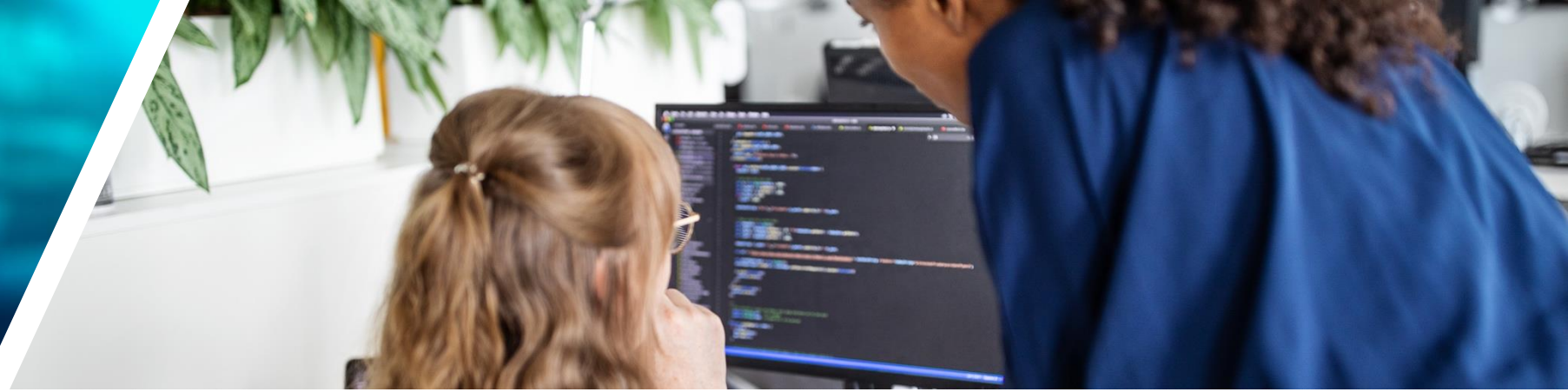


VERSION CONTROL

CSC 207 SOFTWARE DESIGN

LEARNING OUTCOMES

- Understand what version control is and why it is useful
- Understand the basic git commands

WHAT IS VERSION CONTROL?

- Software development teams need a way to track how their software changes over time.
- A master repository of files exists on a server.
- People “clone” the repo to get their own local copy.
- As significant progress is made, people “push” their changes to the master repo, and “pull” other people’s changes from the master repo.
- The repo **keeps track of every change**, and people can revert to older versions.

Further reading:
https://www.atlassian.com/git/tutorials/what-is-version-control

WHY VERSION CONTROL?

WHY VERSION CONTROL?

- **Backup and restore** - because accidents happen
- **Synchronization** - multiple people can make changes
- **Short term undo** - that last change made things worse?
- **Long term undo** - find out when a bug was introduced
- **Track changes** - all changes related to a bug fix
- **Sandboxing** - try something out without messing up the main code
- **Branching and merging** - (better defined sandboxes)

https://betterexplained.com/articles/a-visual-guide-to-version-control/

PLAIN TEXT FORMATS

- Plain text formats (Java code, python code, markdown, LaTeX, any other "human-readable" file formats) are well suited to version control.
- Since changes are tracked, one can easily look at the diff (difference) between two versions of the file being tracked.
- Formats like .docx or .xlsx are less meaningfully stored in a version control system, since the differences when you change them don't have any human-interpretable meaning. For those formats, the software developers introduce features like "track changes mode".

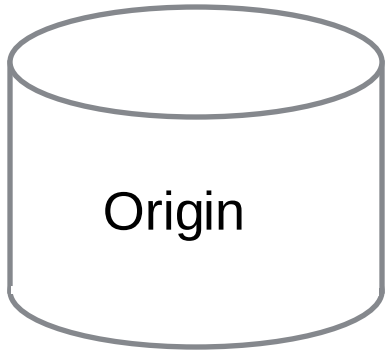
GIT

- For CSC207, we will usually create the remote repository for you.
- You will **clone** your remote repository, work on files locally, **add** and **commit** changes to your local repository, and **push** your changes to the remote repository.
- We'll only teach the basics in this course, so that you know enough to use it in your group project. (learn as you go)

