Version Control Workshop 2

Frederick Brunn
Project Manager

Shilling IEEE events

- PCB Design Workshop with Professor Westerfeld
  - Melville Library Room **E4315**
  - **7:00 PM tonight (right after this workshop)**
  - Learn more industry-important skills that no ECE courses currently teach
  - This will be an introduction to Eagle Software. All you'll need for this workshop is a laptop that can run Windows, Mac, or Linux. We'll be using Eagle CAD, which you can download and install during or before the workshop.
  - http://www.cadsoftusa.com/download-eagle/freeware/

In this workshop we will cover...

- .gitignore
- Comparing commits
- Stashing changes
- Branches
  - Branch management
  - Merges
  - Rebasing

.gitignore

- .gitignore is a text file optionally placed in the root of a Git repository that describes files and folders to be ignored when adding new files to be tracked in a commit.
- Takes wildcards
- Does NOT cancel out folders and files that are already tracked. To untrack those, use `git rm --cached` once you have your .gitignore in place.
- https://github.com/github/gitignore/blob/master/C.gitignore

.gitignore

- .gitignore is a text file optionally placed in the root of a Git repository that describes files and folders to be ignored when adding new files to be tracked in a commit.
- Takes wildcards
- Does NOT cancel out folders and files that are already tracked. To untrack those, use `git rm --cached` once you have your .gitignore in place.
- https://github.com/github/gitignore/blob/master/C.gitignore

.gitignore

- A good resource for pre-made .gitignore files for each language / IDE / etc. can be found at the following repository hosted on github
- https://github.com/github/gitignore

Comparing commits

- `git diff <commit 1> <commit 2>` can be used to find all the differences between two commits.

- This opens up a text viewer that scrolls through all changes in all files between the two commits.

Comparing commits

- `git diff` is a *very* powerful tool. It can be used for all sorts of comparisons given various arguments, including
  - one file between same or different commits, in same or different branches
    - `git diff <branch 1>:<filepath> <branch 2>:<filepath>`
      - Same file, two different branches
    - `git diff <branch>~<number of commits back>:<filepath> <branch>~<number of commits back>:<filepath>`
      - Same file, same branch, different numbers of commits back
    - `git diff <branch>:<filepath> <filepath>`
      - Same file, same or different branch, compared to working copy

Comparing commits

- `git diff` is a *very* powerful tool. It can be used for all sorts of comparisons given various arguments, including
  - entire directories
    - as the before commands, except replace the filepath with the path to a directory
  - staged working copy files vs. their status at last commit
    - `git diff --cached`

Comparing commits

- If alternative difftools are installed, `git difftool` is usually set to run the tool
- Use `git difftool` with your regular diff arguments
- GUI-based difftools allow you to very helpfully see the changes made across a file in side-by-side or inline windows
- `git difftool --tool-help` to see what tools are available on your machine
- `git difftool -tool=<tool>` to launch a specific tool

Comparing commits

- Recommend tools:
  - vimdiff is available on most machines with git shell, but requires vim commands to use
  - kdiff3 is a good Windows GUI difftool
- Configure the following git config properties to edit the default difftools
  - diff.tool
    - The default diff tool to use.
  - diff.guitool
    - The default diff tool to use when --gui is specified.

Comparing commits

- Recommend tools:
  - vimdiff is available on most machines with git shell, but requires vim commands to use
  - kdiff3 is a good Windows GUI difftool
- Configure the following git config properties to edit the default difftools
  - diff.tool
    - The default diff tool to use.
  - diff.guitool
    - The default diff tool to use when --gui is specified.

Stashing changes

- Sometimes, you want to save your current changes without committing them anywhere.
  - When you are in the middle of something, you learn that there are upstream changes that are possibly relevant to what you are doing. When your local changes do not conflict with the changes in the upstream, a simple `git pull` will let you move forward.
  - However, there are cases in which your local changes do conflict with the upstream changes, and `git pull` refuses to overwrite your changes.

