CS 520: VCS and Git

Intermediate Topics

Ben Kushigian

https://people.cs.umass.edu/~rjust/courses/2017Fall/CS520/2017_09_19.zip

Our Goal

Our Goal (Overture)

- Overview the basics of Git w/ an eye towards its architecture and design
- Touch some intermediate Git topics
- Clear up common points of confusion
  - Branch vs Fork?
  - Merge vs Pull Request?
  - Pull vs Fetch?
  - Fork vs Clone?

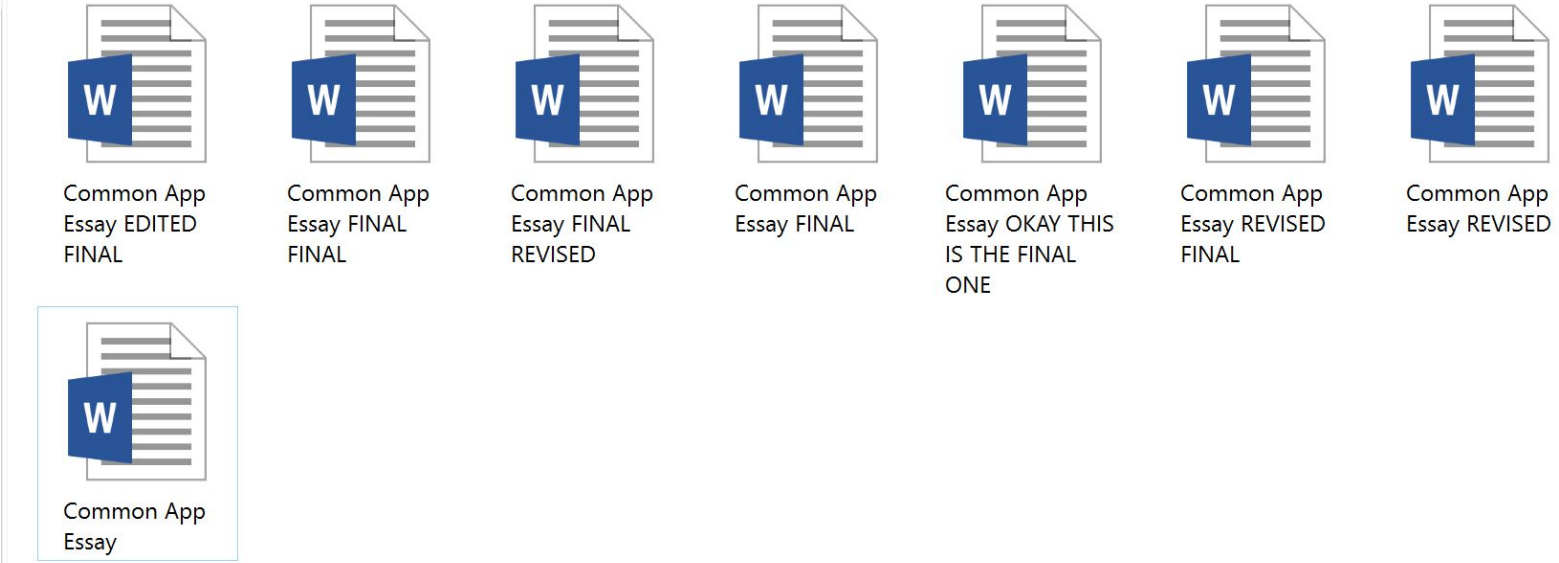
What Is VCS?

What are Version Control Systems

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

Why Use VCS?

Why Use Version Control?



Why Use Version Control?

- Easy to revert to previous versions
- Work on multiple features in parallel
- Makes collaboration easier
- Narrate the evolution of codebase with messages
- Nice tools such as GitHub (and GitLab (and BitBucket...)) with advanced features such as pipelines, issue tracking, wikis, etc...
- Can store a *backup* remotely and automatically - easy to keep this up to date!
- Helps keep your working space clean

Who Uses VCS?

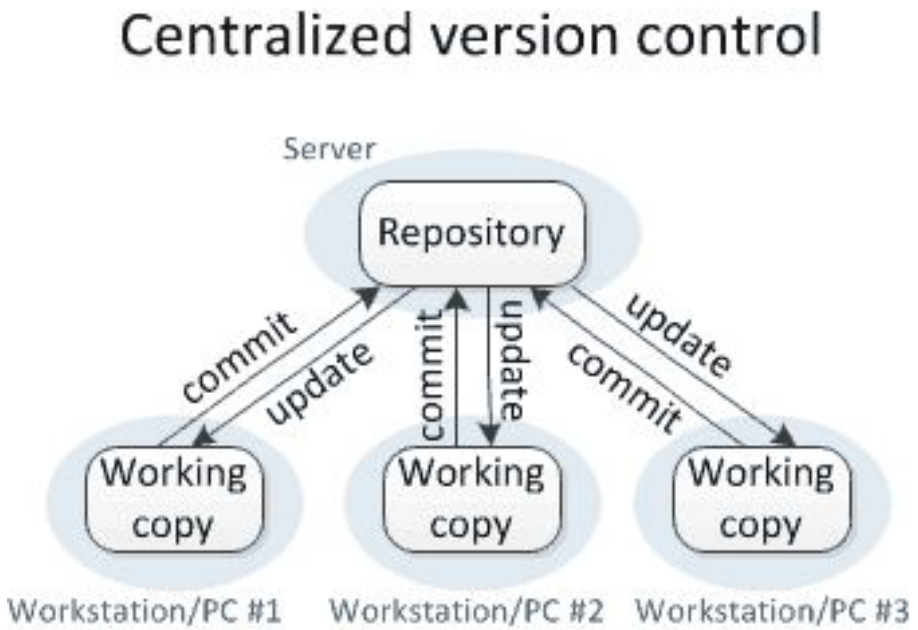
Who Uses Version Control?

