Pro Git(https://git-scm.com/book/ko/v2/)를 바탕으로 작성하였습니다.

YoonJoon Lee
SoC KAIST

Fall 2018

What we will take a look in this series

1. Getting Started
2. Git Basics
3. Git Branch
4. Git Server - GitLab

What we will take a look today

1.Getting Started

2. Git Basics

3. Git Branch

4. Git Server - GitLab

What I will talk about in this section

1. Branches in a Nutshell
2. Basic Branching and Merging
3. Branch Management
4. Branching Workflows
5. Remote Branches
6. Rebasing

Branch Management

The `git branch` command does more than just create and delete branches. If we run it with no arguments, we get a simple listing of your current branches:

```
$ git branch
  iss53
* master
  testing
```

To see the last commit on each branch, you can run `git branch -v`:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

The useful `--merged` and `--no-merged` options can filter this list to branches that we have or have not yet merged into the branch we're currently on.

```
$ git branch --merged
  iss53
* master
```

```
$ git branch --no-merged
  testing
```

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Branching Workflows

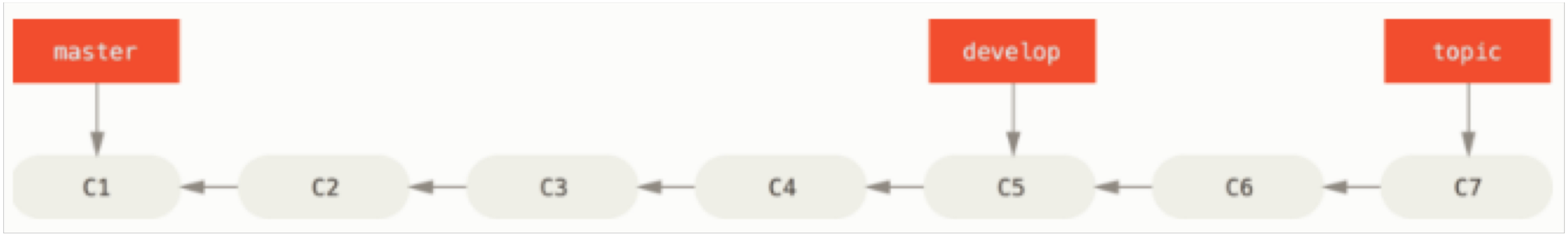
We'll cover some common workflows that this lightweight branching makes possible, so we can decide if we would like to incorporate them into our own development cycle.

Long-Running Branches

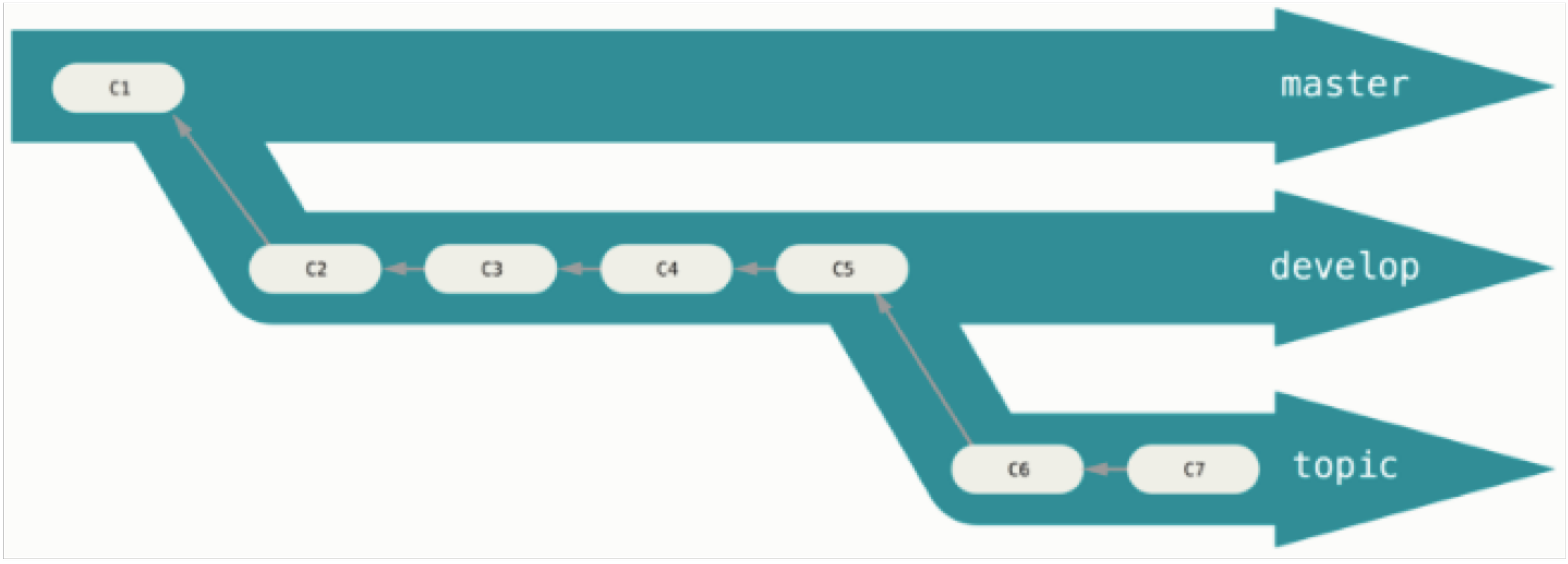
Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch — possibly only code that has been or will be released.

