

CS 520

Theory and Practice of Software Engineering
Fall 2026

Version Control

1

Working in Teams



TEAMWORK

Large ambitious goals usually require that people work together.

2

Our Goal

- Learn about different kinds of VCS
- Overview the basics of git
- Touch some intermediate git topics
- Clear up common points of confusion
 - Branch vs. Fork?
 - Merge vs. Pull Request?
 - Pull vs. Fetch?
 - Fork vs. Clone?

3

What Is VCS?

4

What is a Version Control System?

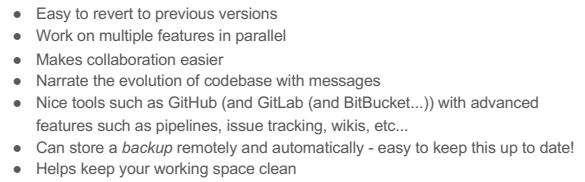
A *Version Control System* (VCS) records changes to a file set over time, making it easy to review or revert to specific versions later

5

Why Use VCS?

6

Types of VCS



8



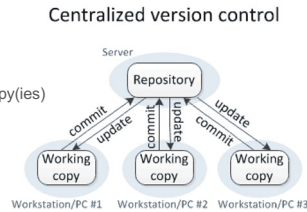
10

Types of VCS

12

Types of VCS -- Centralized

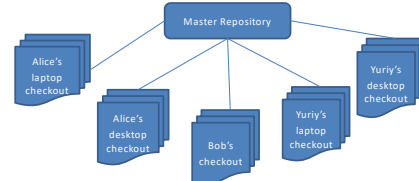
- There exists a single "central" copy of the project
 - All developers commit to this single copy
- Each developer has local working copy(ies)
 - As soon as they commit, the central repo reflects the changes



13

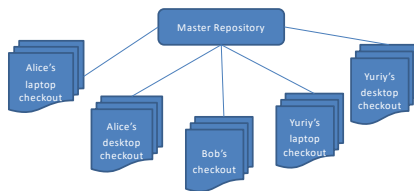
Centralized version control

- (old model)
- Examples: Concurrent Versions System (CVS)
Subversion (SVN)



14

Doing work



- I **update** my checkout (working copy)
- I **edit**
- I **update** my checkout again
- I **merge** changes if necessary
- I **commit** my changes to the Master

15

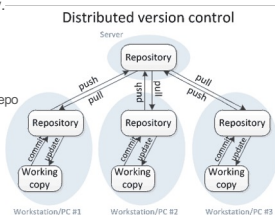
Problems with centralized VC

- What if I don't have a network connection?
- What if I am implementing a big change?
- What if I want to explore project history later?

16

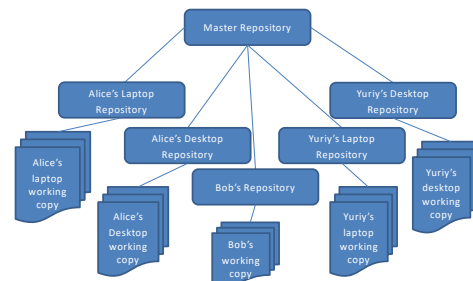
Types of VCS -- Distributed

- Each developer has their own repository.
 - Created by the developer, or
 - Cloned from an existing (remote) repository
- Developers work on their own repos
 - They can commit, branch, etc.
 - Activity is local unless it is *pushed* to remote repo
 - Remote activity is not seen until dev *fetches* from the remote repo
- Examples: Mercurial (hg), git



17

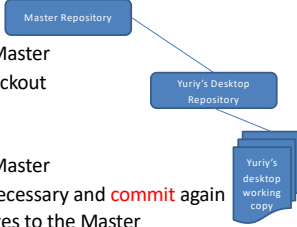
Distributed version control model



18

Doing work

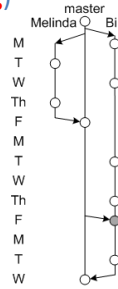
- I **pull** from the Master
- I **update** my checkout
- I **edit**
- I **commit**
- I **pull** from the Master
- I **merge** tips if necessary and **commit** again
- I **push** my changes to the Master