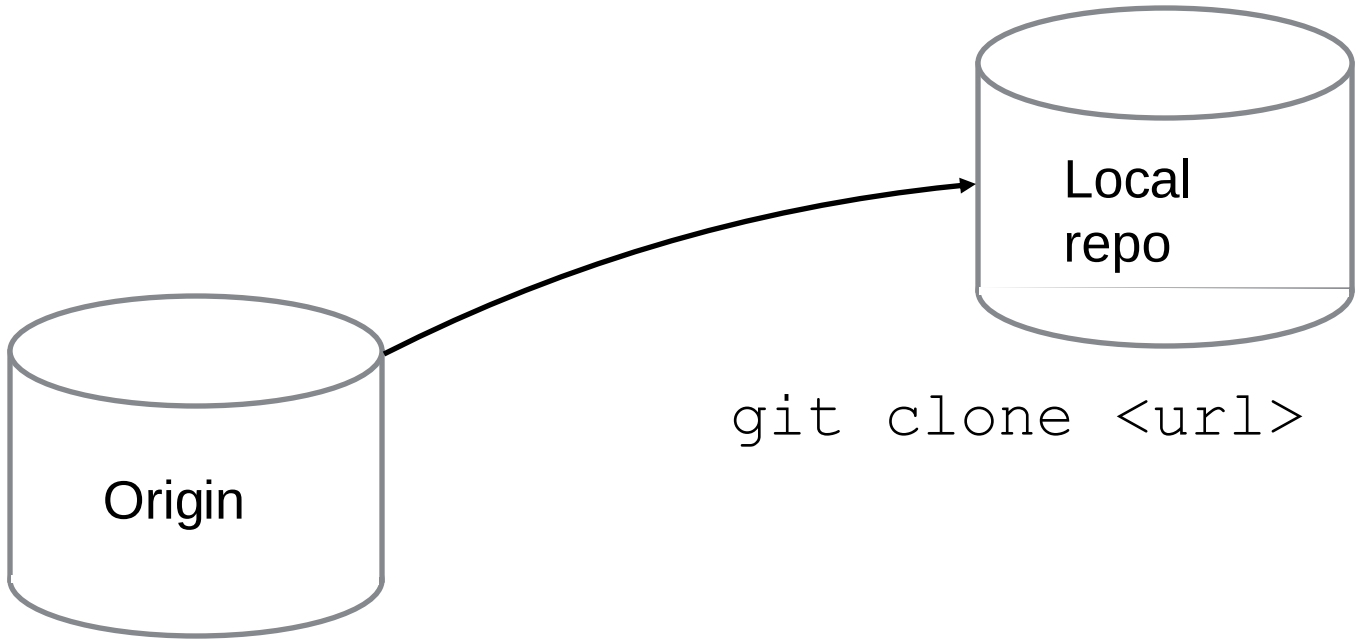
Further reading: In the spirit of thinking about software design, we encourage you to read https://en.wikipedia.org/wiki/Git, which discusses the motivation for why Linus Torvalds decided to create git, as well as the system's design.

REMOTE REPOSITORY

- This is the repo that lives on another server.
- By convention we call it the *origin*