- Programmers (obviously)
- Applications (Microsoft Word, Google Docs, ...)
- Organizations - VCS can be used to sync lots of data (not just code)

Types of VCS

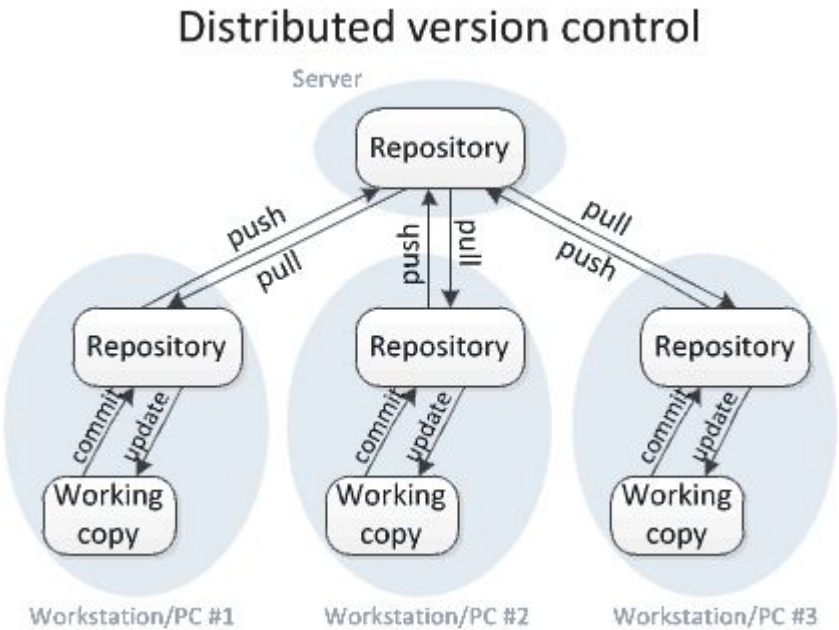
Types of VCS -- Centralized

- There exists a single "central" copy of the project
  - All developers commit to this single copy
- Each developer has a working copy locally
  - As soon as they commit they update the central repository

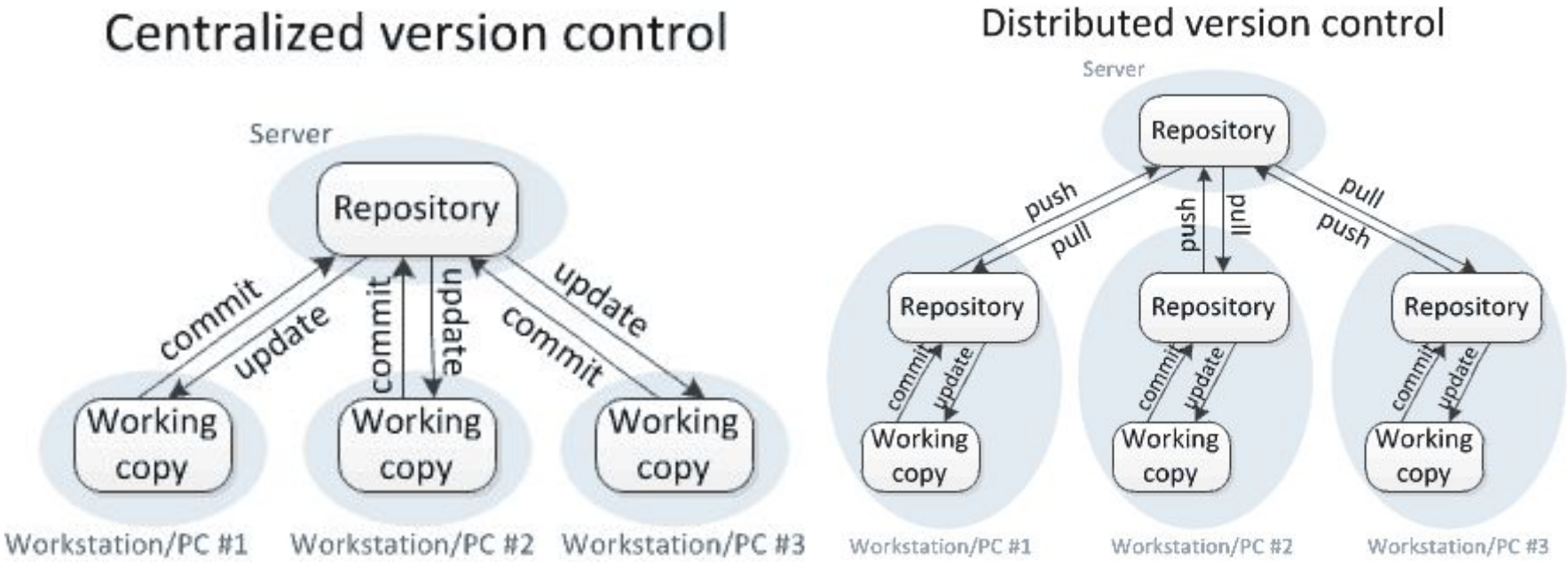


Types of VCS -- Distributed

- Each developer has their own repository. This can be
  - Created by the developer, or
  - Cloned from an existing (*remote*) repository
- Devs work on their own repos
  - They can commit, branch, etc as they see fit
  - 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