Stashing changes

- Sometimes, you want to save your current changes without committing them anywhere.
  - In such a case, you can stash your changes away, perform a pull, and then unstash

```
git stash
git pull
git pop
```

Stashing changes

- Sometimes, you want to save your current changes without committing them anywhere.
  - In such a case, you can stash your changes away, perform a pull, and then unstash

```
git stash
git pull
git pop
```

Stashing changes

`git stash`

- Git stash pushes all of the current changes to an internal stack structure and resets your working copy

`git pop`

- Git pop retrieves the most current stashed change of the stack and applies it to the working copy

- Either can be used with arguments to specify stashing / popping individual files

Branches

- A branch represents an independent line of development.
- Each branch has its own:
  - working directory
  - staging area
  - project history.
- New commits are recorded in the history for the current branch, which results in a fork in the history of the project.
- When you want to add a new feature or fix a bug, you create a new branch to contain all of your changes separately from the master / from other branches

Branch management

- To create a new branch, use `git branch <branch name>`
  - This creates a new branch with the name you specify, but does not switch to it
  - You can use `git branch -b <branch name>` to do this in a single line
- To switch between branches, checkout the other branch
  - `git checkout <branch name>`
  - As with all uses of checkout, this overwrites your working copy with any differences

Branch management

- To rename branches, use `git branch -m <new name>` while checked into the branch to rename
- To delete branches, use `git branch -d <branch name>`
- Branches are metadata- new branches from a remote server are fetched with `git fetch` or `git pull`, and new local branches are pushed to the remote with `git push`

Branch merging

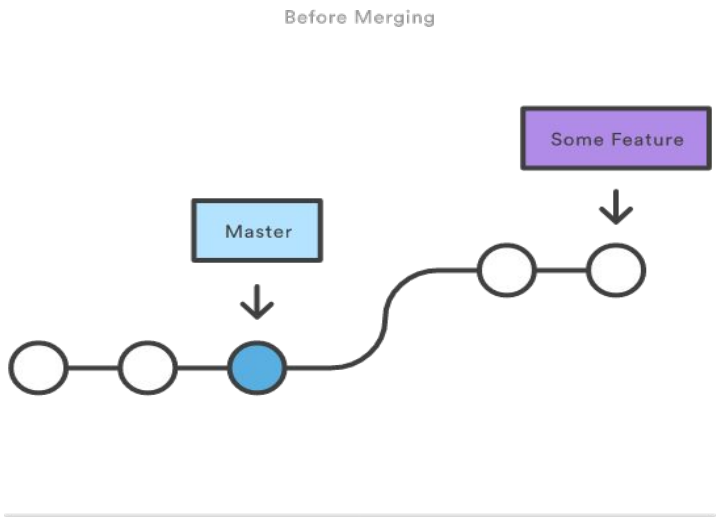
- Once your new feature is complete or your new bugs are fixed, it is time to merge your feature / bugfix branch into the master branch again
- In multi-programmer projects, you might not have permission to automatically merge your changes back in
  - In this case, you must create a **pull request** through the website's interface, stating the nature of your branch and why it should be merged into master
  - Once they approve your PR, the project administrator (s) merge your branch into master