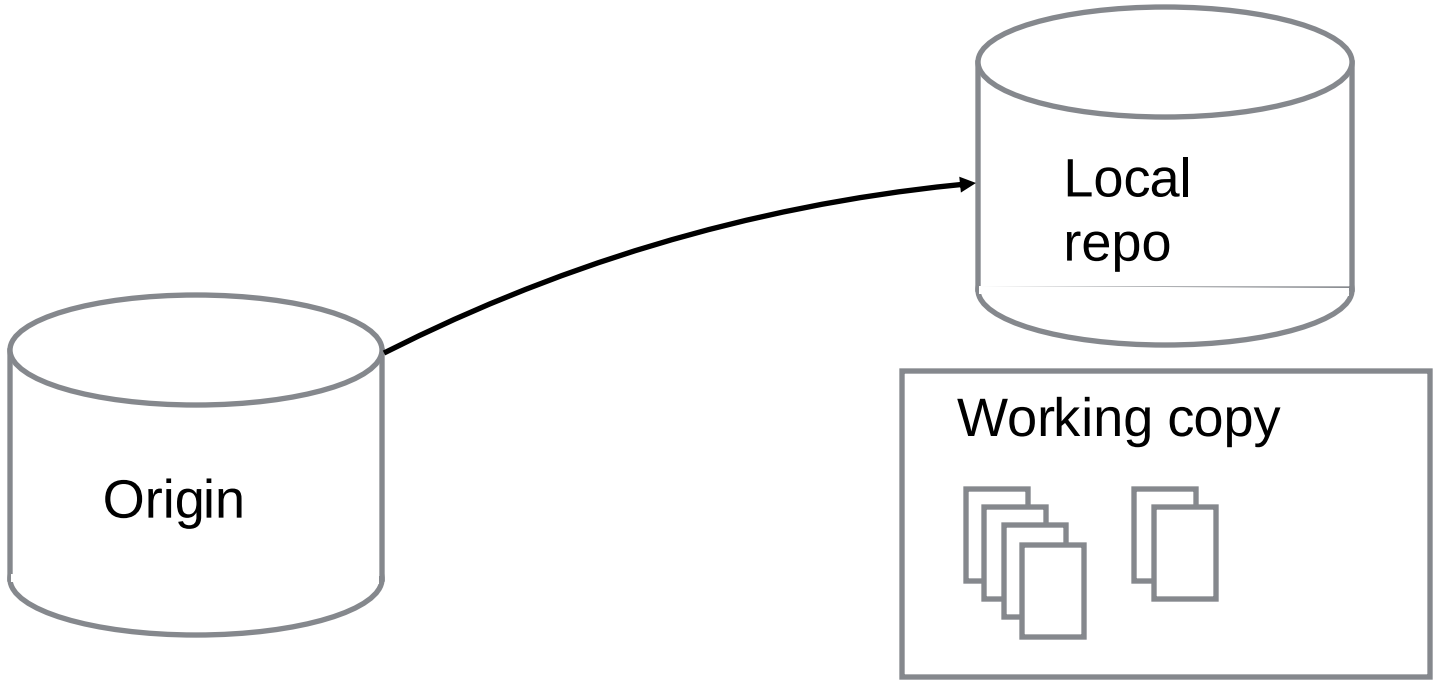


CLONE TO GET LOCAL REPO



- The “clone” command makes a copy of the remote repository on your local machine.