19

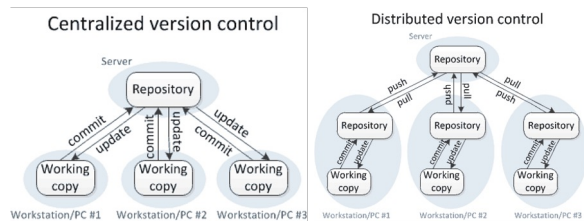
History view (log)

- Bill and Melinda work at the same time
- At the end, all repositories have the same, rich history



20

Pros and Cons of Centralized VCS



21



22

A Motivating Example: What is this git command?

NAME
git-_____ - _____ file contents to the index

SYNOPSIS
git _____ [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]

DESCRIPTION
This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically _____s the current content of existing paths as a whole, but with some options it can also be used to _____ content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

23

A Motivating Example: What is this git command?

NAME
git-add - Adds file contents to the index

SYNOPSIS
git add [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]

DESCRIPTION
This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

24

A Motivating Example: What is this git command?

NAME
git-_____ - Switch branches or restore working tree files

SYNOPSIS
git _____ [-q] [-f] [-m] [<branch>]

DESCRIPTION
Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, git _____ will also update HEAD to set the specified branch as the current branch.

25

A Motivating Example: What is this git command?

NAME
git-checkout - Switch branches or restore working tree files

SYNOPSIS
git checkout [-q] [-f] [-m] [<branch>]

DESCRIPTION
Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, git checkout will also update HEAD to set the specified branch as the current branch.

26

A Motivating Example: What is this git command?

NAME
git-_____ - Forward-port local commits to the updated upstream head

SYNOPSIS
git _____ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]]
git _____ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]
git _____ --continue | --skip | --abort | --edit-todo

DESCRIPTION
If <branch> is specified, git _____ will perform an automatic git checkout <branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote and branch.<name>.merge options will be used (see git-config[1] for details) and the --fork-point option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the _____ will abort.

27

A Motivating Example: What is this git command?

NAME
git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo

DESCRIPTION
If <branch> is specified, git rebase will perform an automatic git checkout <branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote and branch.<name>.merge options will be used (see git-config[1] for details) and the --fork-point option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the rebase will abort.

28

Our goal with git

Be able to understand the git man-pages

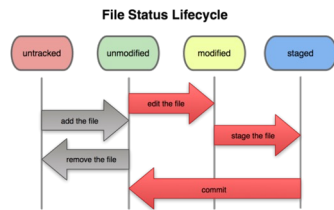
29

Git Basics
How Git Works

30

Git Basics -- Tracked vs. Untracked

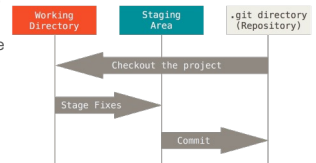
- **untracked file** - a file not currently under version control
- **tracked file** - a file that is under version control



31

Git Basics -- Three Main Stages

1. **Committed:** Everything in the file is currently in the database
2. **Modified:** Changed the file but have not committed to the database
3. **Staged:** Marked the file for addition to the database in the next commit



Note that all of the above pertain to *tracked* files.

32

Git Basics -- Creating Repositories

Initializing a repository

- `git init` - Create an empty git repository or reinitialize an existing one
 - `--bare` - create a bare repository
 - `[directory]` - `git init` is run inside the provided directory
- `git init` creates a `.git` folder in the directory chosen

33

Git Basics -- Creating Repositories

Cloning a Repository

- `git clone` - Clone a repository into a new directory
 - `--depth <depth>` - Create a shallow clone with a history truncated to `<depth>` commits
 - `--branch <name>` - Point local HEAD to specific branch (more on HEAD in a bit!)
 - `--origin <name>` - Use `<name>` to keep track of remote repo instead of 'origin'
- Basically, clone just:
 - calls `init`
 - points some meta variables at an existing repository
 - copies the data to the new repo