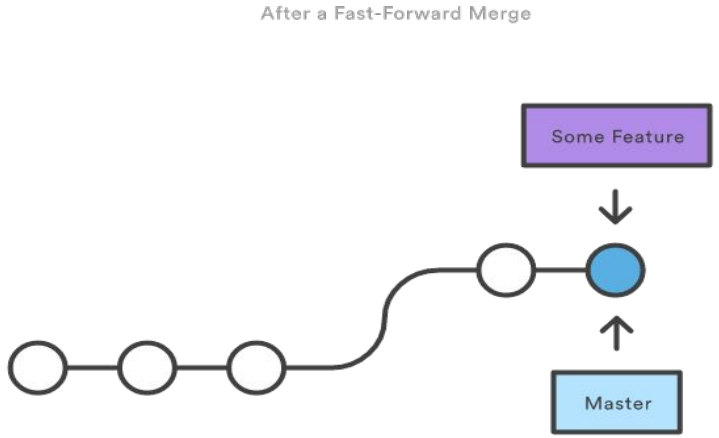
Branch merging

- There exist two types of merges between branches
  - Fast-forward merge
  - Occurs when no commits have been performed in master, but commits have been performed in the branch to be merged
    - Or between any two branches where additional commits have been performed in one with an otherwise similar history

Branch merging

- Diagram of fast-forward merge taken from atlassian.com
- Fast-forward merge combines the histories of both branches
- No merge conflicts, very easy



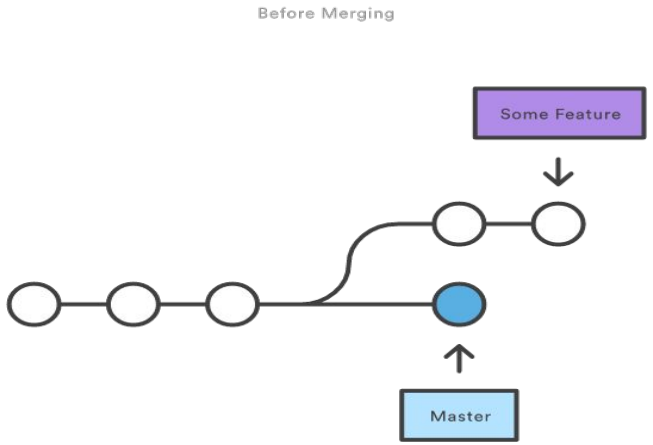


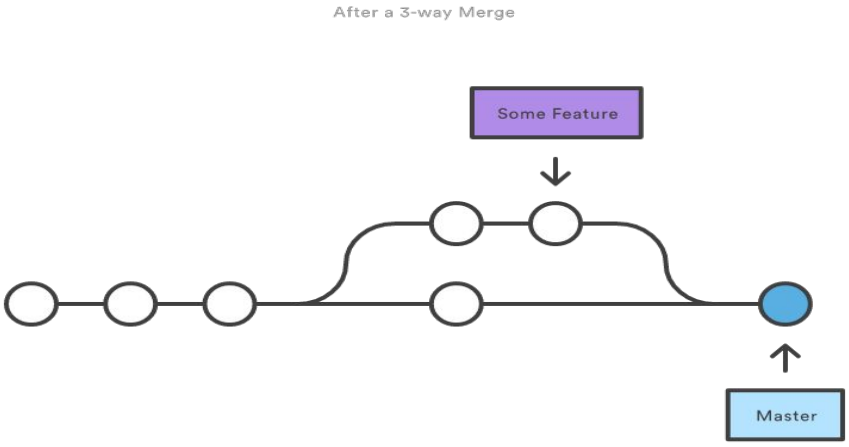
Branch merging

- There exist two types of merges between branches
  - Three-way merge
  - Necessary when branches both have commits that the other lacks
  - Three-way merges use a dedicated commit to tie the two branches together

Branch merging

- Diagram of fast-forward merge taken from atlassian.com
- The commit at the end of the two branches is the commit used to tie the branches together