CLONE TO GET LOCAL REPO



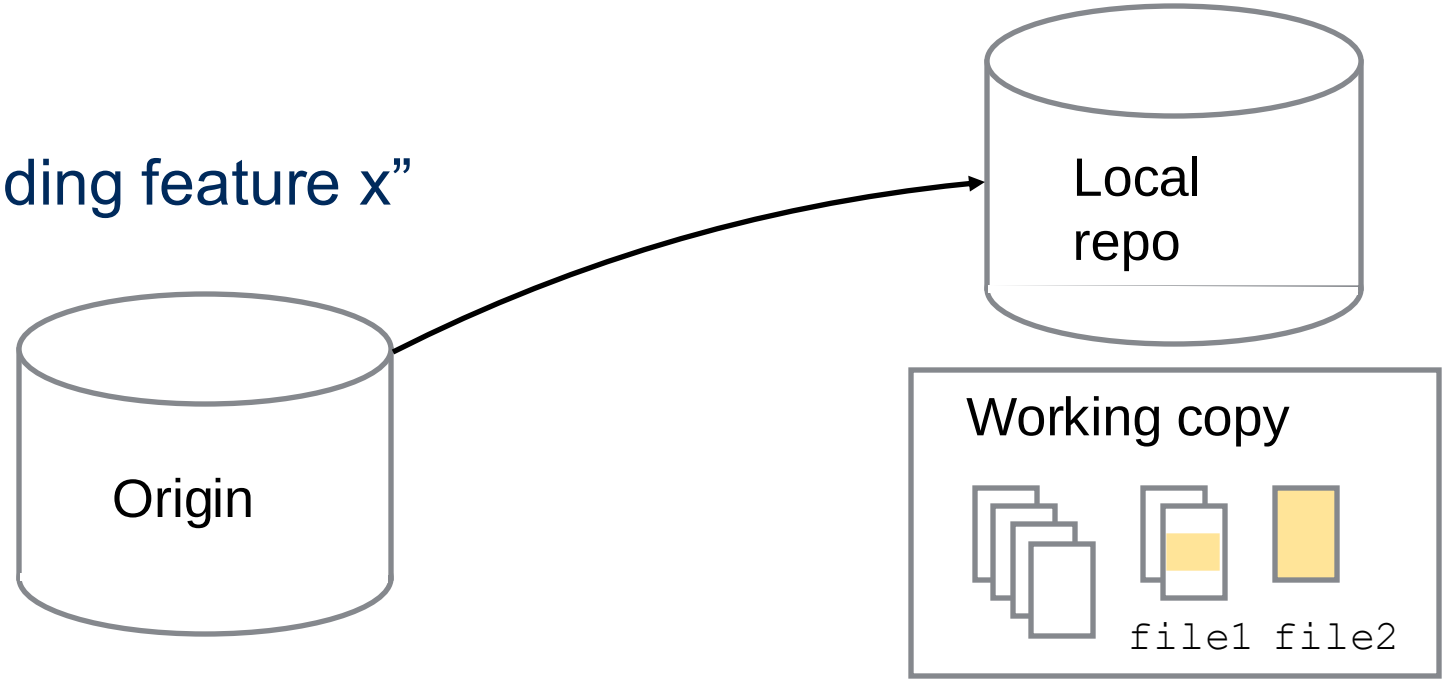
- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a *distributed* version control system)

LOCAL REPOSITORY

- The actual repository is hidden. What you see and edit is the working copy of the files from the repository.
- You can create new files, make changes to files, and delete files (as you usually do).
- When you want to “commit” a change to the local repository, you need to first “stage” the changes.

HOW TO GET WORK DONE

- Make changes to files, add new files. When you are ready to commit your work to your local repo, you need to tell git which files have changes that you want to add this time.
- git add file1 file2
- git commit -m “adding feature x”



STAGING CHANGES

- `git add` doesn't add files to your repo. Instead, it marks a file as being part of the current change.
- This means that when you make some changes to a file and then add and commit them, the next time you make some changes to a file you will still have to run `git add` to add the changes to the next commit.

GIT STATUS

- A file can be in one of 4 states:
  - **untracked** – you have never run a git command on the file
  - **tracked** – committed
  - **staged** – `git add` has been used to add changes in this file to the next commit
  - **dirty/modified** – the file has changes that haven't been staged