Pros and Cons of Centralized VCS




git

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.

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.

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.

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.

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.

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.
```

Our Goal (Reprise)

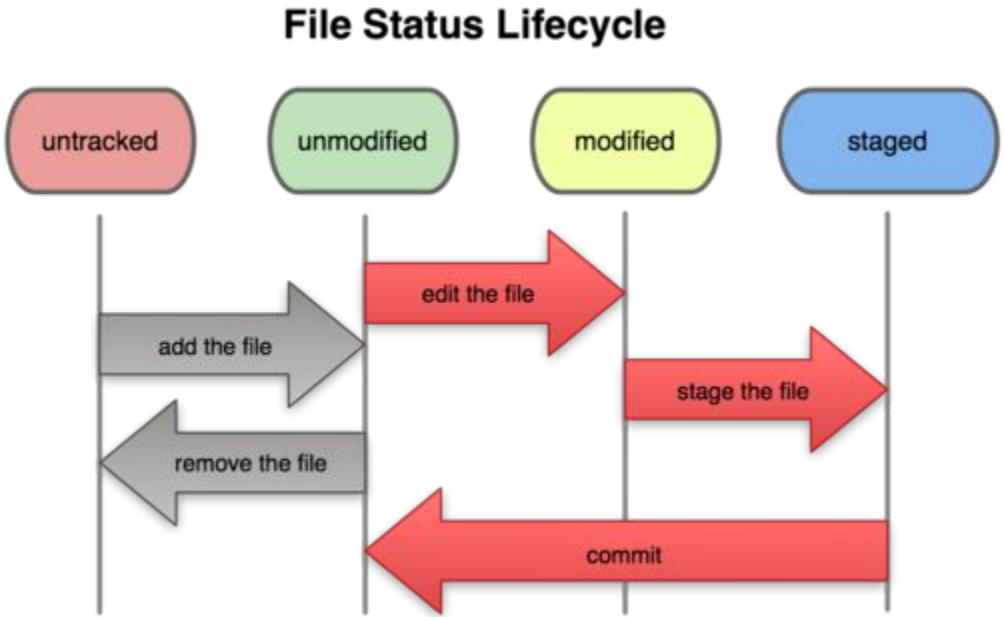
Be able to understand the git man-pages

Git Basics

How Git Works

Git Basics -- Tracked vs Untracked

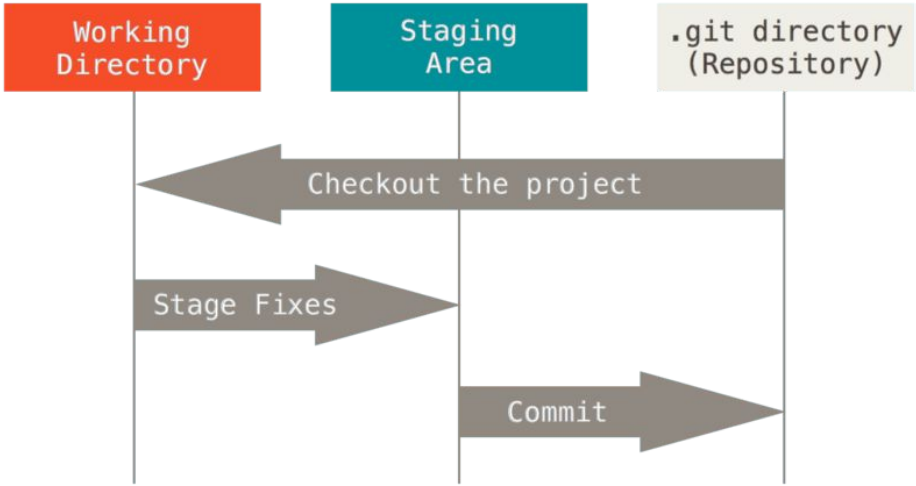
- **untracked file** - a file not currently under version control
- **tracked file** - a file that *is* under version control



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.



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

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

.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

.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*

Git Vocabulary

Git Vocabulary

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

