PHY604 Lecture 2

August 26, 2021

Note:

- Lecture 1 slides, readings, and example programs posted on website:

https://you.stonybrook.edu/cdreyer/phy604_fall2021/

redirects to:

https://dreyer-research-group.github.io/phy604_fall2021.html

Review: Real/Floating point numbers are more complicated

- Infinite real numbers on the number line need to be represented by a finite number of bits
- Finite memory results in limited **size and precision** of floating point numbers
  - Not all real numbers (even simple ones) can be stored in a finite number of digits in a base-2 representation
  - Example: $1/10=0.1_{10}$ = $0.0001100110011..._2$ does not have a finite representation in base 2 just as $1/3=0.333333..._{10}$ has no finite representation in base 10
- This means that even simple floating point numbers are often approximated with some small error
  - This means that floating point arithmetic is not exact! (on all computers and programming languages)
- Errors can compound if not treated carefully!

Review: Roundoff error: Another example

- Consider computing exp(-24) via a truncated Taylor series:

$$e^x \simeq S(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

- Error in the approximation (**i.e., truncation error**) is less than:

$$\frac{|x|^{n+1}}{(n+1)!} \max\{1, e^x\}$$

- But if we compute S(-24) by adding terms until they are less than machine precision (8 byte):
  - S(-24)=3.7814382919759864E-007
  - Exp(-24)=3.775134544279097 7E-011
  - Error is larger than the result (much larger than truncation error)!!
  - Looking at terms, we see we are relying on cancellations of terms

Review: Truncation errors are different from roundoff

- Translating continuous mathematical expressions into discrete forms introduces truncation error

- For example: $e^x \simeq S(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}$

- Error: $\frac{|x|^{n+1}}{(n+1)!} \max\{1, e^x\}$

- Or $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$ vs. $D_h(x) = \frac{f(x+h) - f(x)}{h}$

Review: Epsilon check for comparing floats

- Take two real numbers `a` and `b`
- We take `a==b` if `abs(a-b) < epsilon`

- Have to be very careful with this!!! We should think about:
  - The choice of `epsilon` based on the precision we require/expect for `a` and `b`
  - The choice of `epsilon` based on the magnitude of `a` and `b`
  - What will happen in special cases (`0, NaN, inf`)
  - …

Today’s lecture:

- Good programming practices:
  - Version control
  - Testing
  - Misc. good practices

Software engineering practices

- Some basic practices that can *greatly* enhance your ability to write maintainable code
  - Version control
  - Build environments
  - Testing procedures
  - Automatic code error checking
  - Profiling
  - Documentation
- There are many tools that will help you write safe code and find bugs as they are introduced. These let you focus more on the science.
- Main goal of this lecture is to just show you what kind of tools are out there and how they can help your workflow

Coding experiences to try and avoid

- *You swear that the code worked perfectly 6 months ago*, but today it doesn't, and you can't figure out what changed
- *Your research group is all working on the same code*, and you need to sync up with everyone's changes, and make sure no one breaks the code
- *Your code always worked fine on machine X*, but now you switch to a new system/architecture, and you code gives errors, crashes, ...
- *Your code ties together lots of code*: legacy code from your advisor's advisor, new stuff you wrote, all tied together by a driver. The code is giving funny behavior sometime—how do you go about debugging such a beast?

Version control

- What is it?
  - A system that records changes to a file or set of files over time so that you can recall specific versions late

- Why is it important?
  - So that if the code stops working, you can go back to specific previous versions to see what changes broke it
  - Allows you to compare changes over time
  - If multiple people are working on a file, see who last modified something that might be causing a problem, who introduced an issue and when, etc.

Types of version control: Local

- Previous versions (or patch sets) stored elsewhere on local machine

- Can be as simple as copying files into a different folder to store them before making changes
  - Will take up a lot of memory if not done in a smart way

- There are some tools to make this more consistent such as GNU RCS

- Pros: Simplicity
- Cons: Single point of failure

Types of version control: Centralized

- Have a single server that contains all the versioned files, stores history and changes

- User communicates with the server to:
    - Checkout source
    - Commit changes back to the source
    - Request a log (history) of a file from the server
    - Diff your local version with the version on the server

- Has advantages over local version control:
    - Everyone knows what everyone else is doing on a project
    - Administrators have control over who can do what

- Cons: Does not scale well for large projects, single point of failure

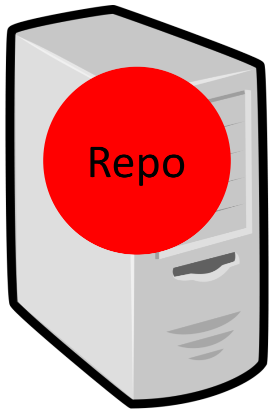
Types of version control: Distributed

- Clients fully mirror (i.e., clone) the repository and its history on their local machine
  - Not just the latest snapshot of the files
  - No single point of failure: if any server dies, any client repository can be copied back to restore it

- Deals well with multiple different groups simultaneously working on a project
  - Easy to “fork”