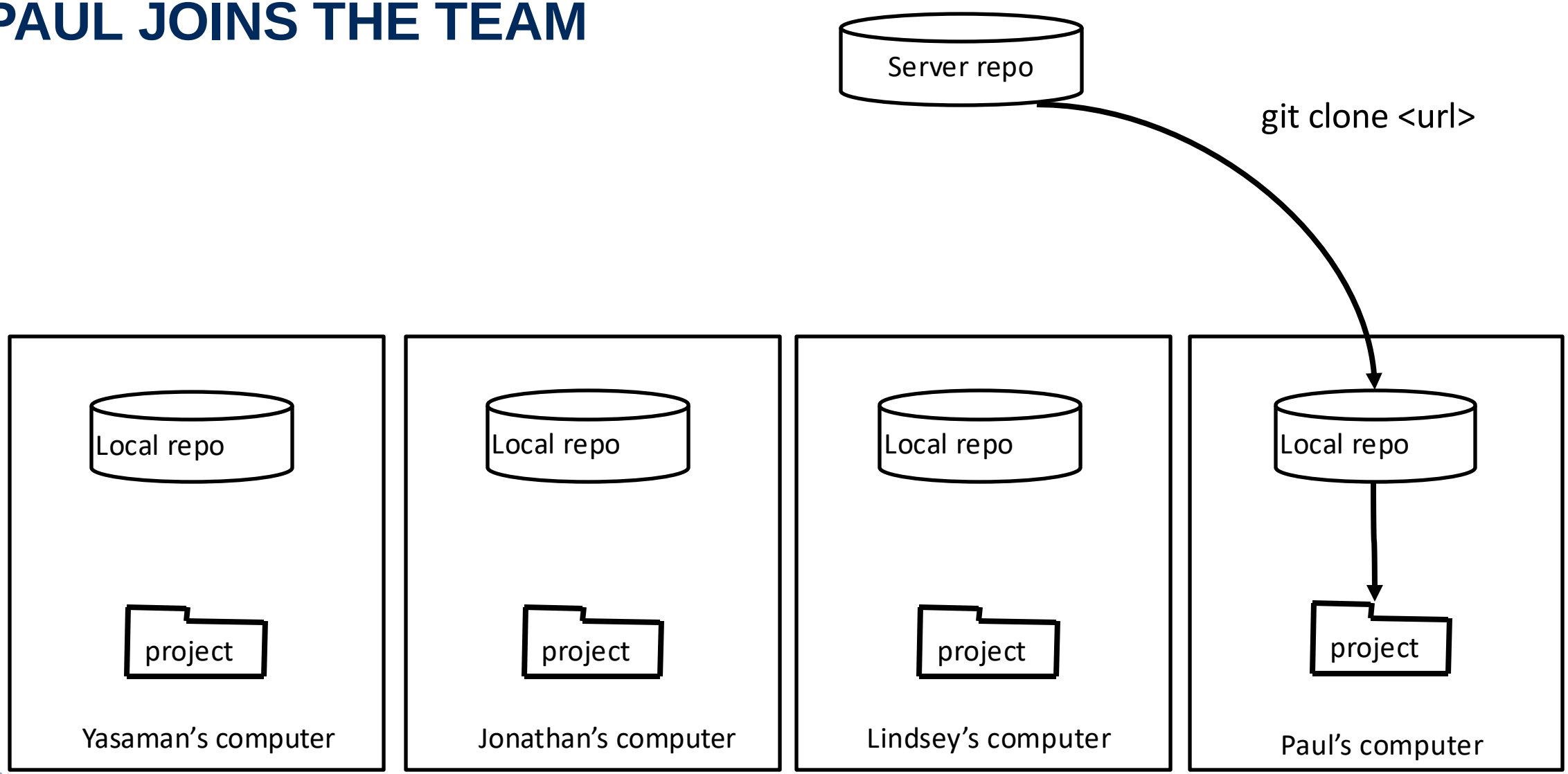
TIP: Use `git status` *regularly*. It helps you make sure the changes you have made really make it into the repo!