Branch merging

- To merge a branch into master, use `git merge <branch name>`

Merge conflicts

- Sometimes, the two branches will share files which have been changed differently
- In this case, attempting to merge results in merge conflicts which must be resolved.
- ```
  $ git merge branch
  Auto-merging index.html
  CONFLICT (content): Merge conflict in index.html
  Automatic merge failed; fix conflicts and then commit the result.
  ```
  ```

Merge conflicts

- To see which files have merge conflicts, use `git status`
  - Do this after your attempted merge
- Those files will have the conflicting sections marked, and now it is up to you to manually fix the file so that both sections' changes are represented in the file
  - Once the file has been resolved, use `git add <file name>` to mark it as such
    - `git add <filename>` gives an error on files with merge conflicts

Merge conflicts

- After all changes are merged, use `git status` to check that all changes are merged, then use `git commit` to add the “tying” commit of the 3-way branch
- You can use `git mergetool` in a similar way to git difftool to load any installed external merging tools
- This can be useful for big projects with lots of files that need to be merged between branches

Overall workflow

- `git branch -b <branch>`
- Make changes
- `git merge <branch>`
  - Fix any conflicts and `git commit` after that if necessary

Common workflow patterns

- Long-running branches
  - Have main development branches aside from the stable branch (master)
    - The newest commits end up in the farthest branches, and as they are realized to be stable / bugfixed, the more bleeding-edge branches are merged into less bleeding-edge branches, and so on until finally reaching the stable branch (and are well-developed)

Common workflow patterns

- Long-running branches
  - Ex: Chrome OS (and many Linux distros)
  - From https://support.google.com/chromebook/answer/1086915?hl=en&source=genius-rts
  - **Stable channel**: This channel is fully tested by the Chrome OS team, and is the best choice to avoid crashes and other problems. It's updated roughly every 2-3 weeks for minor changes, and every 6 weeks for major changes.
