

PHY604 Lecture 2

August 28, 2025

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 your 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 later
- 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

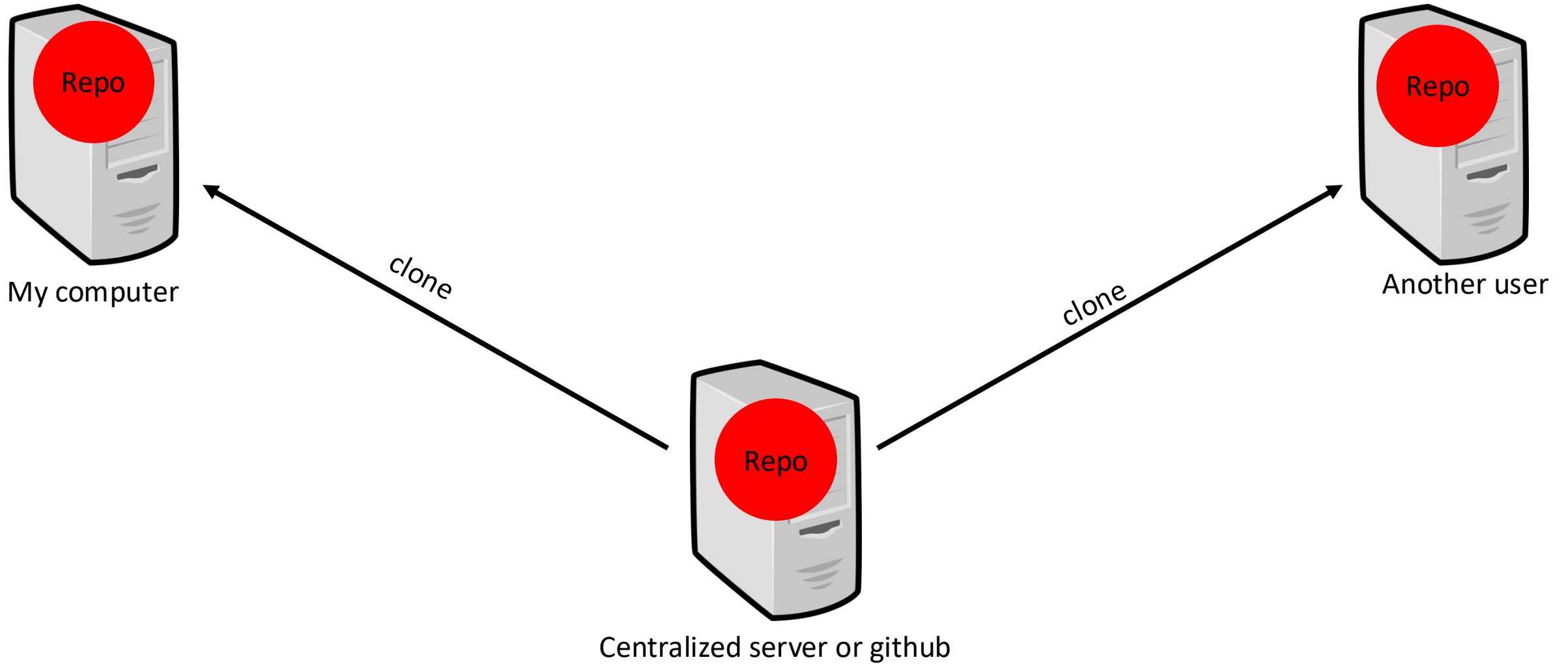
- Clients fully mirror (i.e., clone) the repository and its history on their local machine
 - Not just the latest snapshot of the files
 - No single point of failure: if any server dies, any client repository can be copied back to restore it
- Deals well with multiple different groups simultaneously working on a project
 - Easy to “fork”
- Common DVCS: **Git**, Mercurial, Bazaar