34

.git/

- What's in it?
 - `branches/`:
 - `COMMIT_EDITMSG`: most recent commit message
 - `config`: configure your git repository
 - `description`: only used by the GitWeb program ([source](#))
 - `hooks/`: This contains client or server-side hook scripts ([more info](#))
 - `index`: The "staging area"
 - `info/`: keeps a global `exclude` file for your project
 - `logs/`: keeps track of history of HEAD and refs
 - `objects/`: where the actual content is stored
 - `refs/`: keeps track of refs and tags

35

.git/

- What's in it?
 - `branches/`:
 - `COMMIT_EDITMSG`: most recent commit message
 - `config`: configure your git repository
 - `description`: only used by the GitWeb program ([source](#))
 - `hooks/`: This contains client or server-side hook scripts ([more info](#))
 - `index`: The "staging area"
 - `info/`: keeps a global `exclude` file for your project
 - `logs/`: keeps track of history of HEAD and refs
 - `objects/`: where the actual content is stored in a database
 - `refs/`: keeps track of refs and tags

36

Git Vocabulary

37

Git Vocabulary

- index: staging area (located .git/index)
- content
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream

38

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream

39

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream

40

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged
- commit
- ref
- branch
- HEAD
- upstream

41

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit
- ref
- branch
- HEAD
- upstream

42

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref
 - branch
 - HEAD
 - upstream

43

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
 - branch
 - HEAD
 - upstream

44

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD
- upstream

45

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- upstream

46

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- upstream: complicated, basically "backwards in time" (but not quite!)

47

Git Basics

Working Locally

48

Git Basics: *Changing Content* -- `git add`

`git add` does *two things*:

1. given an untracked file it will
 - a. start tracking it
 - b. update `.git/index` using the current *content* found in the working tree to prep the content for the next commit (i.e., the content is **staged**)
2. given a modified unstaged file it will
 - a. stage its *contents* for commit

`--patch, -p`: start an interactive staging session that lets you choose portions of a file to add to the next commit.

49

Git Basics: *Changing Content* -- `git commit`

`git commit` updates the Git database with staged content in `.git/index`

- Note that staged files can have *unstaged changes*
- By default this will open an editor for you to enter a commit message

`--message=<msg>, -m <msg>`: Add `<msg>` as the commit message. If multiple messages are given, concatenate as separate paragraphs

`--patch, -p`: Use the interactive patch selection interface to choose which changes to commit (similar to `git add -p`)

50

Git Basics *Making Queries*

51

Git Basics: *Making Queries* -- `git status`

`git status` shows the working tree status. This command displays:

- paths that have differences between the index file and the current HEAD
- paths that have differences between the working tree and the index file
- paths in the working tree that are not tracked by git

`--short, -s`: Give the output in the short-format

`--ignored`: Show ignored files

52

Git Basics: *Making Queries* -- `git log`

`git log` inspects commit history with multiple display options

- `git log` is basically a wrapper around `git rev-list` and `git diff`-* (don't worry about these - I sure don't!)

Some Examples

```
git log
git log --graph
git log --graph --all
git log --graph --all --oneline
```

53

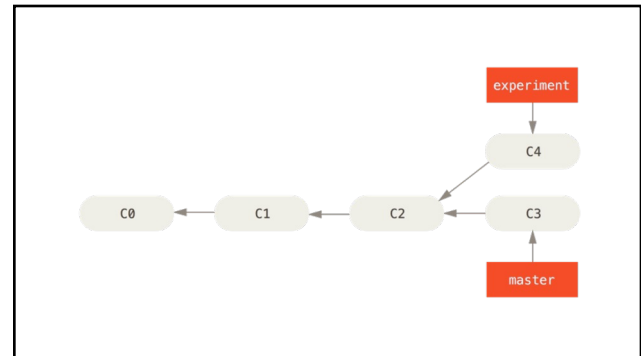
Git Basics: *Making Queries* -- `git log`

...Some Examples

```
git log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold
cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset) %C(bold
cyan)(committed: %cD)%C(reset) %C(auto)%d%C(reset)%n''
%C(white)%s%C(reset)%n''           %C(dim white)- %an <%ae>
%C(reset) %C(dim white)(committer: %cn <%ce>)%C(reset)'
```