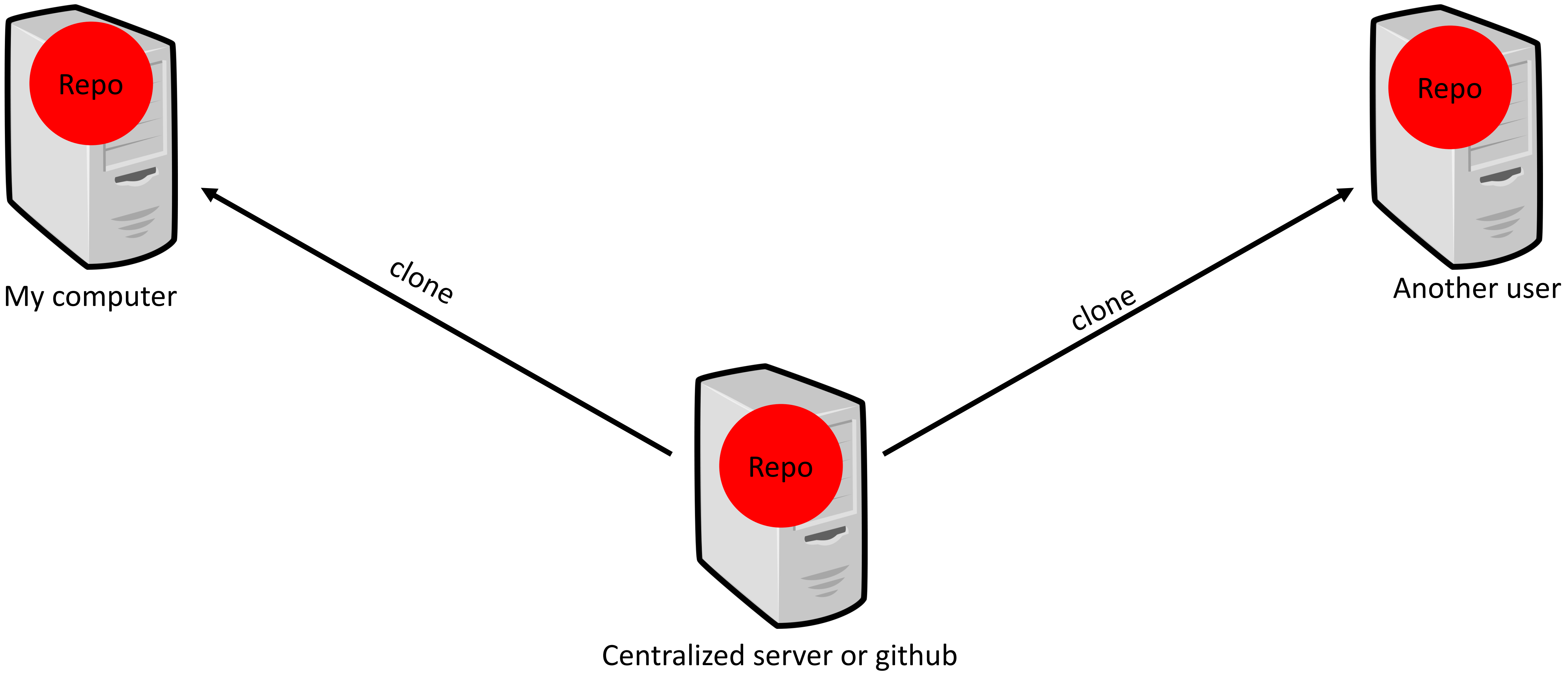
- Common DVCS: **Git**, Mercurial, Bazaar