Distributed version control

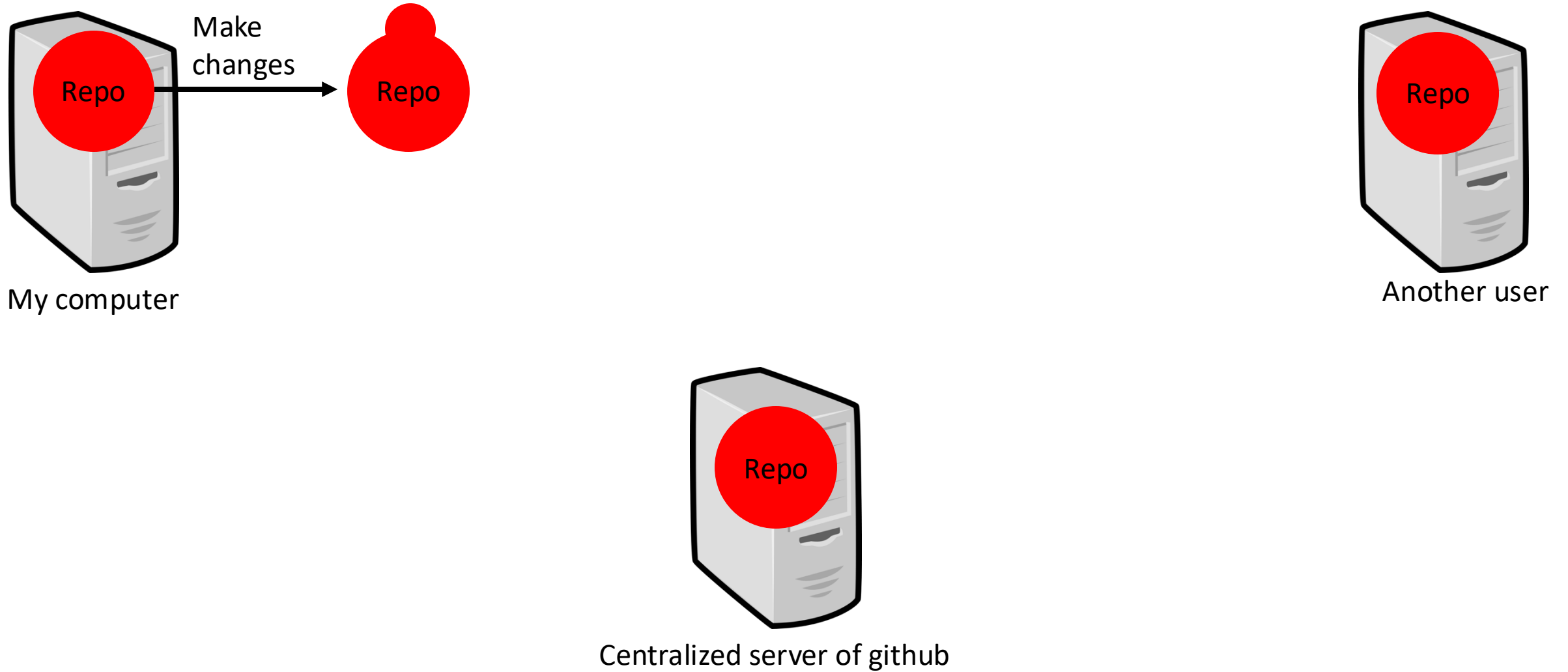


Centralized server or github

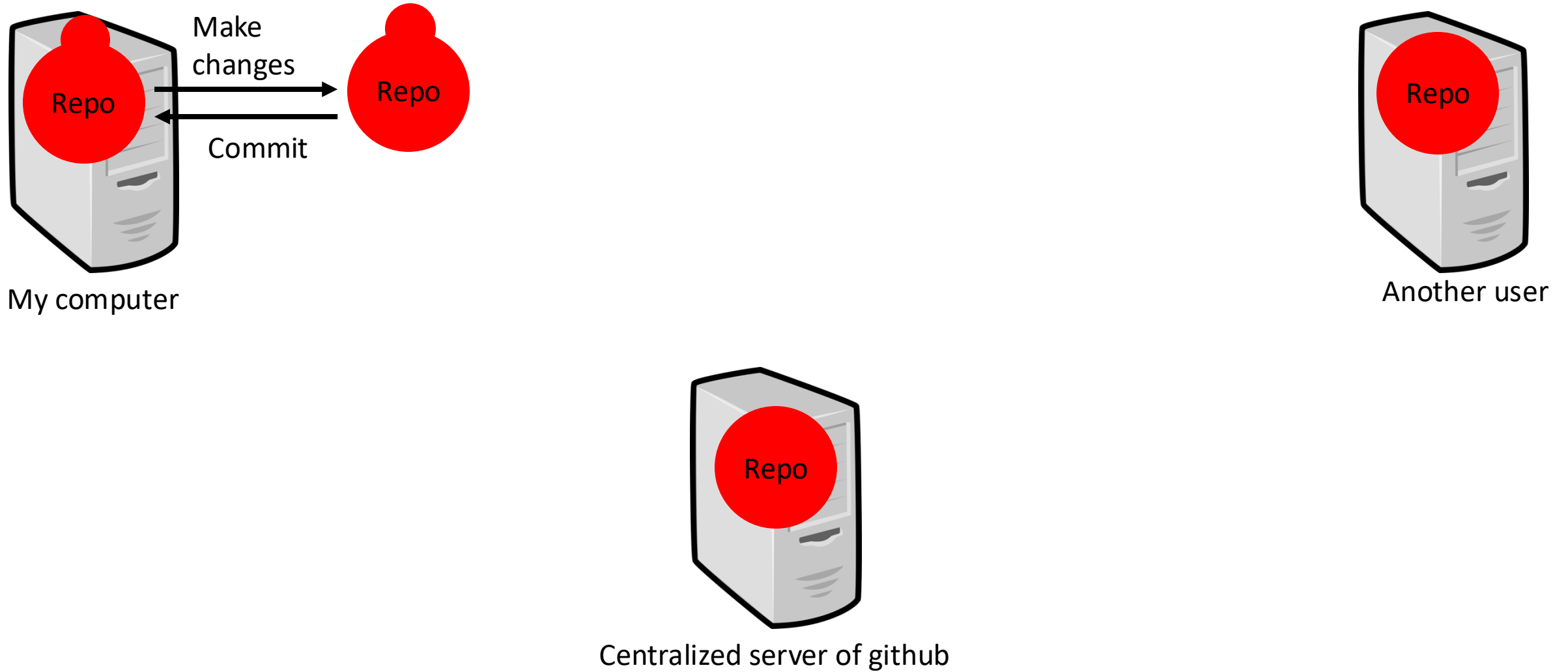
Distributed version control



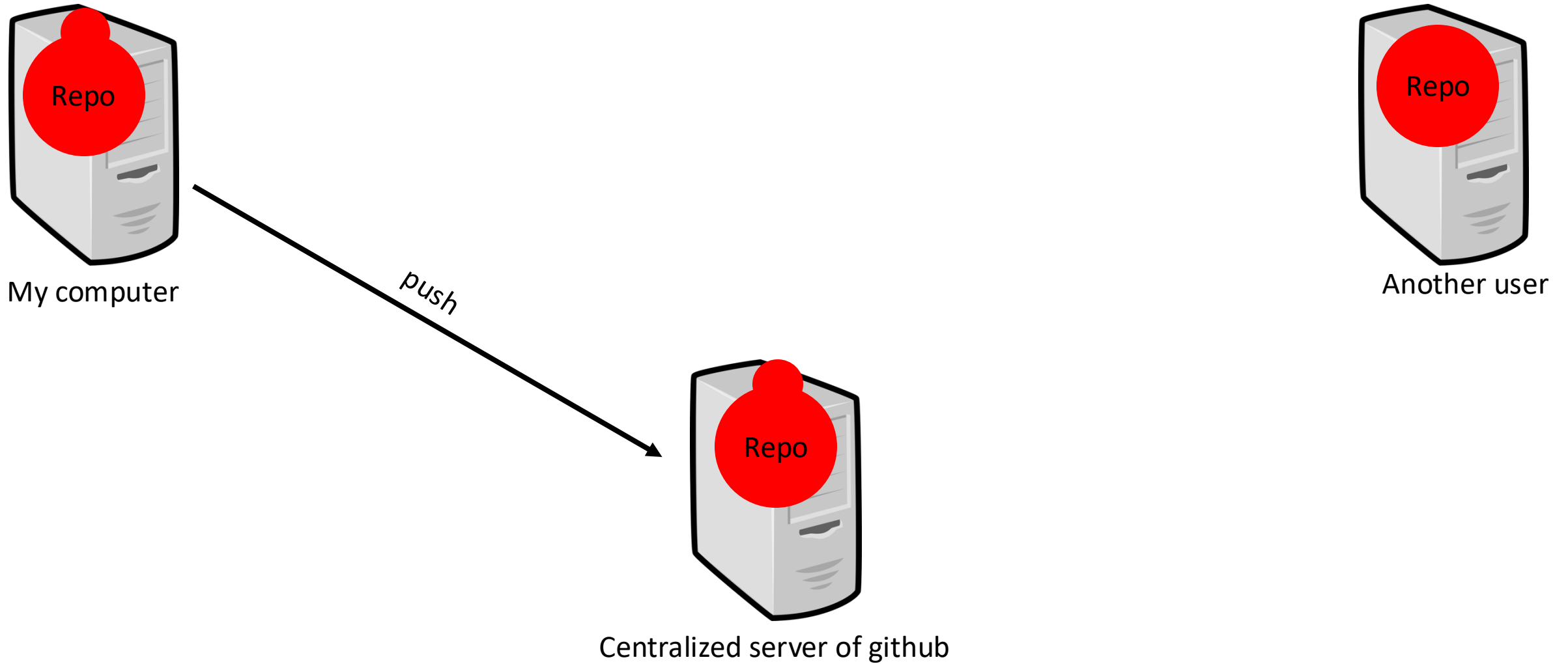
Distributed version control



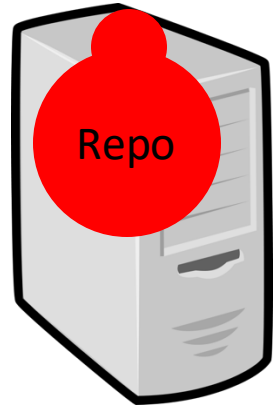
Distributed version control



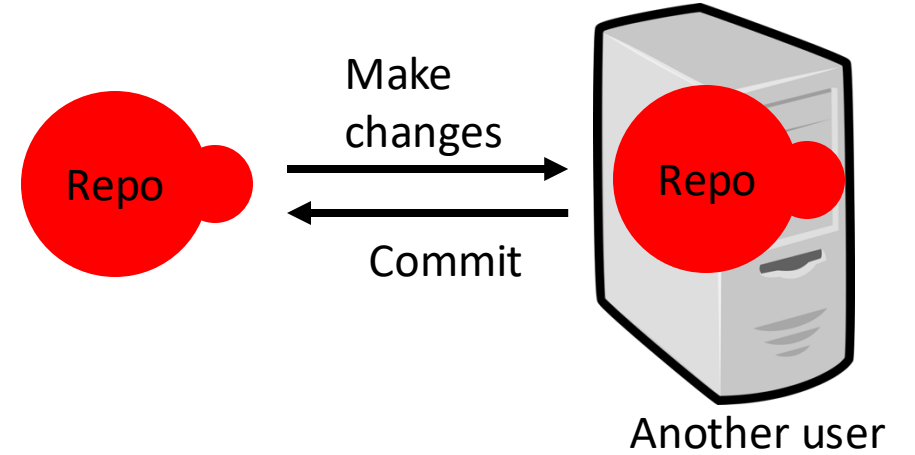
Distributed version control



Distributed version control