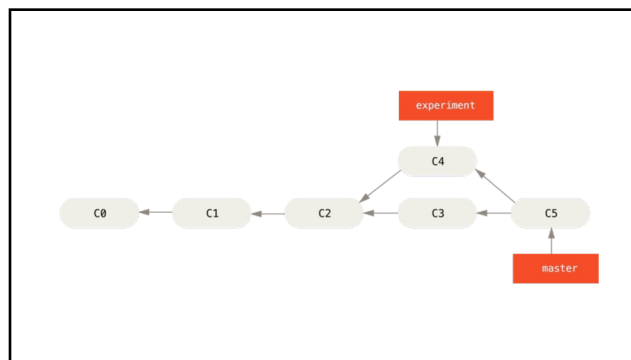
54

Git Merge



55

56



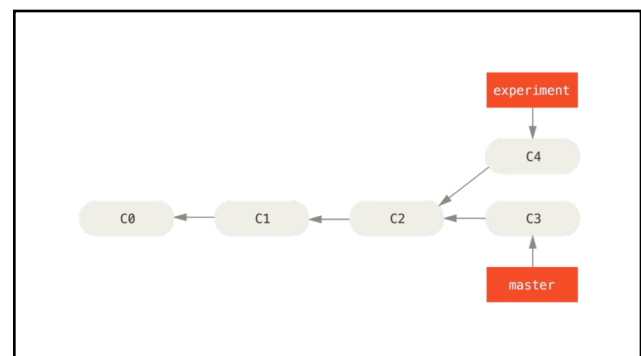
57

Git Rebase

58

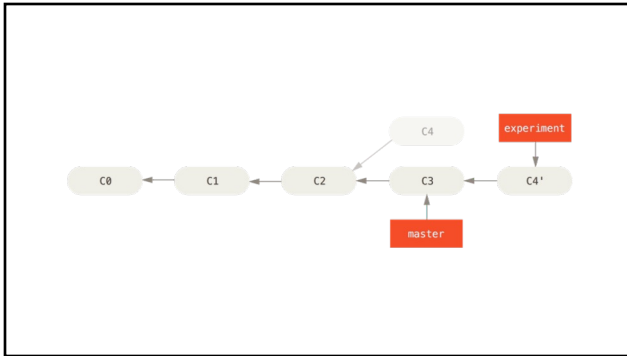
Changing Commit History with Rebase

- Git rebase lets us change our commit history
- rebase is a powerful tool, but we will only scratch the surface