Distributed version control

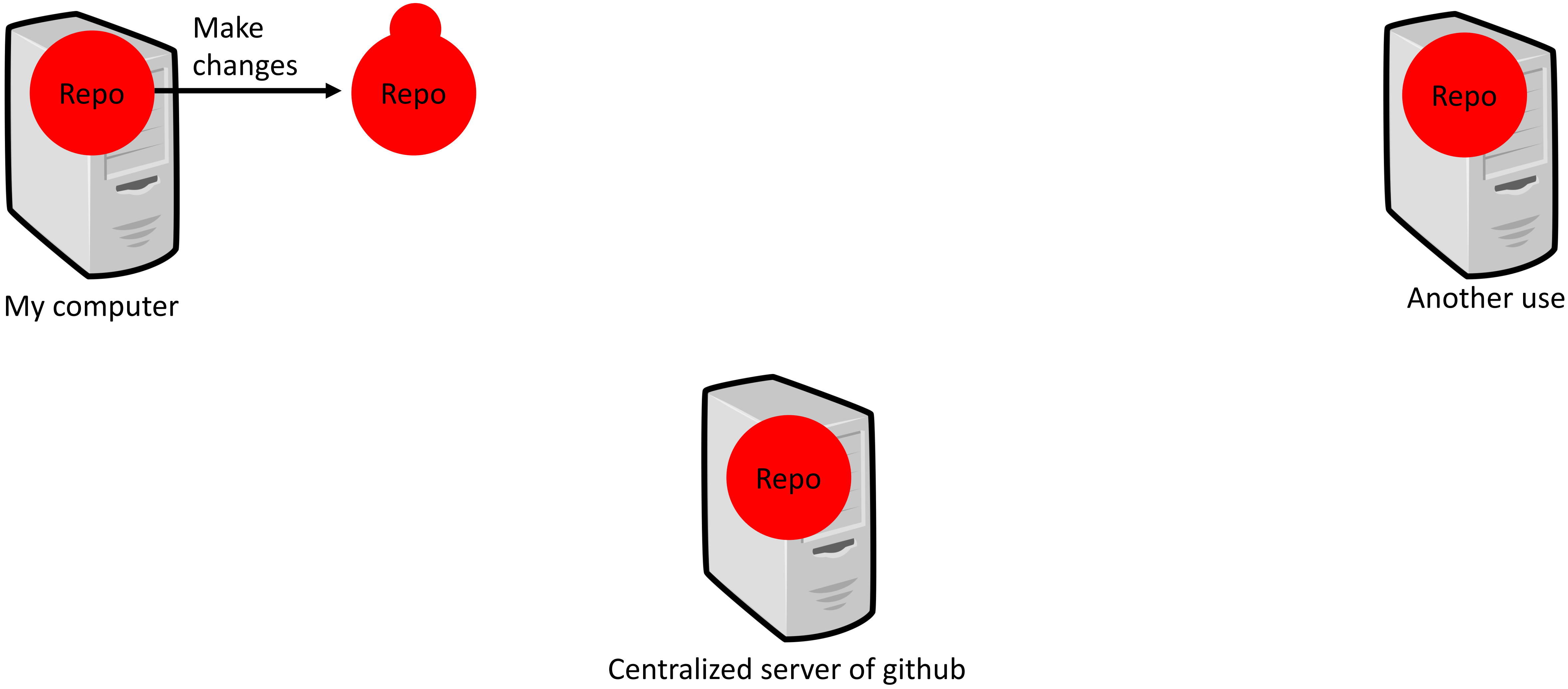


Centralized server or github

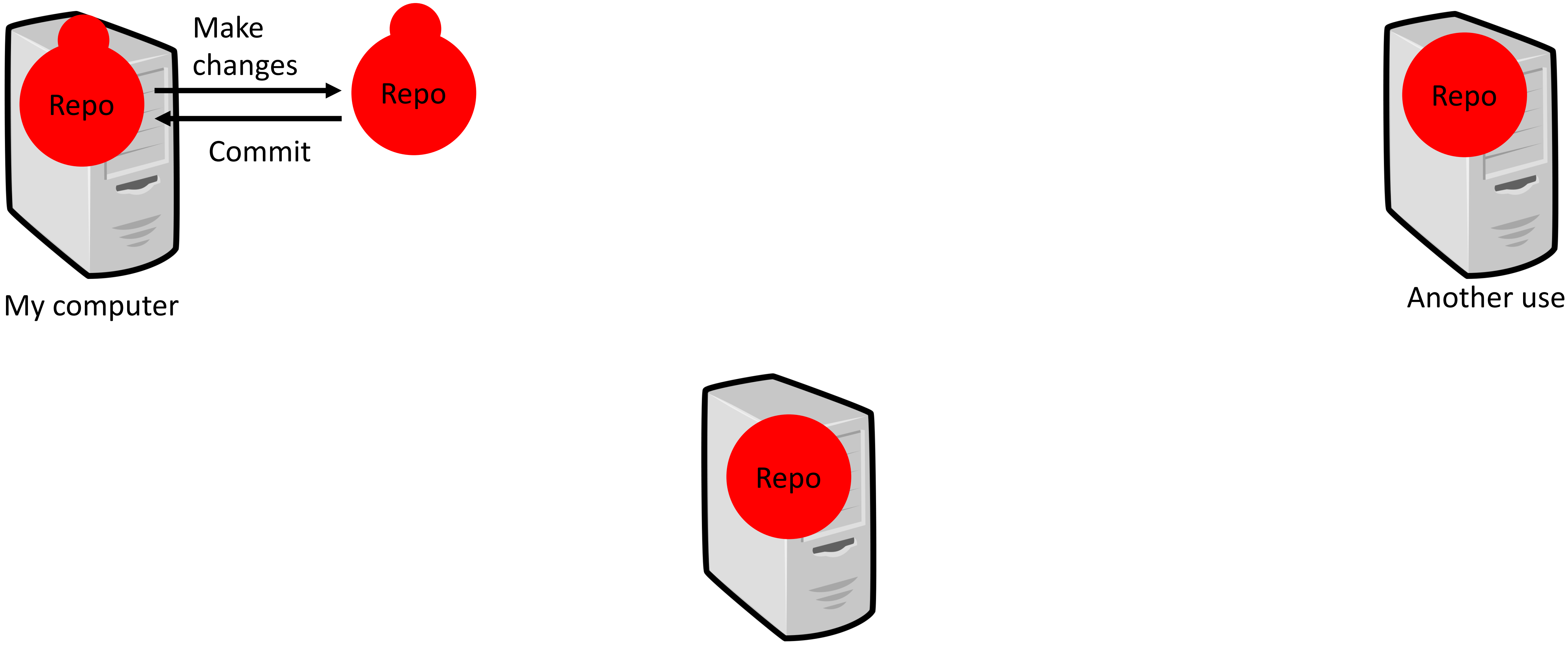
Distributed version control



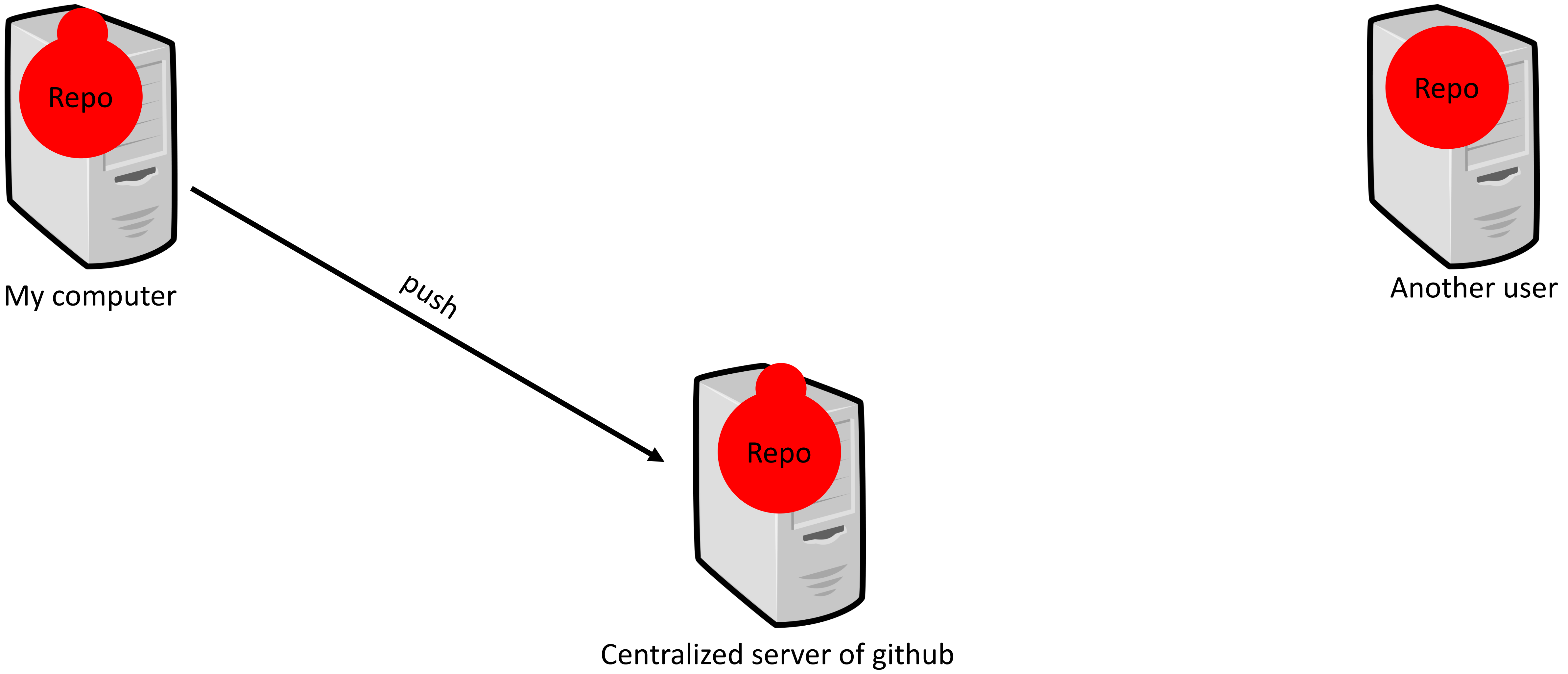
Distributed version control



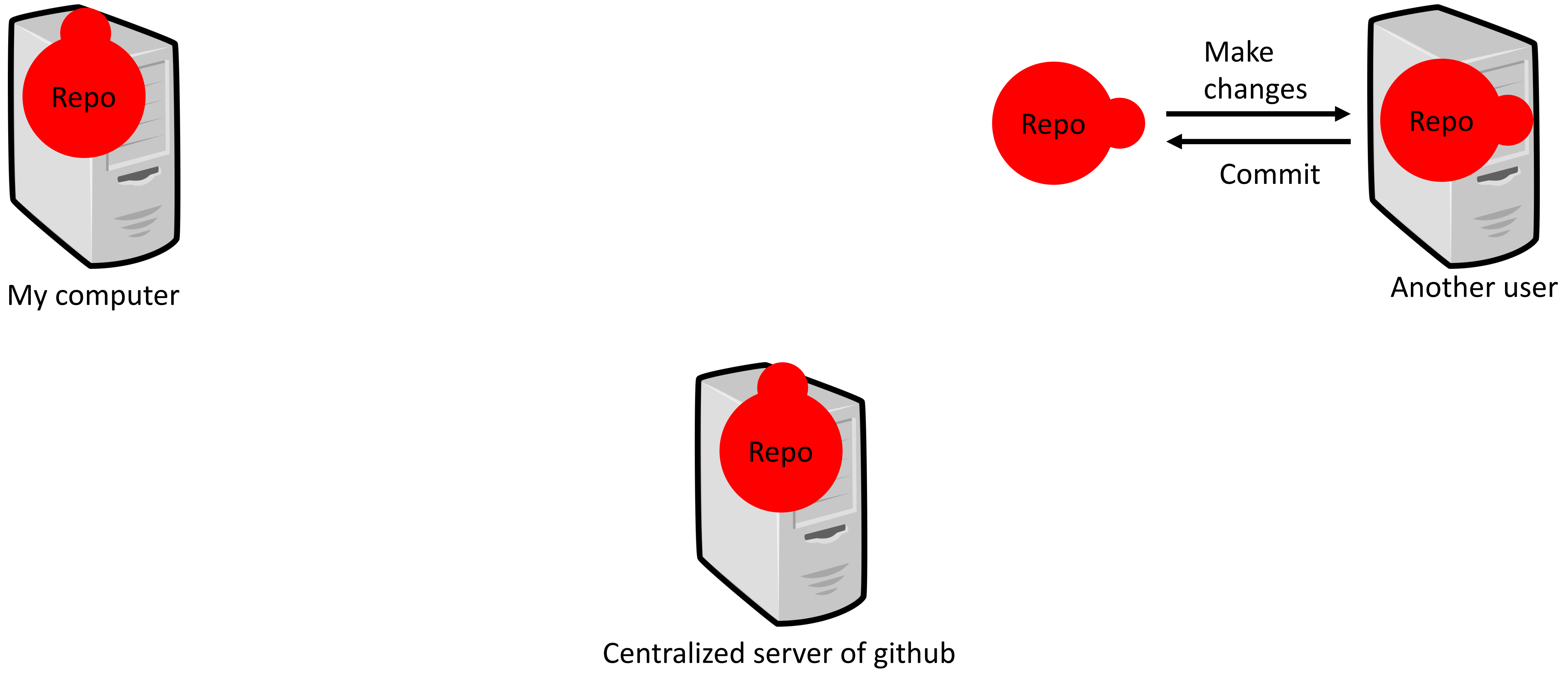
Distributed version control



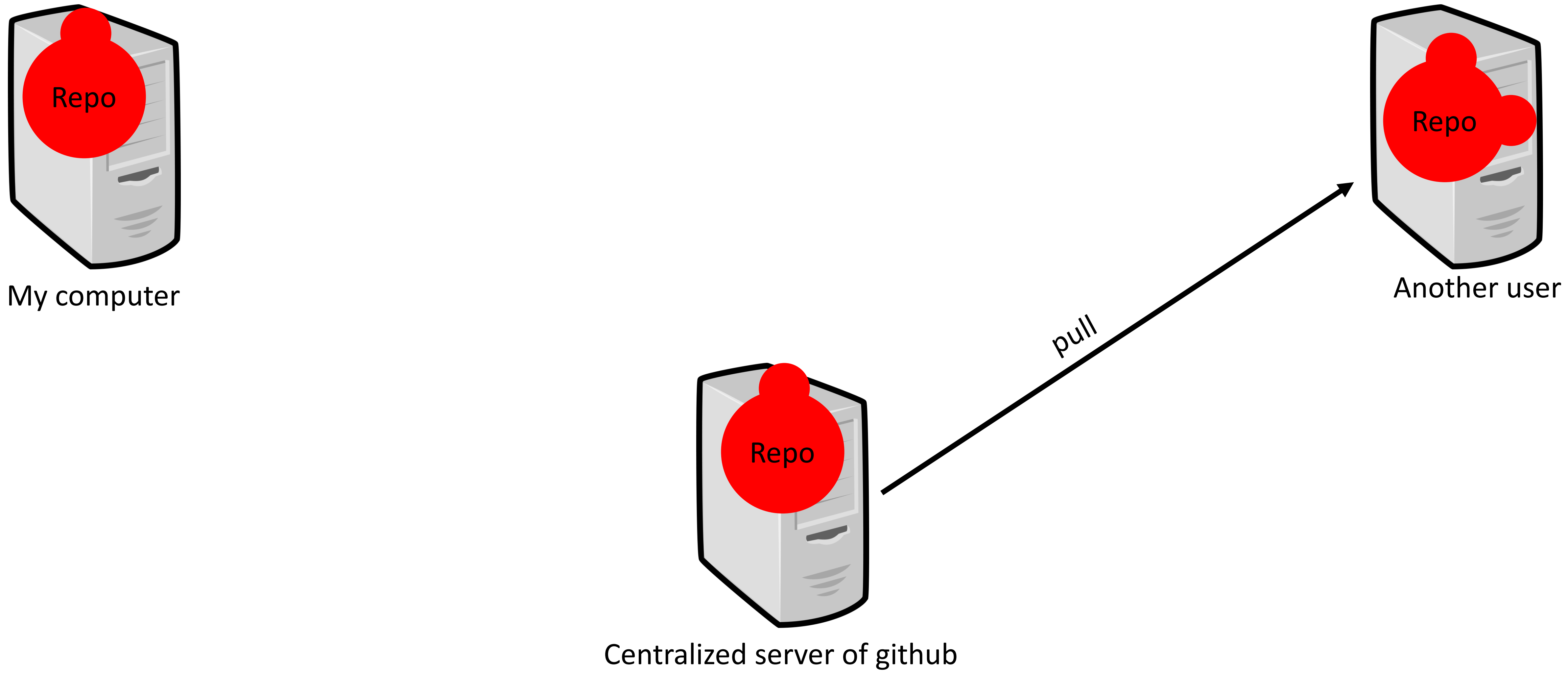
Distributed version control