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

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

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

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

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

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

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

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

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!)

Git Basics

Working Locally

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

A *super-duper* useful argument:

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

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`)

Git Basics

Making Queries

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

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

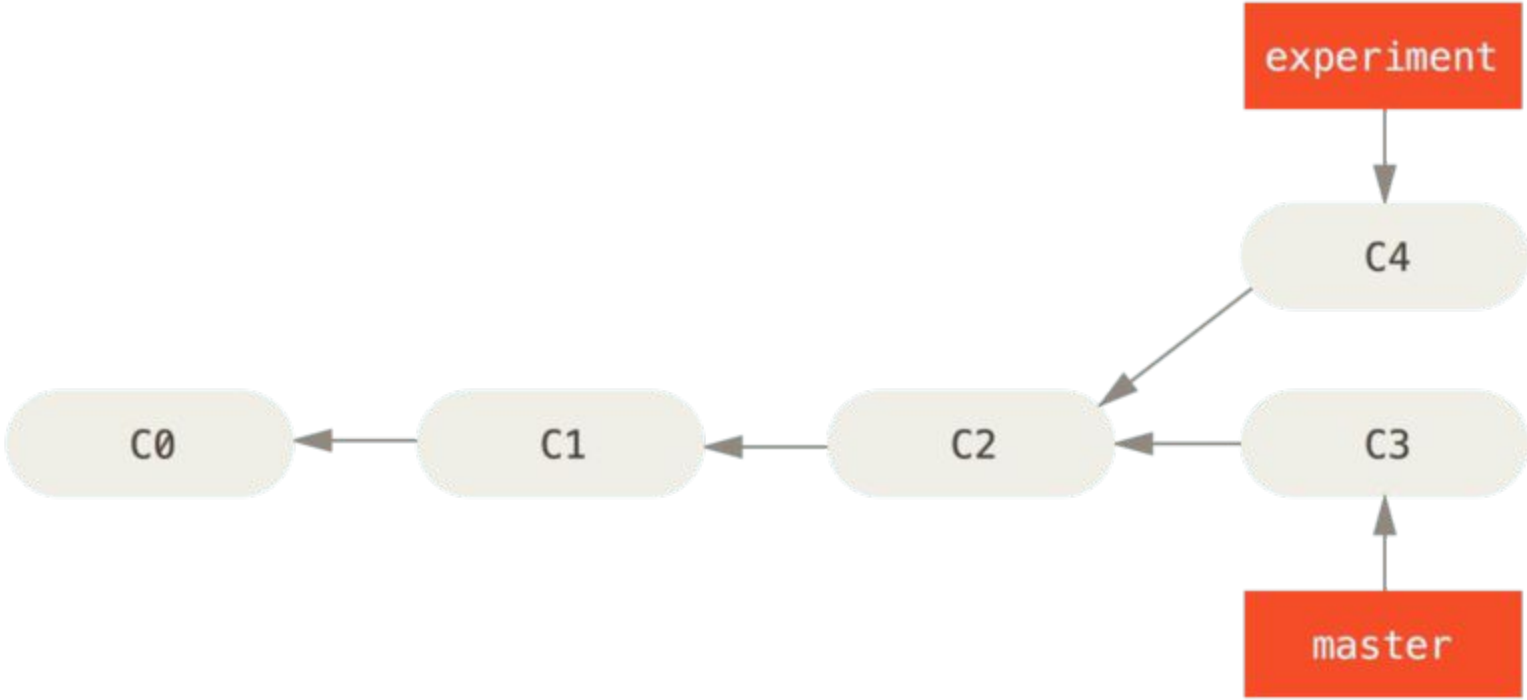
git log
git log --graph
git log --graph --all
git log --graph --all --oneline

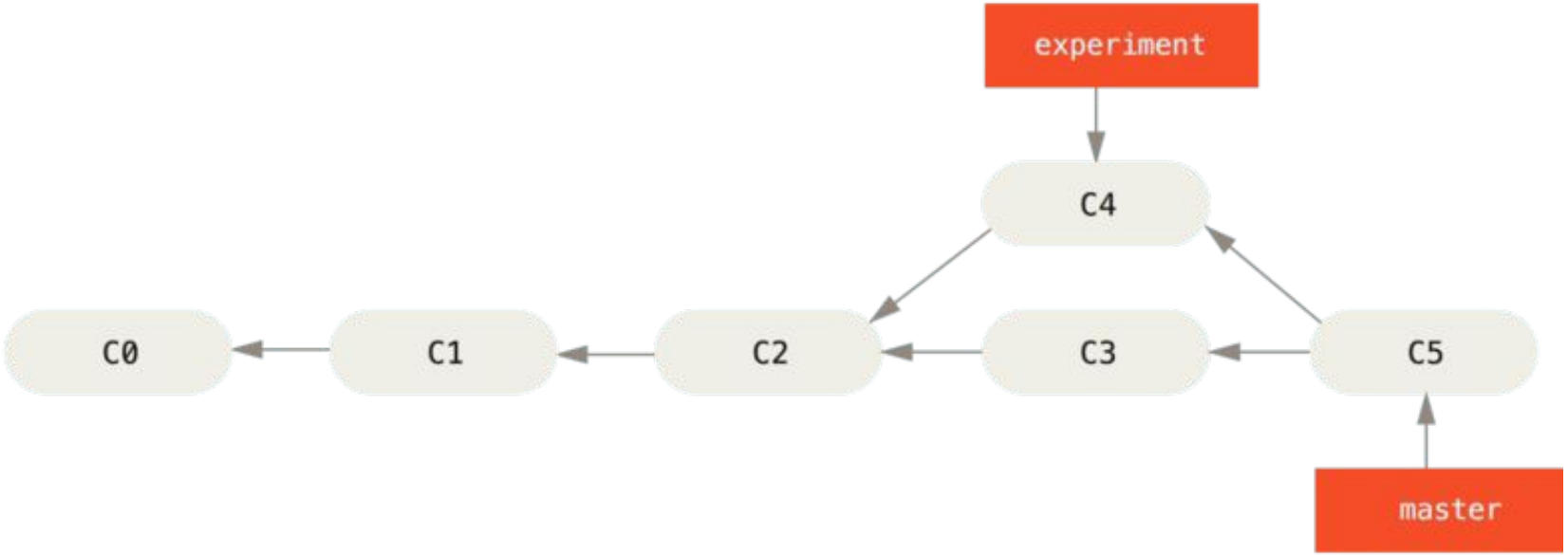
Git Basics: *Making Queries* -- `git log`

...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)'
```