They have another parallel branch named `develop` or `next` that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into master.

It's used to pull in `topic` branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.



It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.



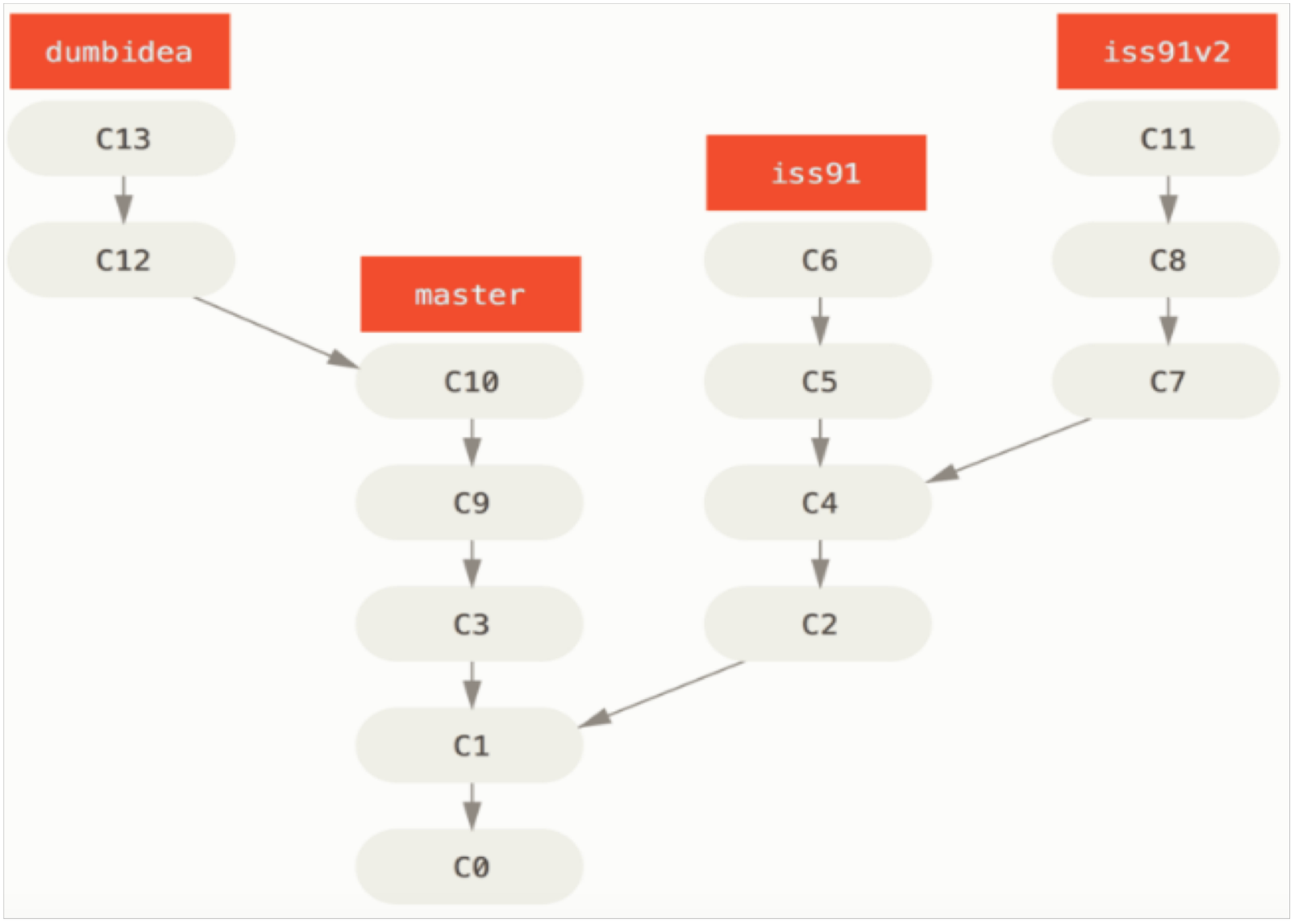
We can keep doing this for several levels of stability. Some larger projects also have a **`proposed`** or **`pu`** (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch.