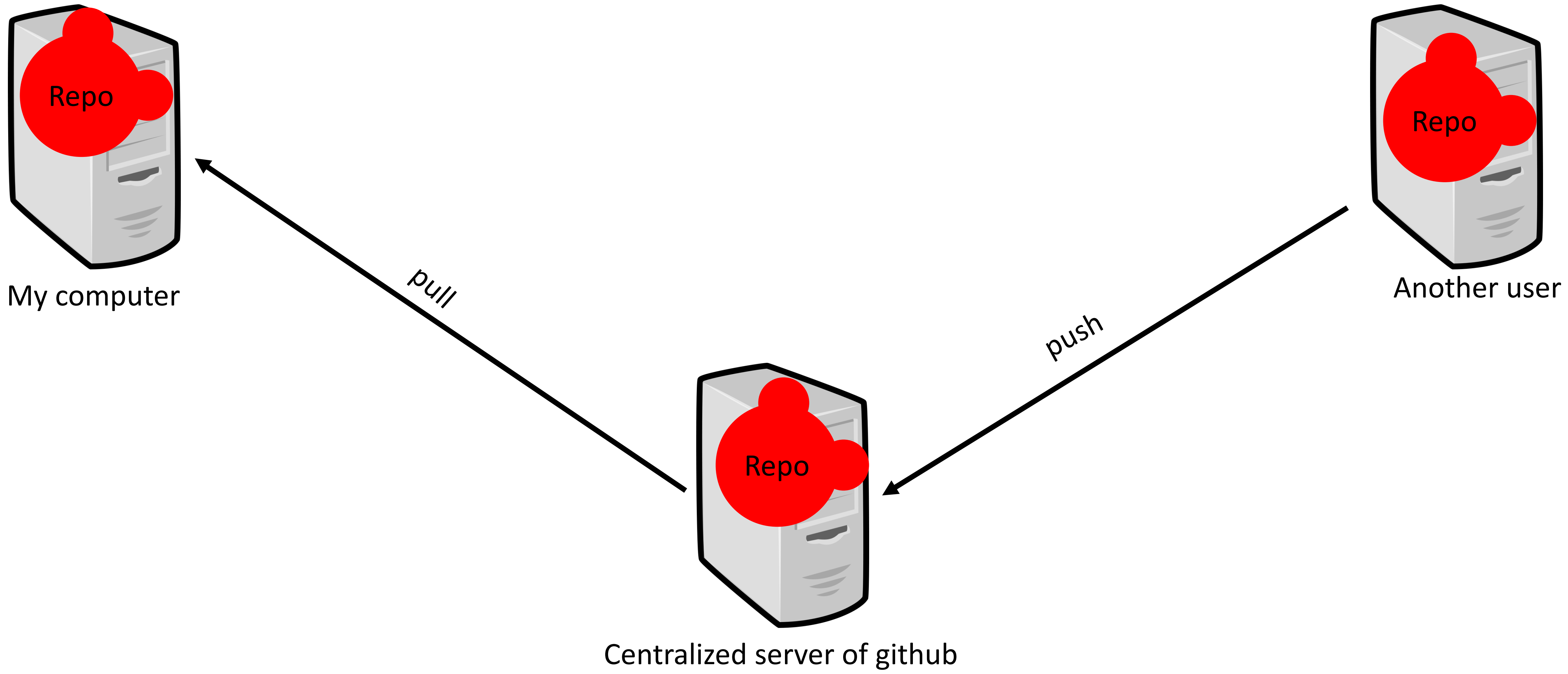
Distributed version control



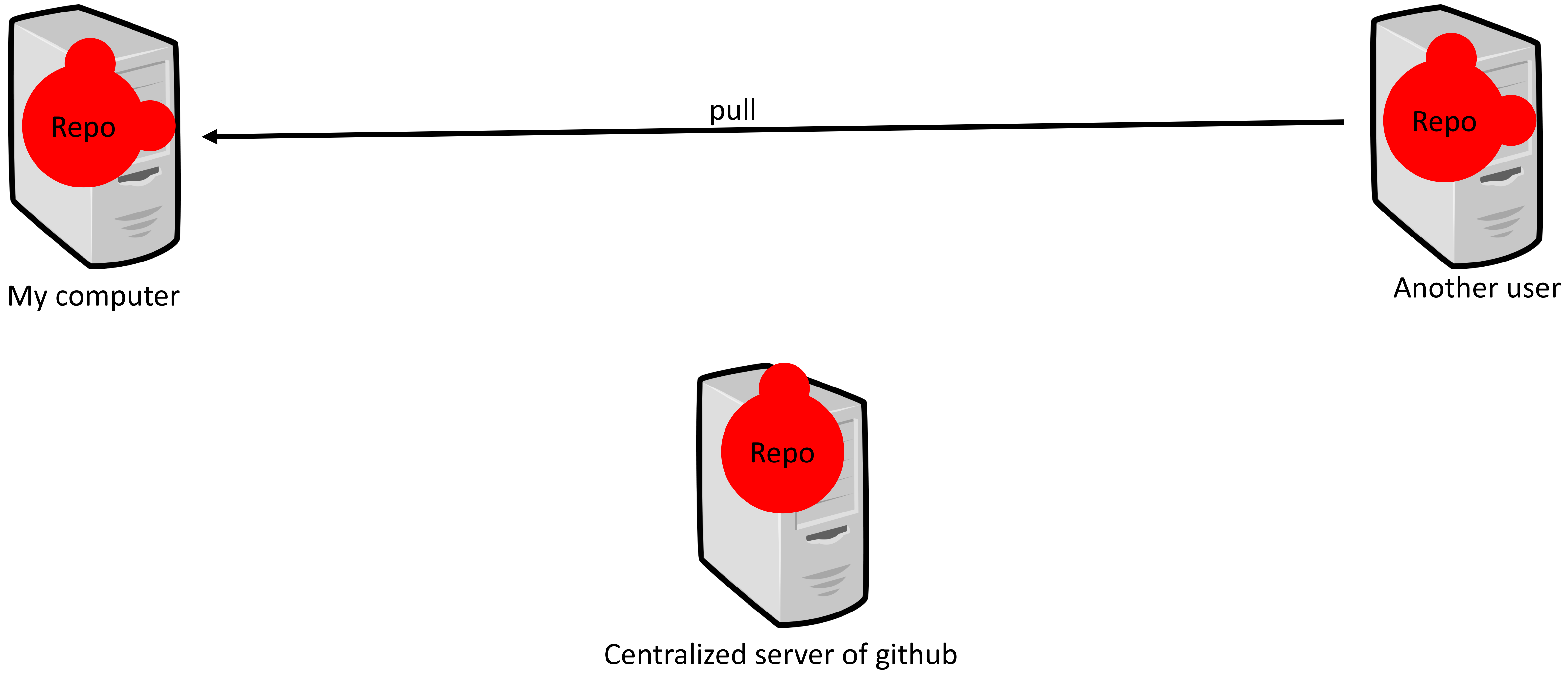
Distributed version control



Distributed version control



Distributed version control



Comments about Git

- Note that with git, every change generates a new “hash” that identifies the entire collection of source.
    - You cannot update just a single sub-directory—it's all or nothing.
- Branches in a repo allow you to work on changes in a separate are from the main source.
    - You can perfect them, then merge back to the main branch, and then push back to the remote.
    - Overall, very light weight!!
- LOTS of resources on the web (see readings)
    - Best way to learn is to practice.
    - There is more than one way to do most things

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Example: “Local” version control with Git

- You can use Git to do local version control on your computer:

- **`git init`** to create a new git repository
- **`git add`** to add file contents to the index
- **`git commit`** to record changes to the repository
- **`git log`** to show previous commits
- …