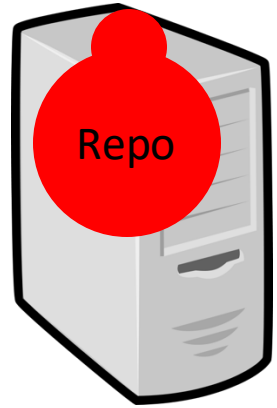


My computer



Centralized server of github

Distributed version control



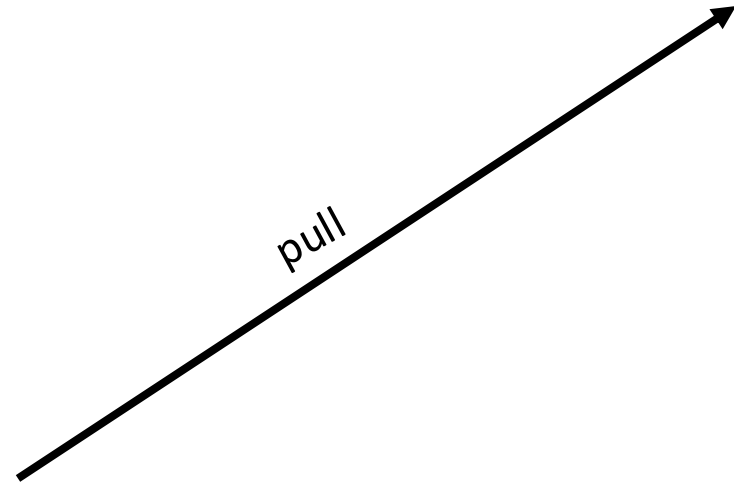
My computer



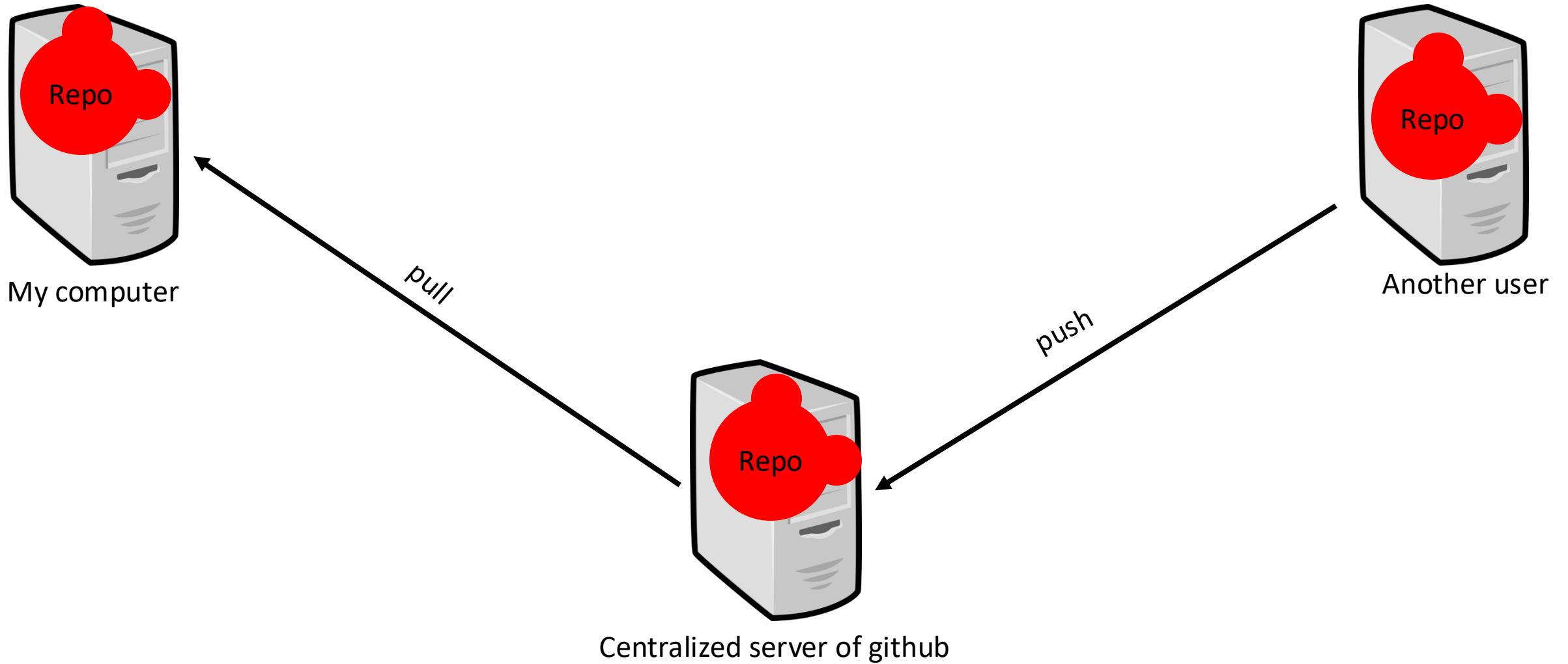
Centralized server of github



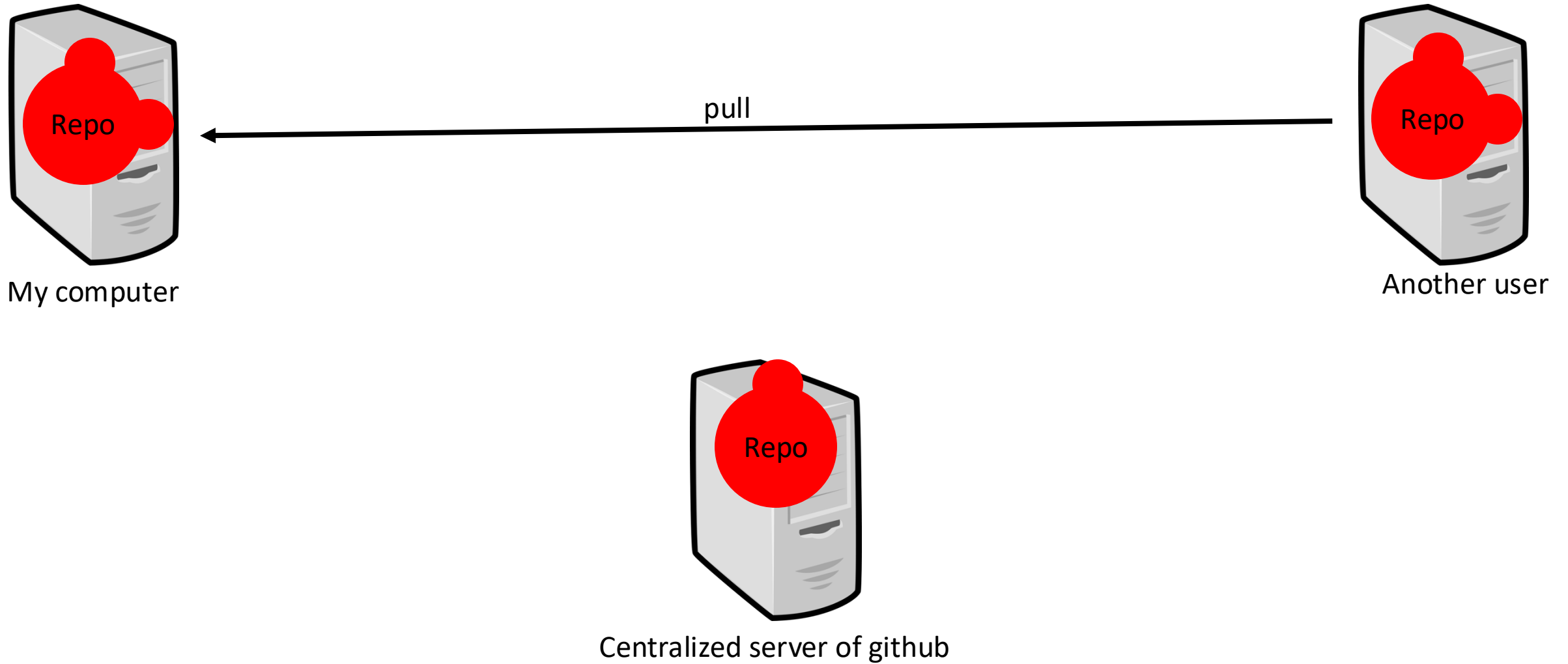
Another user



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 area 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



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/cdreier % 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`)

<code>clone</code>	Clone a repository into a new directory
<code>init</code>	Create an empty Git repository or reinitialize an existing one

work on the current change (see also: `git help everyday`)

<code>add</code>	Add file contents to the index
<code>mv</code>	Move or rename a file, a directory, or a symlink
<code>reset</code>	Reset current HEAD to the specified state
<code>rm</code>	Remove files from the working tree and from the index

examine the history and state (see also: `git help revisions`)

<code>bisect</code>	Use binary search to find the commit that introduced a bug
<code>grep</code>	Print lines matching a pattern
<code>log</code>	Show commit logs
<code>show</code>	Show various types of objects
<code>status</code>	Show the working tree status

grow, mark and tweak your common history

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

collaborate (see also: `git help workflows`)

<code>fetch</code>	Download objects and refs from another repository
<code>pull</code>	Fetch from and integrate with another repository or a local branch
<code>push</code>	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>

After class tasks and further reading

- 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](#)