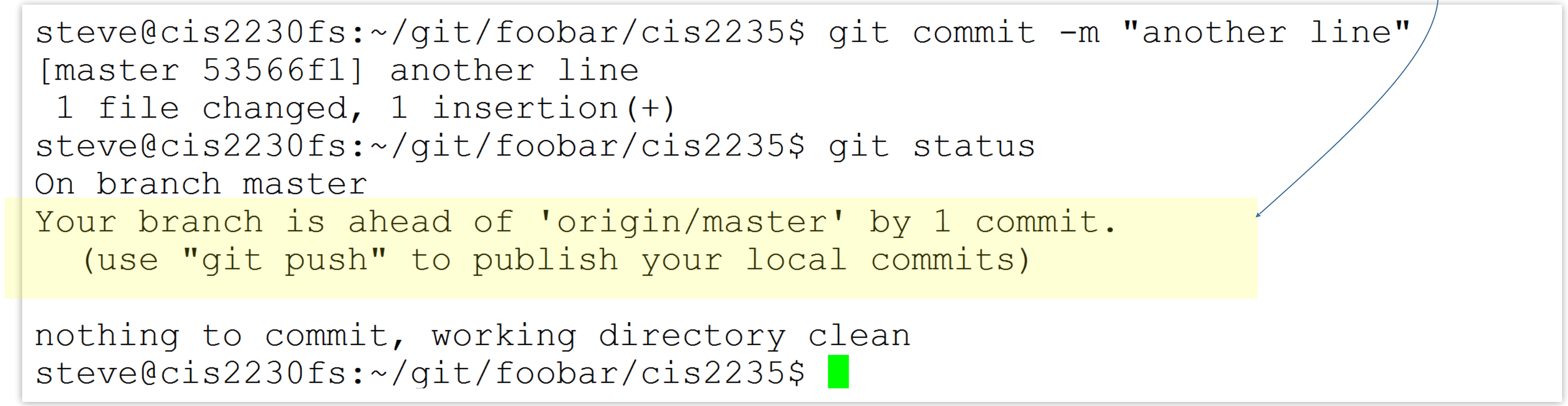
project cloned from a remote git repo

made changes to a file

added and committed

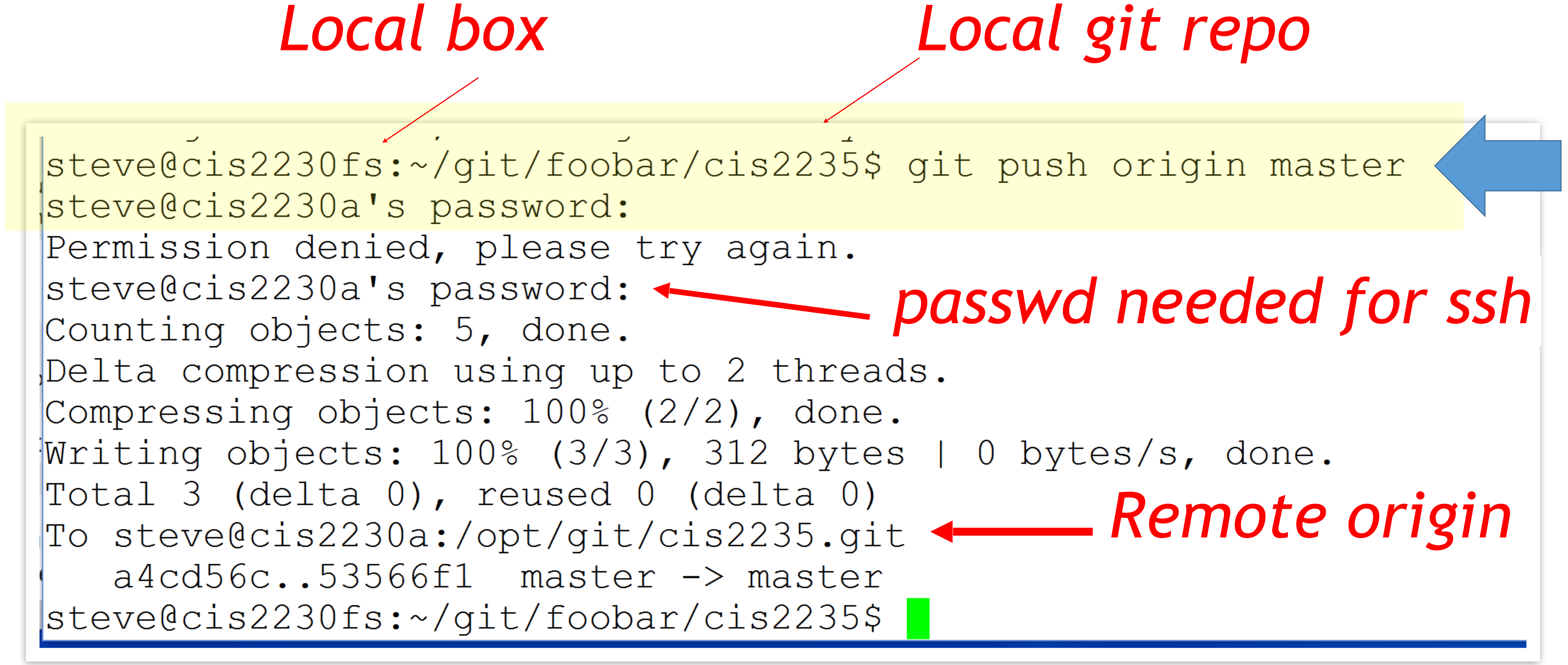
Notice the message...