Branching with git

- One of the killer apps of git is lightweight “branching”
  - Creates a different line of development which can be merged back into the main one
  - Does not require making multiple copies of source code, etc.
- Allows you to work in different directions and later merge together as you wish
- Git will help if there are conflicting changes

Example: Simple remote on group server

- We'll look at the example of having people work with a shared remote repository—this is common with groups.
  - Each developer will have their own clone that they interact with, develop in, branch for experimentation, etc.
  - You can push and pull to/from the remote repo to stay in sync with others
  - You probably want to put everyone in the same UNIX group on the server
- Creating a master bare repo:

```
git init --bare --shared myproject.git
chgrp -R groupname myproject.git
```

(to set permissions)

Example: Simple remote on group server

- This repo is empty, and bare—it will only contain the git files, not the actual source files you want to work on
- Each user should clone it
  - In some other directory. User A does:
    - **`git clone`** ***`/path/to/`*****`myproject.git`**
- Now you can operate on it
  - Create a file (README)
  - Add it to your repo: `git add README`
  - Commit it to your repo: `git commit README`
  - Push it back to the bare repo: `git push`
- Note that for each commit you will be prompted to add a log message detailing the change

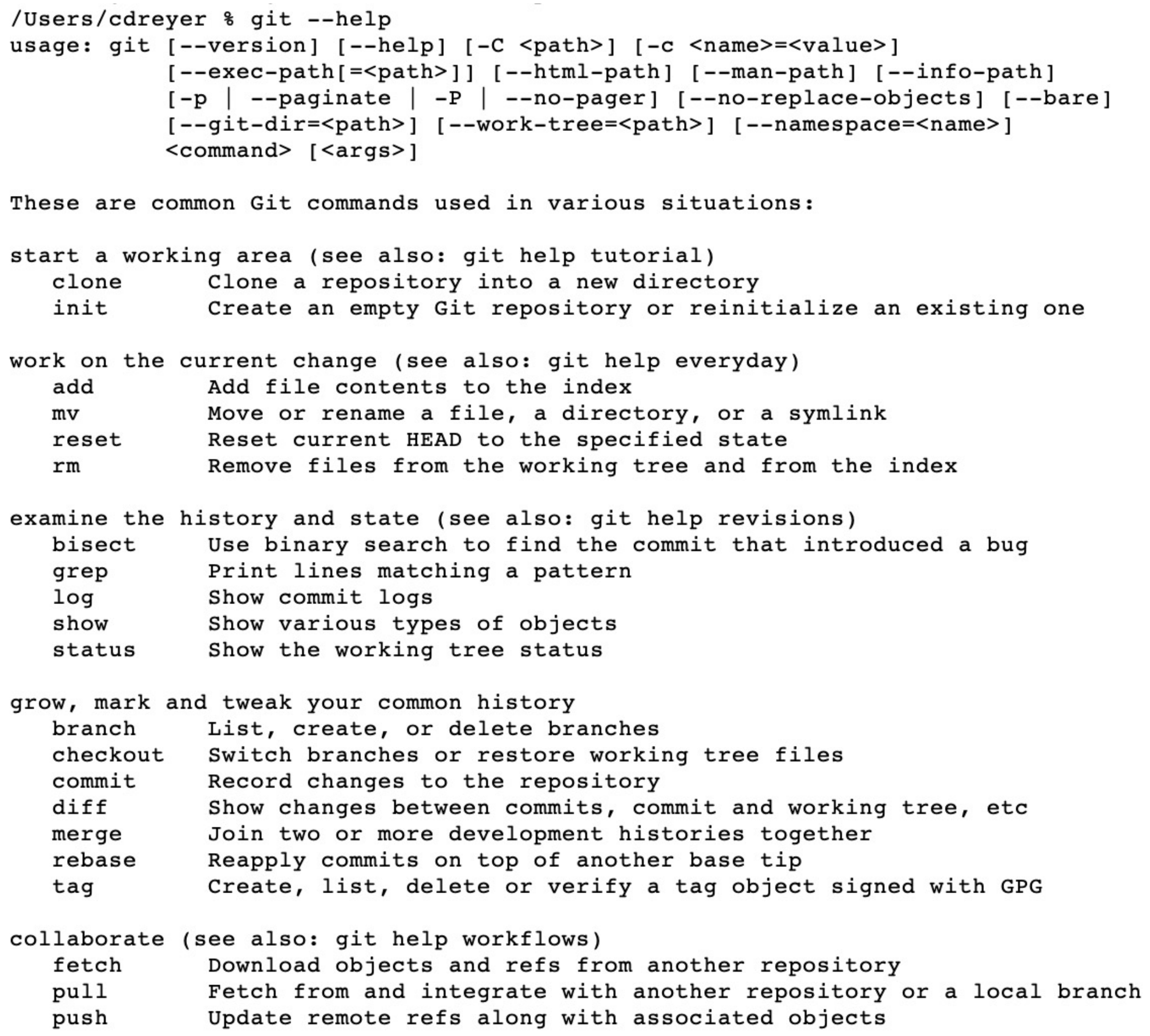
Example: Simple remote on group server

- Now user B comes along and wants to play too:
- In some other directory. User B does:
  - **`git clone`** ***/path/to/***`myrepo.git`
- Note that they already have the README file
  - Edit README
  - Commit you changes locally: **`git commit README`**
  - Push it back to the bare repo: **`git push`**
- Now user A can get this changes by doing: **`git pull`**
- In general, you can push to a *bare repo*, but you can pull from anyone

Using github

- Don't want to use your own server? Use github or bitbucket
  - Free for public (open source) projects
  - Pay for private projects
- How to contribute to someone else's project?
  - Since you are not a member of that project, you cannot push back to it
    - You don't have write access
  - Use pull requests:
    - Fork the project into your own account
    - Push back to your fork
    - Issue a *pull-request* asking for your changes to be incorporated

Common Git commands available using `git --help`



```
/Users/cdreyer % git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects
```

Some last comments about git and github

- If you put the remote repository on a different server, then you always have a backup of your project
  - Since git is distributed, if your remote server dies, each clone is a backup of the entire repo, so you are safe both ways.
- Free (for open source), online, web-based hosting sites exist (e.g. Github)
- Best with Linux or Mac OS (in terminal).
  - Windows? Try: https://git-for-windows.github.io/
- Github provides tools to share your code broadly and engage with your community
  - Pull requests, issue tracking, etc.