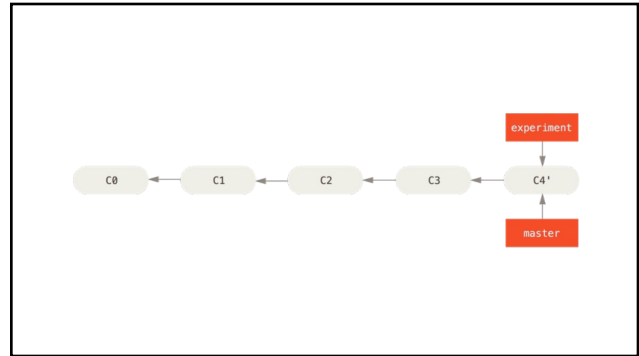


59

60



61

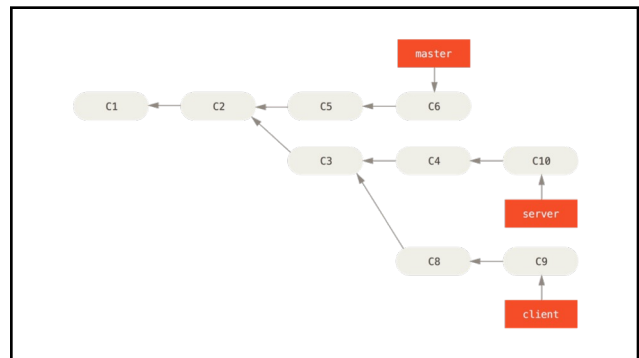


62

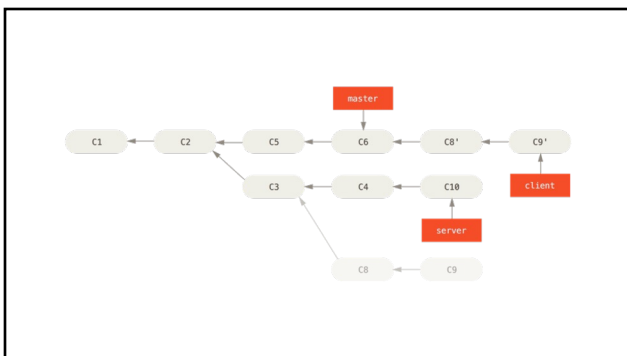
Changing Commit History with Rebase

- Git rebase --onto gives us a bit more power

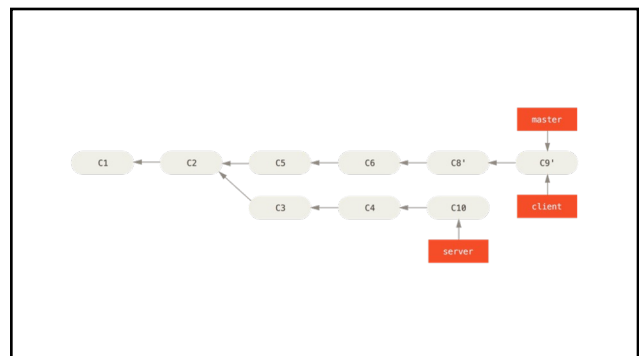
63



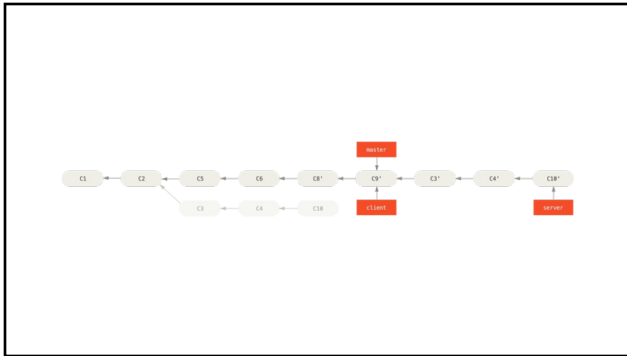
64



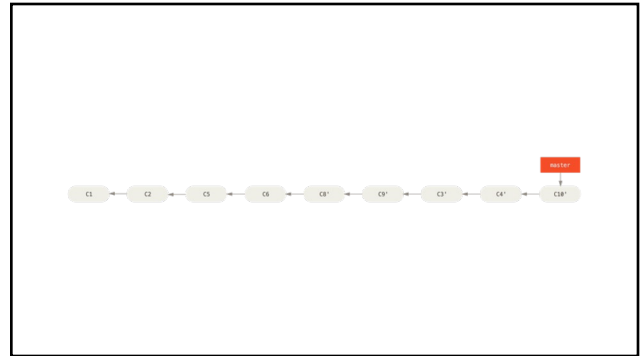
65



66



67



68

Why use Git Rebase?

- Clean history
- If you are issuing a pull request it's nicer for folks looking over it to decide to include it
- Clean "narrative"
- Branches can look messy

69

Points of Confusion

70

Fork vs. Clone

71

Fork vs. Clone

Fork

Fork is NOT A GIT CONCEPT

- it was invented by GitHub
- Fork stores extra information and makes pull requests possible

Clone

Clone IS A GIT CONCEPT

- clone extends init
- exists independent of github

72

Branch vs. Clone

73

Branch vs. Clone

Branch

Branch creates a ref

Clone

Clone creates a new repository

74

Pull vs. Fetch

75

Pull vs. Fetch

Fetch

- Take target branch from a remote repository and store it in `.git/refs/remotes/`
- NOT integrated/merged with local branches!!!!

Pull

- Fetches remote branch and merges with local branch or repository

76