Next: `$ git push origin master`



```
steve@cis2230fs:~/git/foobar/cis2235$ git commit -m "another line"
[master 53566f1] another line
 1 file changed, 1 insertion(+)
steve@cis2230fs:~/git/foobar/cis2235$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
steve@cis2230fs:~/git/foobar/cis2235$ 
```

# example



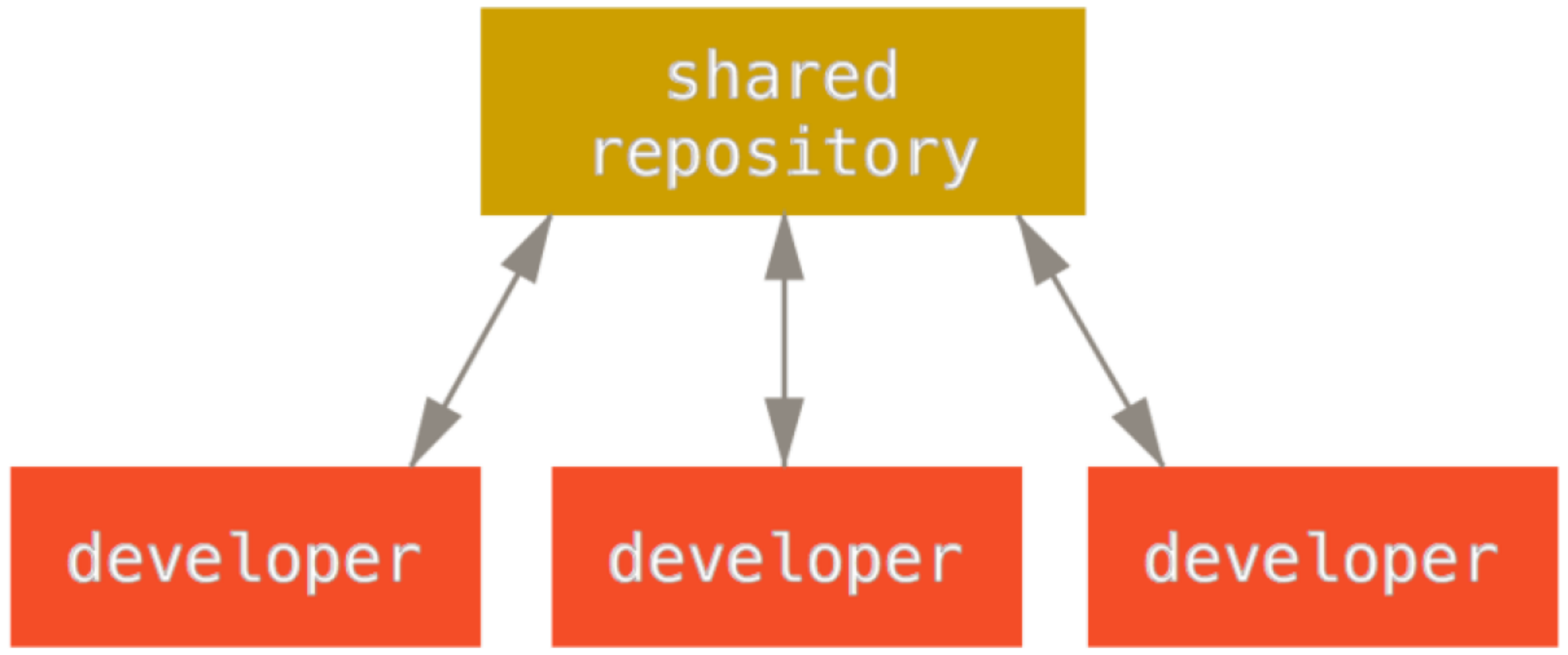
```
steve@cis2230fs:~/git/foobar/cis2235$ git push origin master
steve@cis2230a's password:
Permission denied, please try again.
steve@cis2230a's password:
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 312 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To steve@cis2230a:/opt/git/cis2235.git
   a4cd56c..53566f1  master -> master
steve@cis2230fs:~/git/foobar/cis2235$
```