  - IntelliJ uses colours and symbols to help you easily see the status of files

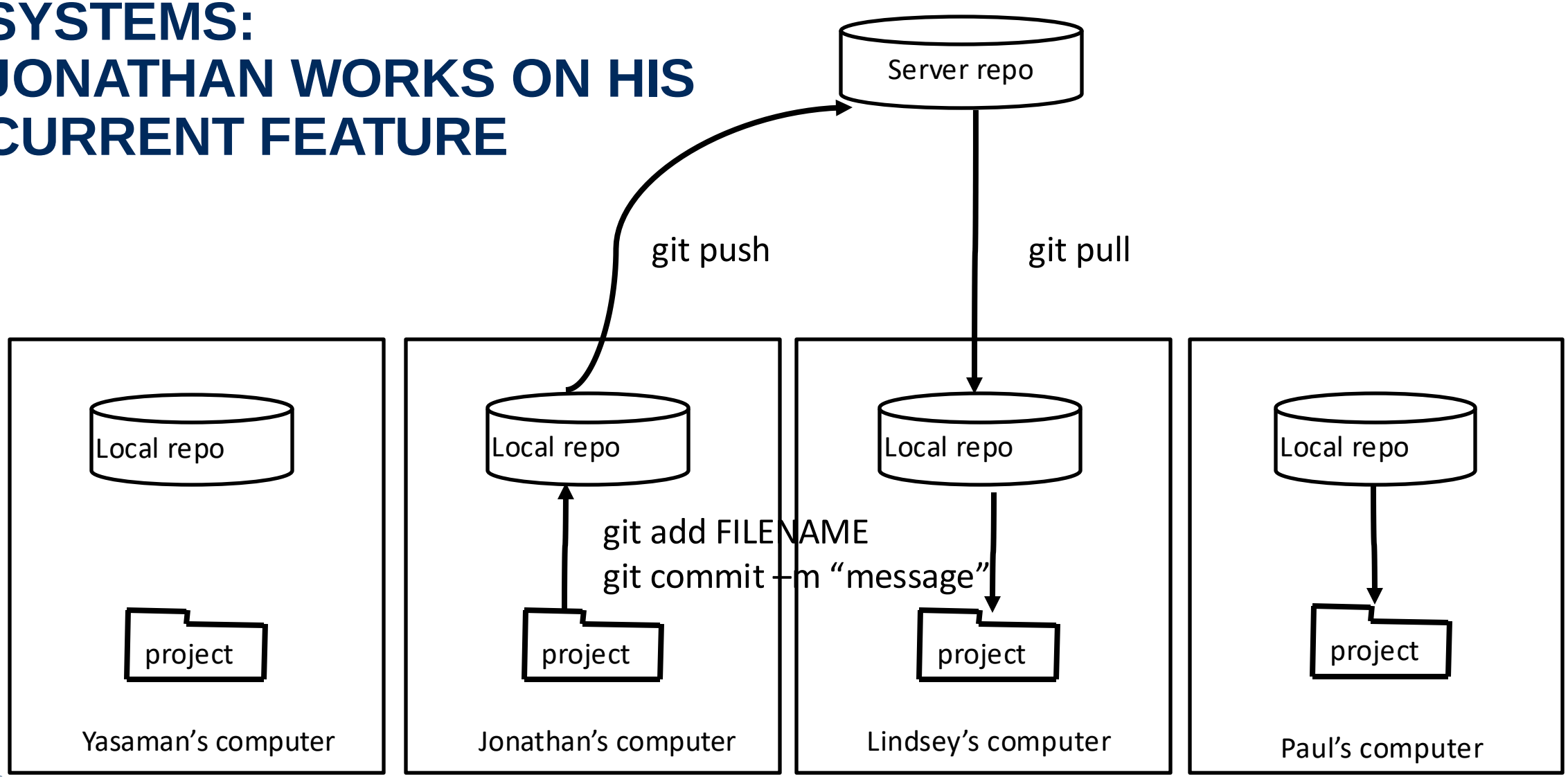
BASIC WORKFLOW (NO BRANCHING)

- Starting a project:
  - `git clone <url>`
- Normal work:
  - After you have made some changes
  - `git status` (see what has really changed)
  - `git add file1 file2 file3`
  - `git commit -m "meaningful commit message"`
  - `git push`
- In IntelliJ, you can either type these commands in the Terminal or use the graphical user interface it provides.