Git Merge

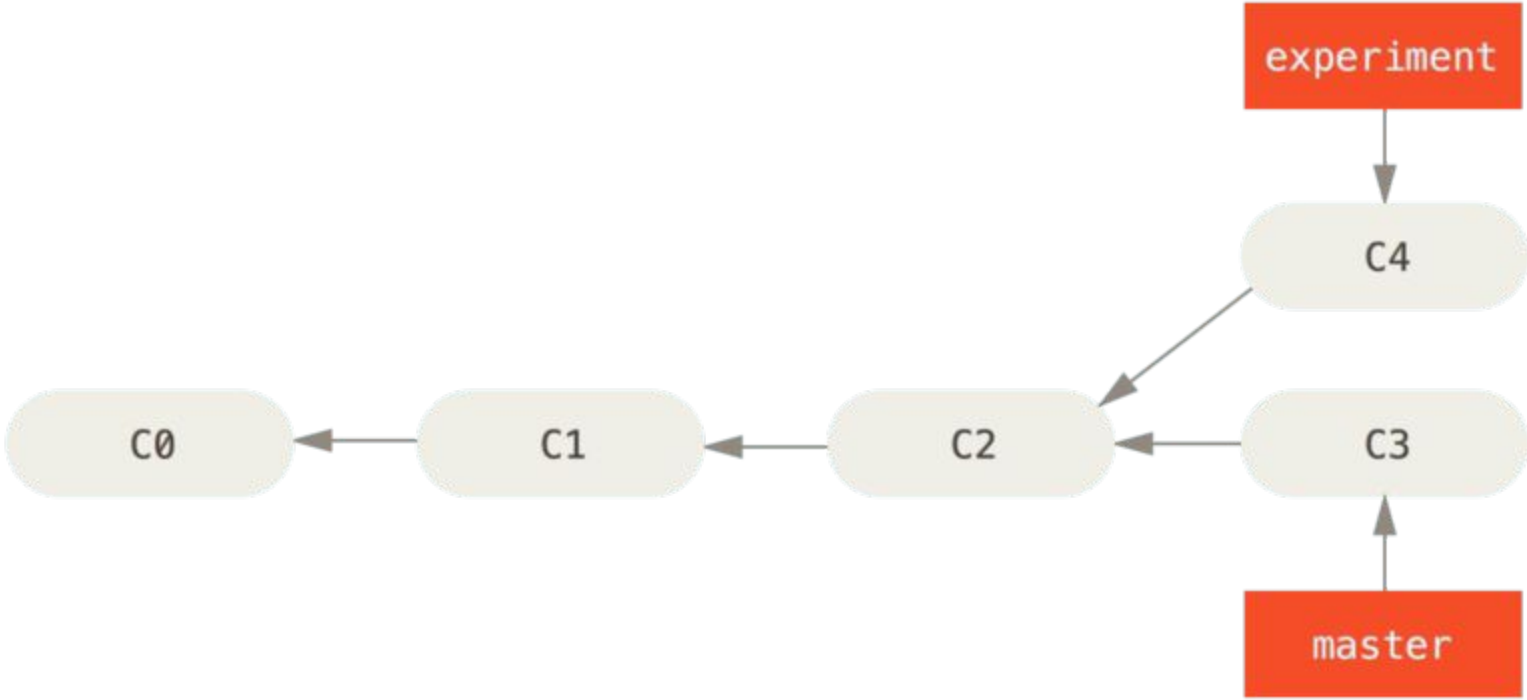

experiment
C4
C0
C1
C2
C3
master

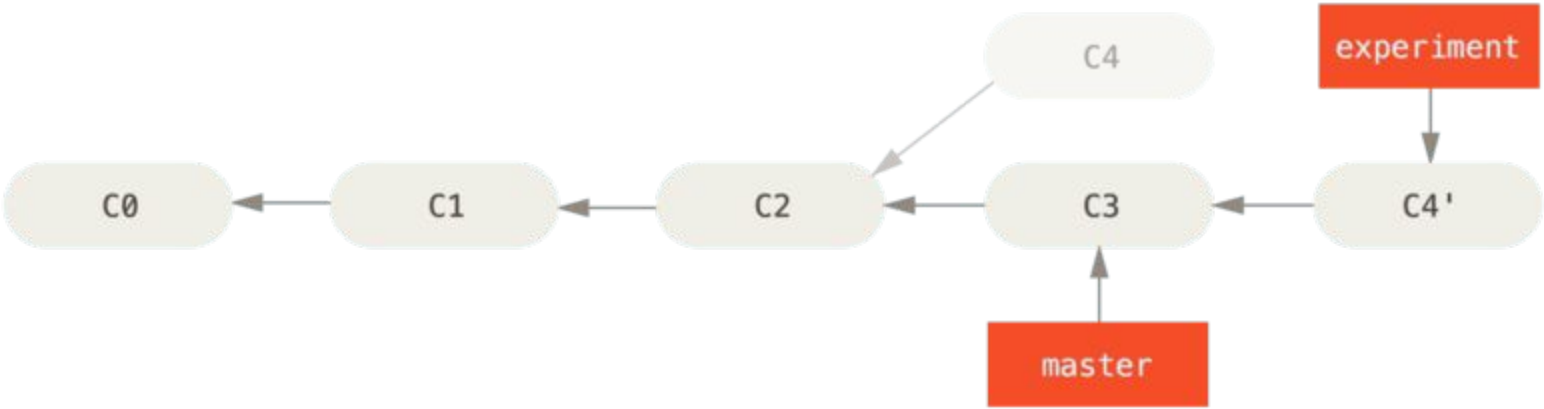

experiment
C4
C0
C1
C2
C3
C5
master

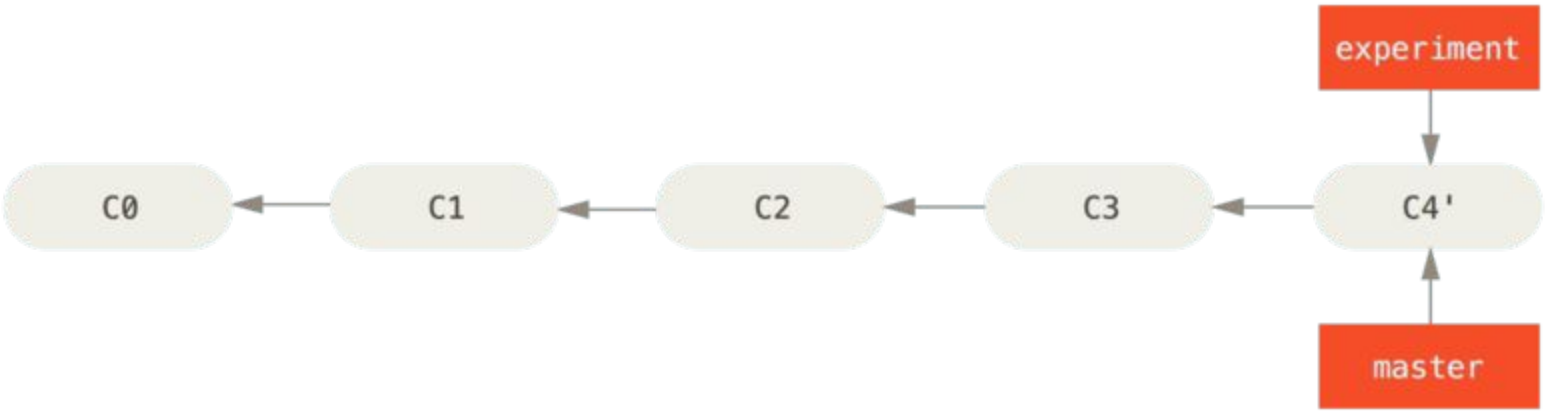
Git Rebase

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

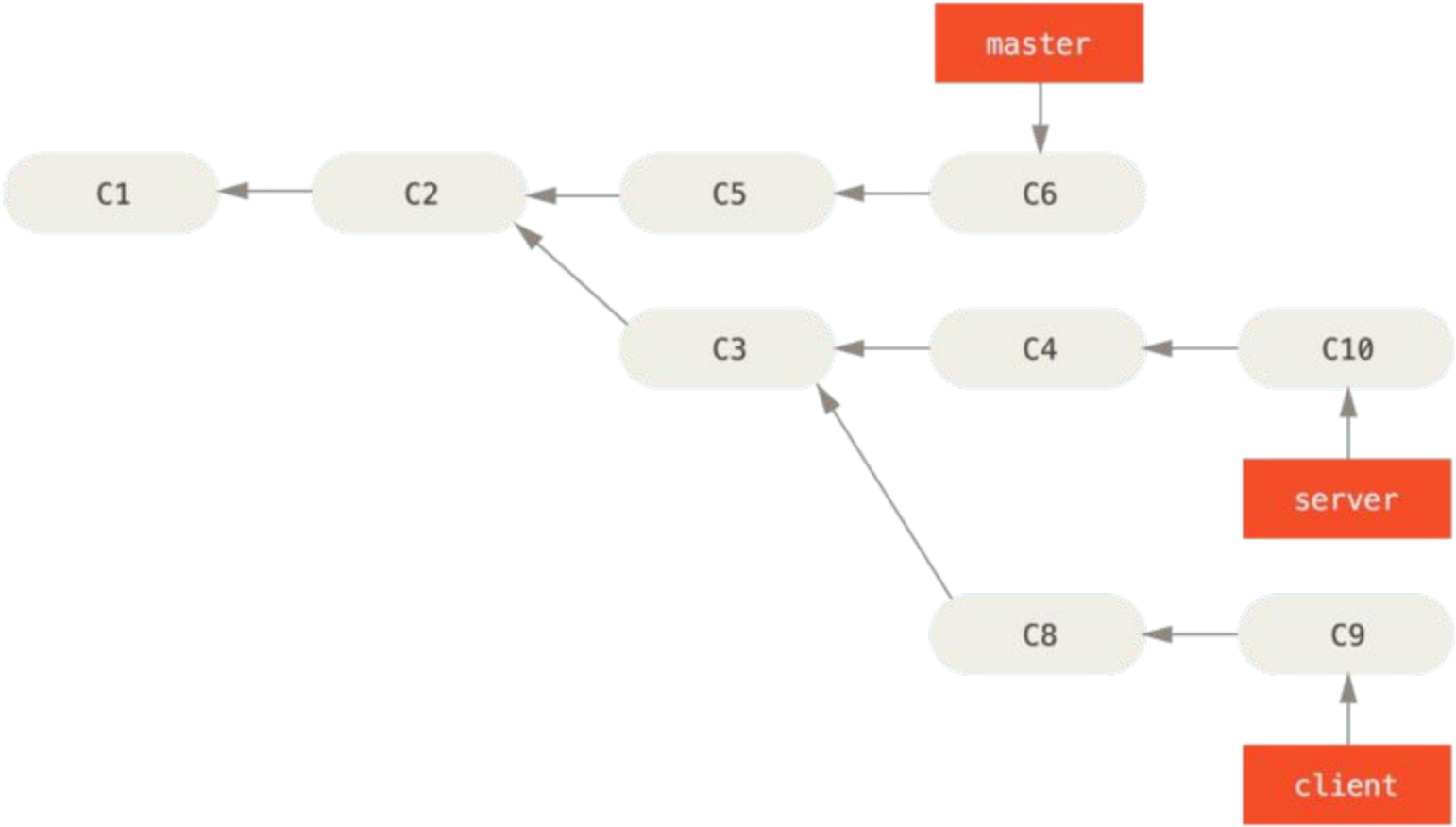

experiment
C4
C0
C1
C2
C3
master

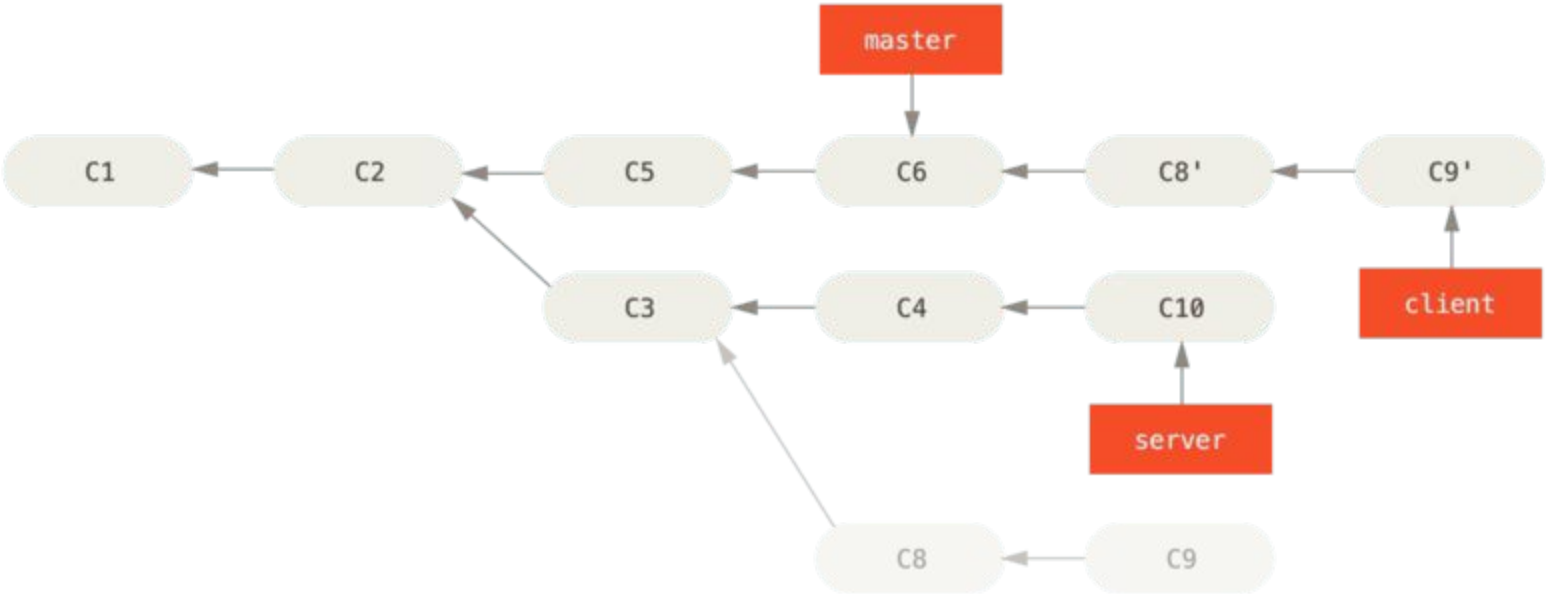