# Distributed git repository

CHAPTER 5: Distributed Git

**FIGURE 5-1**

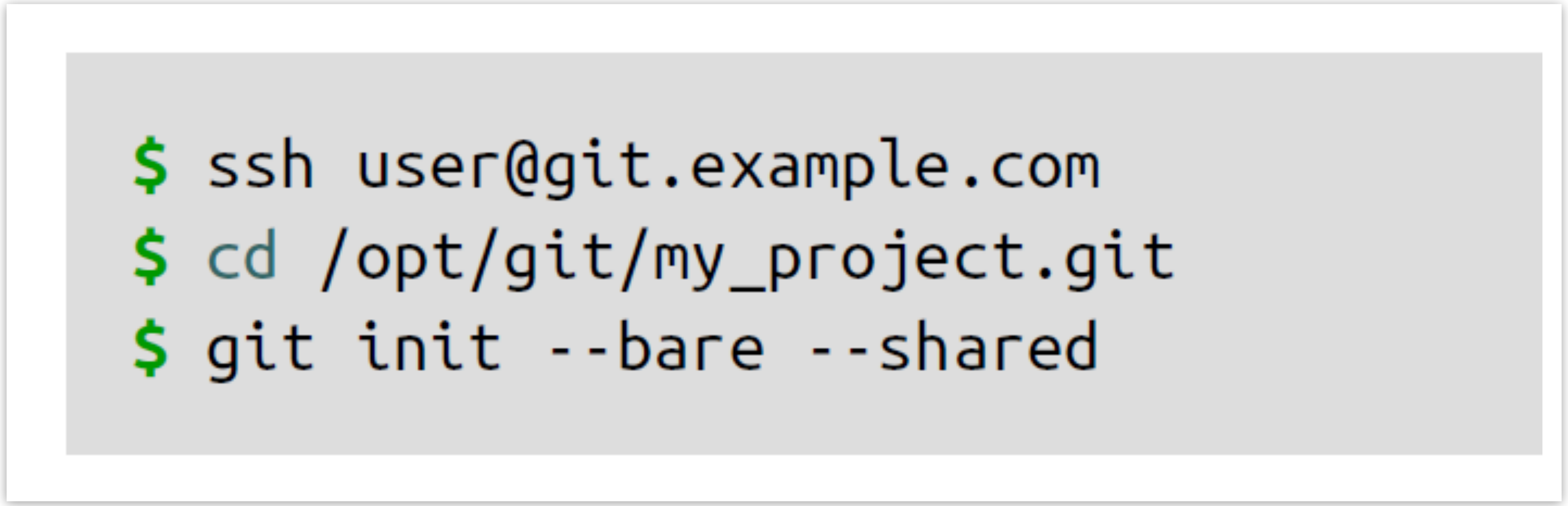
*Centralized workflow.*



## Creating remote repository

A central git repository is one without a ‘working’ directory.
Want to create a bare repository

Users ‘clone’ to it just like before.



```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

# Remote repository setup

If you have an *existing* project:

```
$ git clone --bare myrepo /any/other/dir/project.git
```

Now, copy that project.git dir where you want it

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

Cool. Now what do "others" do to *use* your repo?

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

## Git export

The files are called “indexes”

```
$ git checkout-index -a -f --prefix=<dir>
```

# git command review and workflow

Lab 10

Create remote repo:

- create remote repo dir: check perms and owners
- init the remote git repo with

```
$ git init --bare
```

Have a 1st user populate the repo with some files

- create a working git repo with

```
$ git clone <remote_dir>
```

- add in files

```
$ git add <files>
```

- commit files from stage to local repo

```
$ git commit -m "comment"
```

- push from local to global

```
$ git push
```

# git command review and workflow

Lab 10

Have a 2nd user edit files

- create a working git repo with

```
$ git clone <remote_dir>
```

- edit a file:

```
$ nano <file>
```

- stage:

```
$ git add <file>
```

- commit:

```
$ git commit -m "comment"
```

- push:

```
$ git push master origin
```

## git command review and workflow

Lab 10

Go back to the first user and get changes

```
$ git pull
```

verify the new changes from the 2nd user are now in the 1st users dir