DISTRIBUTED VERSION CONTROL SYSTEMS: PAUL JOINS THE TEAM

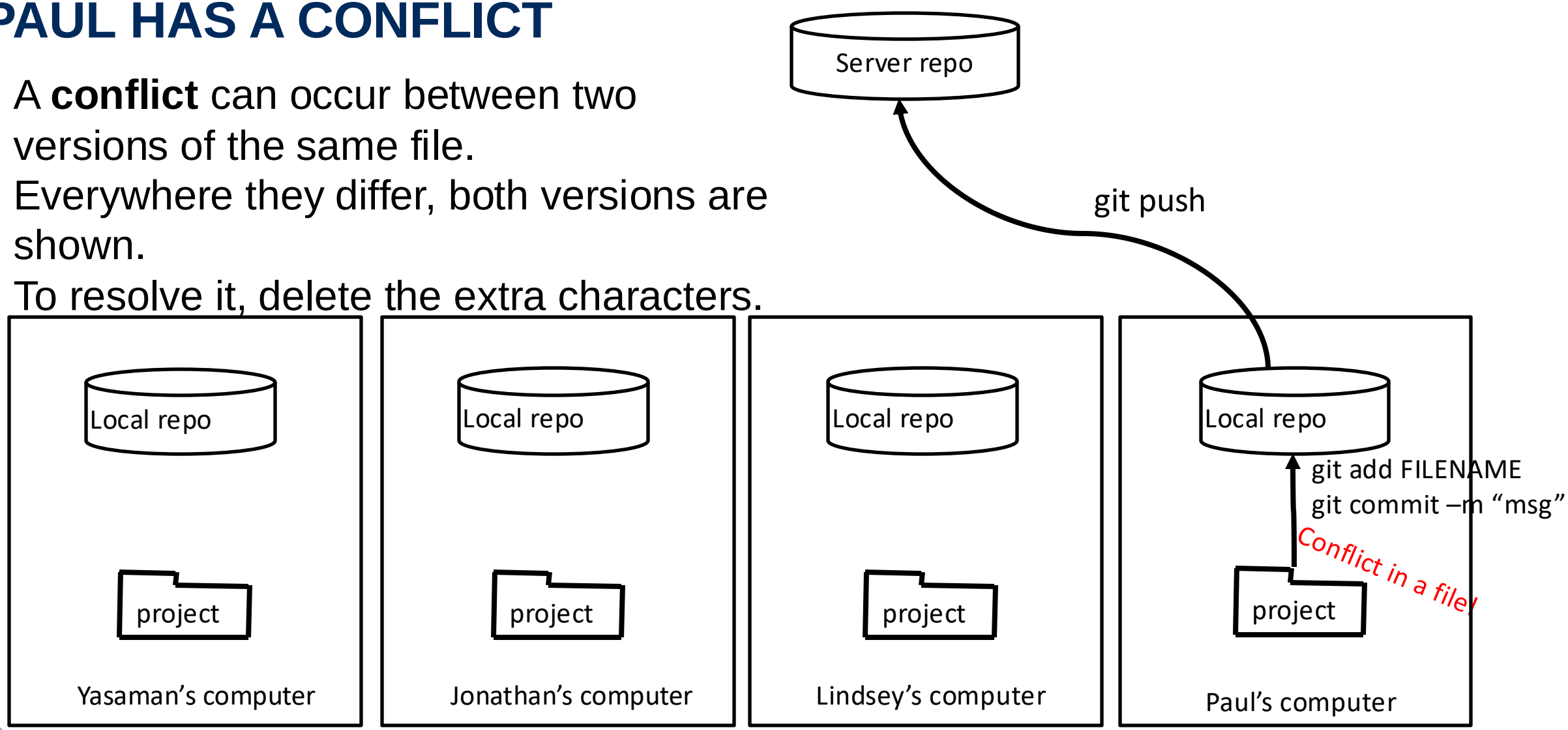


DISTRIBUTED VERSION CONTROL SYSTEMS: JONATHAN WORKS ON HIS CURRENT FEATURE



DISTRIBUTED VERSION CONTROL SYSTEMS: PAUL HAS A CONFLICT

- A **conflict** can occur between two versions of the same file.
- Everywhere they differ, both versions are shown.
- To resolve it, delete the extra characters.



BRANCH AND MERGE WORKFLOW

- To avoid having to constantly resolve conflicts, developers often create a **branch**, where they work on a new feature, then submit a **pull request**.
- One or more members of the team review the pull request, resolve any conflicts (as before), and the branch is merged back in.
- You'll get lots of practice with this in your project this term and this workflow will be more formally covered in a future lab.
- If you are interested, you can also read about other workflows here:

  https://www.atlassian.com/git/tutorials/comparing-workflows

https://xkcd.com/1597/

EXTRA GIT RESOURCES (SUPPLEMENTAL READING)

What is version control and overview of commands:

https://betterexplained.com/articles/a-visual-guide-to-version-control/

A quick guide to getting started using git:

https://towardsdatascience.com/a-quick-primer-to-version-control-using-git-3fbdbb123262

An interactive tutorial with visuals to help you learn git:

https://learngitbranching.js.org