- We'll use git to hand in our homework assignments (more on this later)

Today’s lecture:

- Good programming practices:
  - Version control
  - Testing
  - Misc. good practices

Testing

- Testing is obviously a crucial part of writing programs

- When programs get complicated, testing is not so straight forward:
  - How do I know that a change to one part didn’t break another part?
  - How do I know what I did will work on different architectures?
  - My code crashes after running for 78 hours, where did the error originate from?

- Testing involves running the program or part of the program with some inputs and determining if the outputs are those that are expected (or at least consistent)

- Many types of testing. We will discuss unit testing and regression testing

Unit testing

- Unit testing is the practice in which each smallest, self-contained unit of the code is tested independently of the others

- There are unit testing frameworks out there that help automate the procedure for different codes
  - E.g., **`unittest`** for python

Another simple example: Matrix inversion

- Say your code has a matrix inversion routine that computes A^{-1}

- A unit test for this routine can be:
    - Pick a vector x
    - Compute b = A x
    - Compute x = A^{-1} b
    - Does the x you get match (to machine tol) the original x?

Regression Testing

- Imagine you've “perfected” your program (simulation tool, analysis tool, etc.)
  - You are confident that the answer it gives is “right”
  - You want to make sure that any changes you do in the future do not change the output
  - Regression testing tests whether changes to the code change the solution
- Regression testing:
  - Store a copy of the current output (a benchmark)
  - Make some changes to the code
  - Compare the new solution to the previous solution
  - If the answers differ, either:
    - You've introduced a bug → fix it
    - You've fixed a bug → update your benchmark

Regression testing

- Simplest requirements:
  - You just need a tool to compare the current output to benchmark
  - You can build up a more complex system from here with simple scripting

- Big codes need a bunch of tests to exercise all possible options for the code
  - If you spend a lot of time hunting down a bug, once you fix it, put a test case in your suite to check that case
  - If someone implements a new functionality, ask them to submit a test
  - You'll never have complete coverage, but your number of tests will grow with time, experience, and code complexity

Today's lecture:

- Good programming practices:
  - Version control
  - Testing
  - Misc. good practices

Comments and Documentation

- Many in computer science will say that "good code documents itself"
  - **Do not believe it.**
  - Remember, we are often writing code for programming novices (both the developers and users)
  - The better people can understand your code, the more productive science will be done with it
- No hard-and-fast rules. Comments should explain the basic idea of what a block of code does
  - Only comment "single lines" if there is something special or unusual about them
  - Keep comments up to date with the code
  - Think about what information will be useful for you in the future, and other developers of your code
- Can often use tools to turn comments in the source into external documentation
  - Robodoc: https://rfsber.home.xs4all.nl/Robo/
  - FORD: http://fortranwiki.org/fortran/show/FORD
  - Pydoc: https://docs.python.org/3/library/pydoc.html
  - Others for python: https://wiki.python.org/moin/DocumentationTools

Debugging tools

- Simplest debugging: print out information at intermediate points in code execution
- Running with appropriate compiler glags (e.g., **–g** for gnu compilers) can provide debugging information
  - Can make code run slower, but useful for test purposes
- Interactive debuggers let you step through your code line-by-line, inspect the values of variables as they are set, etc.
  - gdb is the version that works with the GNU compilers. Some graphical frontends exist.
  - Lots of examples online
  - Not very useful for parallel code.
- Particularly difficult errors to find often involve memory management
  - **Valgrind** is an automated tool for finding memory leaks. No source code modifications are necessary.

Building your code with, e.g., Makefiles

- It is good style to separate your subroutines/functions into files, grouped together by purpose
  - Makes a project easier to manage (for you and version control)
  - Reduces compiler memory needs (although, can prevent inlining across files)
  - Reduces compile time—you only need to recompile the code that changed (and anything that might depend on it)
- Makefiles automate the process of building your code
  - No ambiguity of whether your executable is up-to-date with your changes
  - Only recompiles the code that changed (looks at dates)
  - Very flexible: lots of rules allow you to customize how to build, etc.
  - Written to take into account dependencies

We have not really discussed general coding style

- Depends very much on the language, and is often a matter of opinion (google it)
- Some general rules:
  - 1. Use a consistent programming style
  - 2. Use brief but descriptive variable and function names
  - 3. Avoid "magic numbers"
    - Name your constants, specify your flags
  - 4. Use functions and/or subroutines for repetitive tasks
  - 5. Check return values for errors before proceeding
  - 6. Share information effectively (e.g., using modules or namespaces)
  - 7. Limit the scope of your variables, methods, etc.
  - 8. Think carefully about the most effective way to input and output data
  - 9. Be careful about memory, i.e., allocating and deallocating
  - 10. Make your code readable and portable, you will thank yourself (or your collaborators will thank you) later.

After class tasks

- No office hours today (8/26/21)

- If you do not already have one, make an account on github: https://github.com/

- Readings:
  - Wikipedia artical on makefiles
  - Pro Git online book
  - Fortran best practices
  - Good Enough Practices in Scientific Computing