  - **Beta channel**: If you want to see upcoming changes and improvements with low risk, use the Beta channel. It's updated roughly every week, with major updates coming every 6 weeks, more than a month before the Stable channel gets them.
  - **Dev channel**: If you want to see the latest Chrome OS features, use the Dev channel. The Dev channel gets updated once or twice weekly. While this build is tested, it might have bugs, as we want people to see what's new as soon as possible.

Common workflow patterns

- Long-running branches
  - Another pattern is to have a branch dedicated to releases, separate and away from the development branches
  - Whenever a new version is to be released, the primary development branch is merged with the release branch to form a new release

Common workflow patterns

- Topic / feature branches
  - To add a new feature, have a separate branch for it so that your changes can be completely aside, then when your feature is complete, merge back in
  - Usually used when adding new features to open source projects

Common workflow patterns

- Topic / feature branches
  - To add a new feature, have a separate branch for it so that your changes can be completely aside, then when your feature is complete, merge back in
  - Usually used when adding new features to open source projects
  - Never interact with non-development branches

Common workflow patterns

- Hotfix branches
  - Used for emergency patches of releases
  - Branch off of the release branch, make your fixes, then merge back into release branch
  - Usually responsible for sub-release or sub-sub release numbers (version 1.01 or 1.0.1)

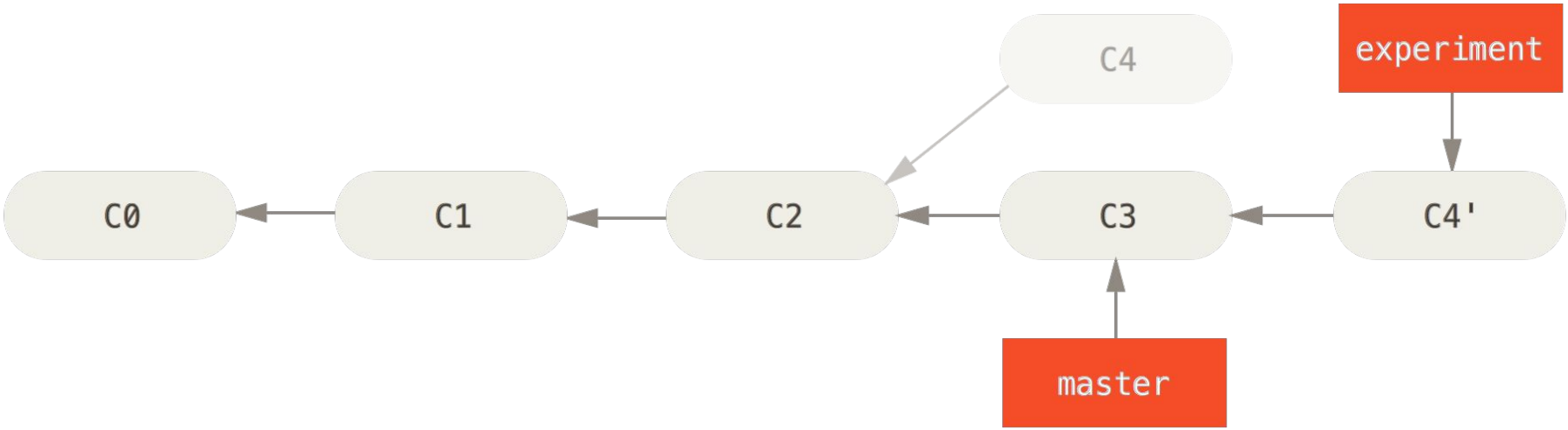
Rebasing

- Rebasing takes all of the changes in your branch and commits them onto another one.
- It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

Rebasing

(image from git-scm.com)

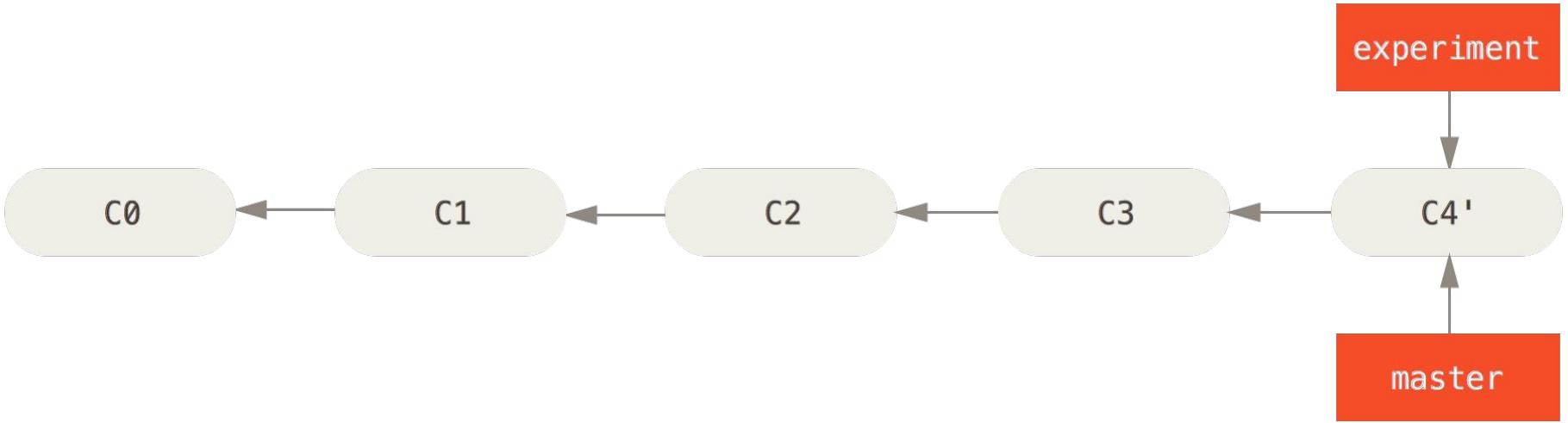
- `git checkout experiment`
- `git rebase master`



Rebasing

(image from git-scm.com)

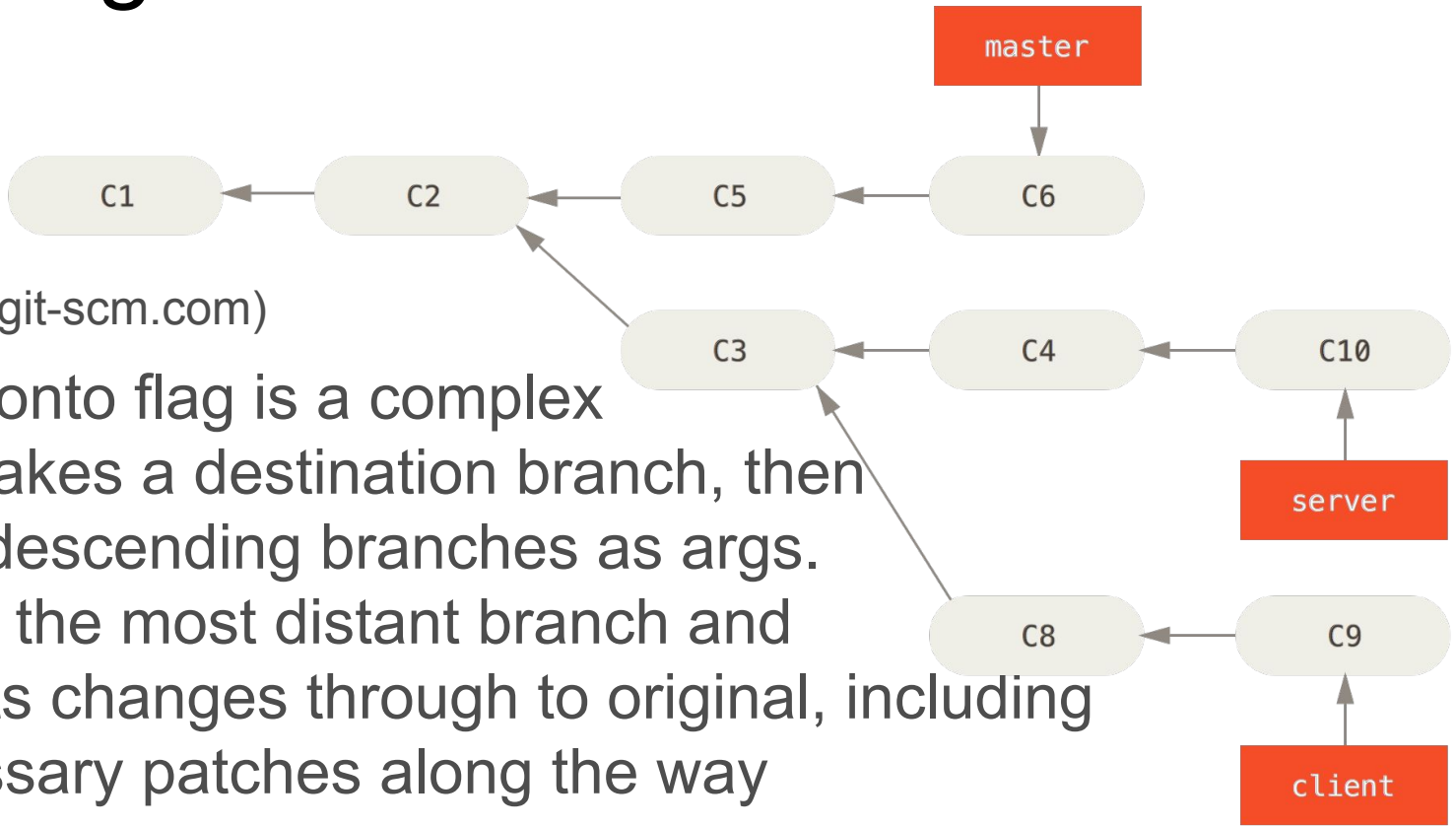
- `git checkout master`
- `git merge experiment`



Rebasing

- Rebasing is useful for having a more simplified commit history- instead of having a tree that loops into itself repeatedly with different branches, your commit history can looks like a single line