C4
experiment
C0
C1
C2
C3
C4'
master

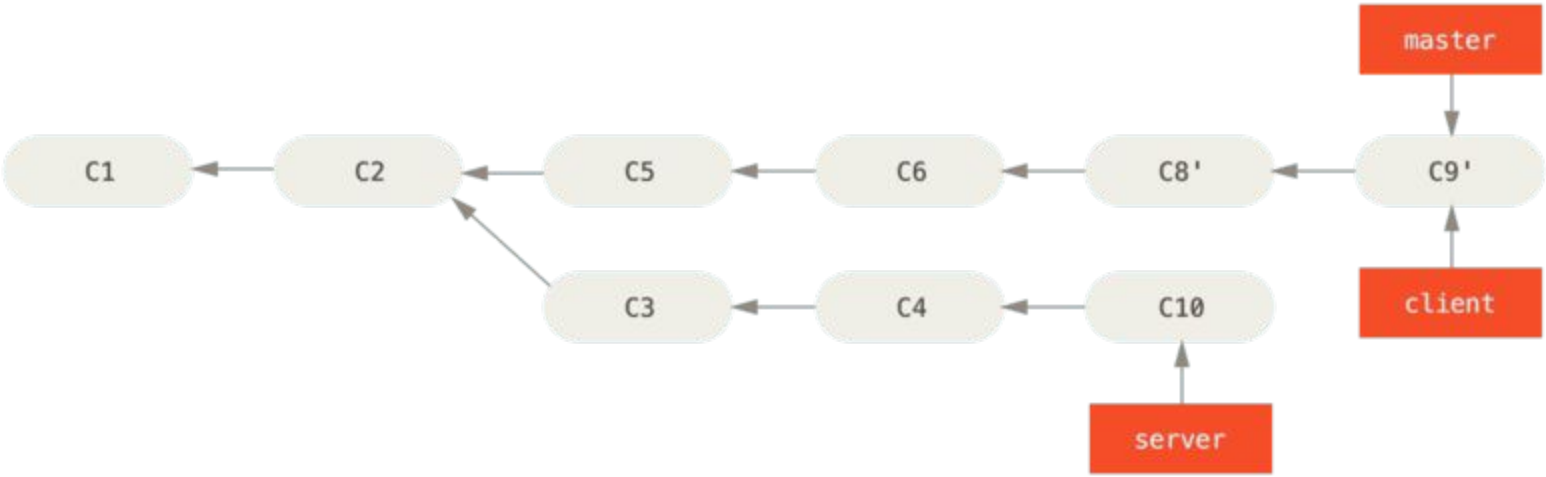

experiment
C0
C1
C2
C3
C4'
master

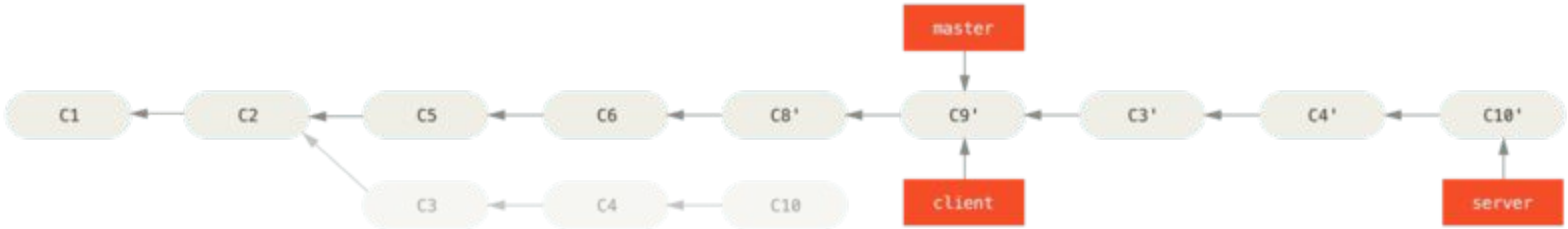
Changing Commit History with Rebase

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


master
C1
C2
C5
C6
C3
C4
C10
server
C8
C9
client


master
C1
C2
C5
C6
C8'
C9'
C3
C4
C10
client
server
C8
C9


master
C1
C2
C5
C6
C8'
C9'
C3
C4
C10
client
server


master
C1
C2
C5
C6
C8'
C9'
C3'
C4'
C10'
C3
C4
C10
client
server


master
C1
C2
C5
C6
C8'
C9'
C3'
C4'
C10'

Why use Git Rebase?

Points of Confusion

Fork vs Clone

-
-

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

Branch vs Clone

Branch vs Clone

Branch

Branch creates a ref

Clone

Clone creates a new repository

Pull vs Fetch

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
- NOTE: This can be DANGEROUS
- Always use git fetch. Pretend like git pull doesn't exist.
- Like, really. Just don't use it
- I can tell you're thinking about using it
    - a) don't
    - b) no, seriously, don't
        - It's a bad idea
        - You will regret it
            - Even if all your friend are doing it. You should really get better friends

The End