- This might seem unnecessary but it can greatly simplify commit histories necessarily using many branches

Rebasing



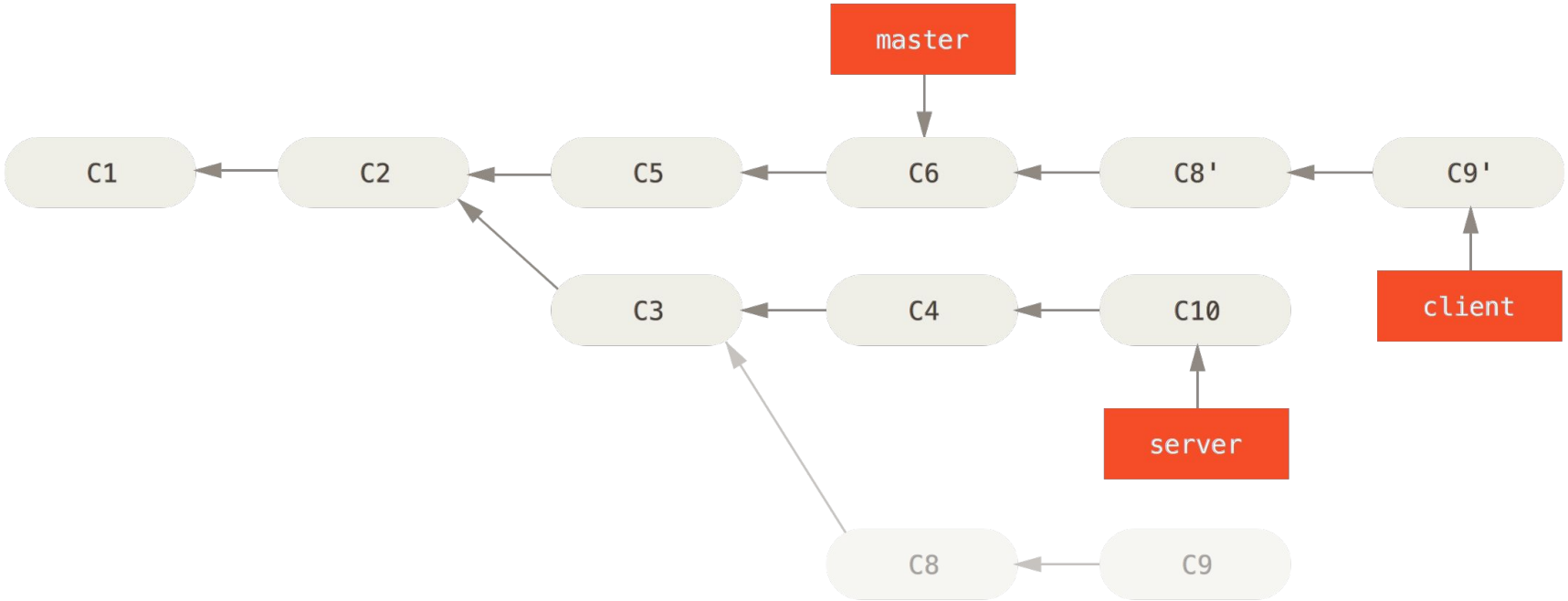
(image from git-scm.com)

- The --onto flag is a complex flag that takes a destination branch, then a tree of descending branches as args.
- Takes the most distant branch and patches its changes through to original, including the necessary patches along the way

Rebasing

(image from git-scm.com)

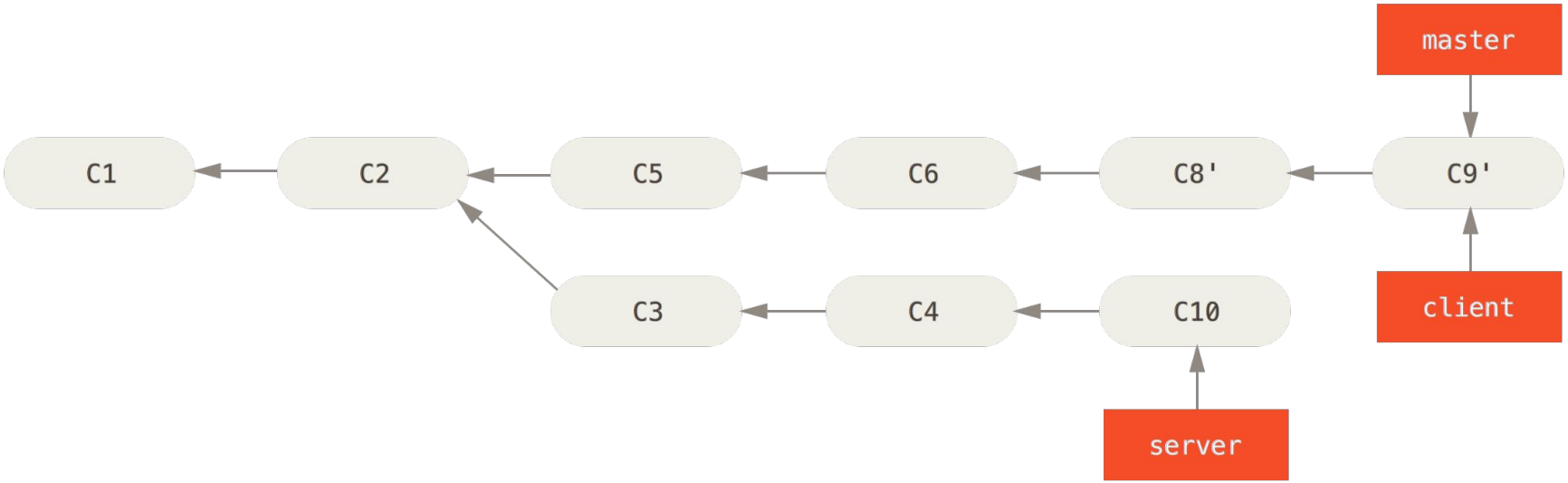
- `git rebase --onto master server client`



Rebasing

(image from git-scm.com)

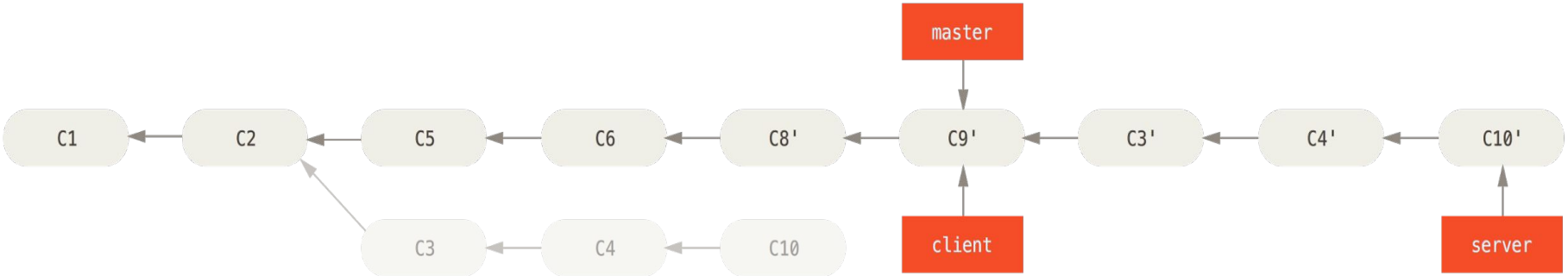
- `git checkout master`
- `git merge client`



Rebasing

(image from git-scm.com)

- `git rebase master server`
  - Takes the server branch's changes and plays them on top of master



Rebasing

(image from git-scm.com)

- `git checkout master`
- `git merge server`
  - Fast-foward merges master to server
- `git branch -d client`
- `git branch -d server`
  - Removes the extra branches

Much simpler commit history.