Topic Branches

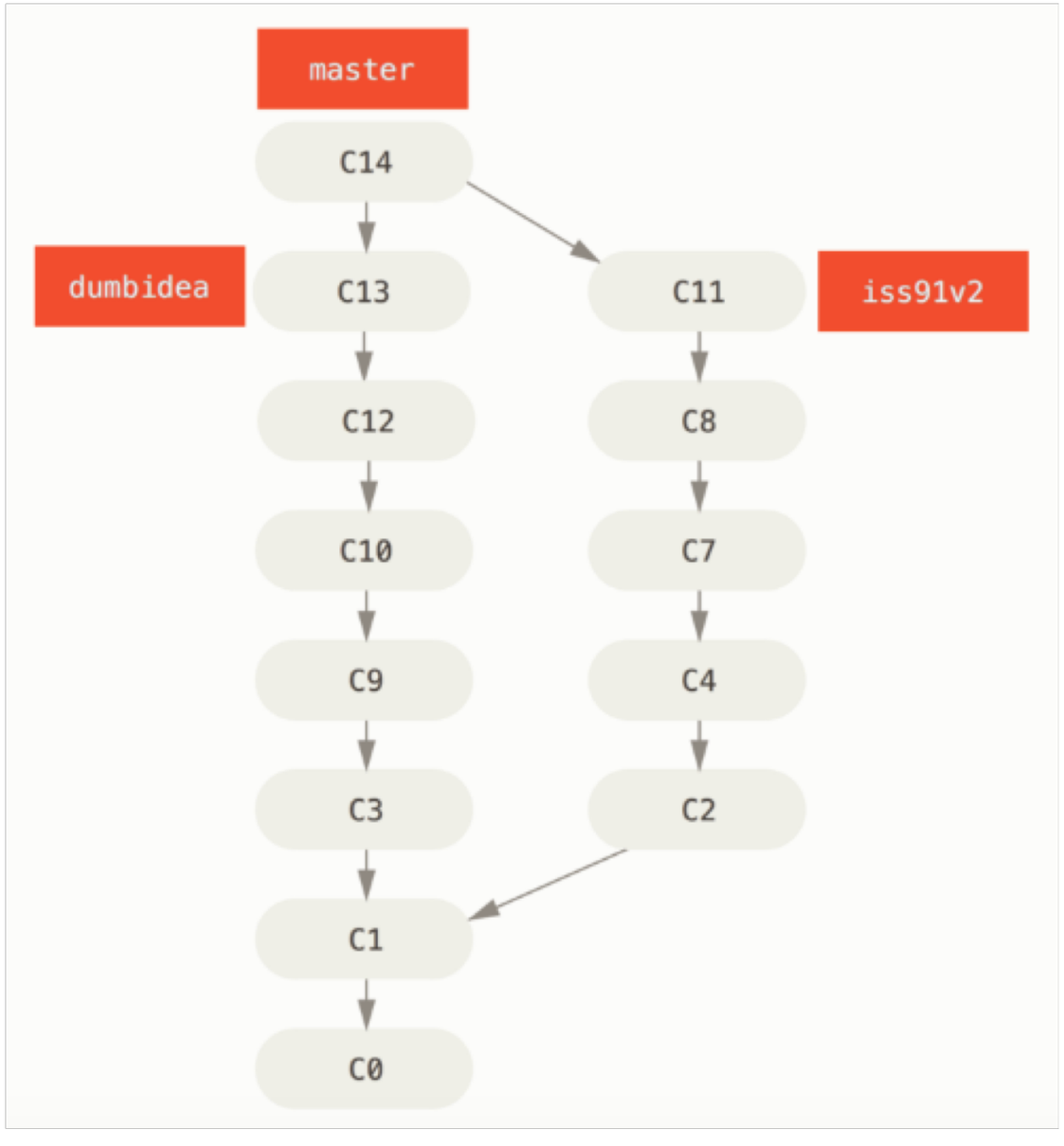
A topic branch is a short-lived branch that you create and use for a single particular feature or related work.

We saw this with the `iss53` and `hotfix` branches we created. We did a few commits on them and deleted them directly after merging them into our main branch.

Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to our `master` branch and working there for a while, and then branching off there to do some work that we're not sure is a good idea (`dumbidea` branch).



Let's say we decide we like the second solution to your issue best (`iss91v2`); and we showed the `dumbidea` branch to our coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two.



Remote Branches

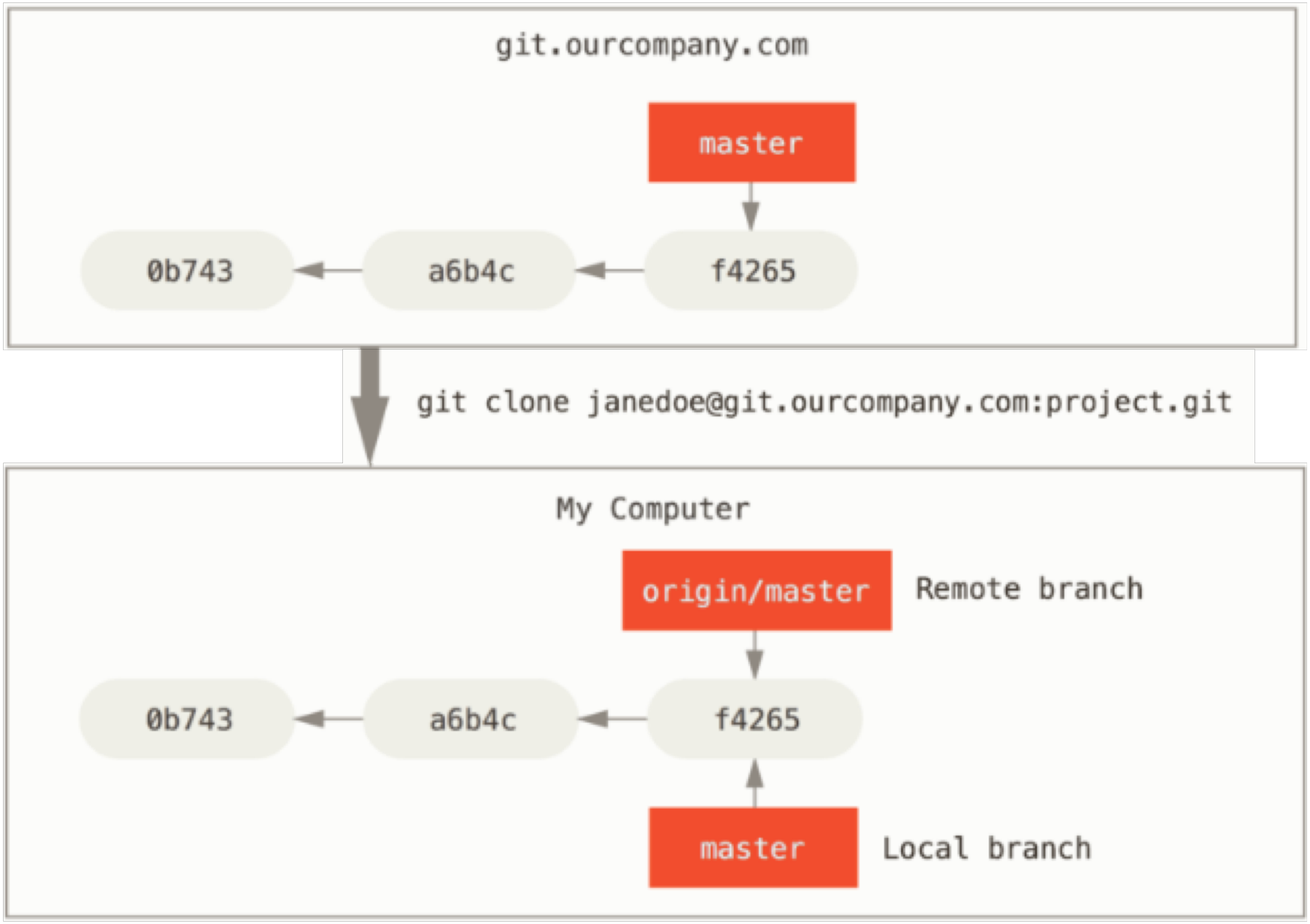
Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. We can get a full list of remote references explicitly with `git ls-remote [remote]`, or `git remote show [remote]` for remote branches as well as more information. Nevertheless, a more common way is to take advantage of remote-tracking branches.

Remote-tracking branches are references to the state of remote branches. They're local references that we can't move; Git moves them for us whenever we do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, o remind you where the branches in your remote repositories were the last time you connected to them.

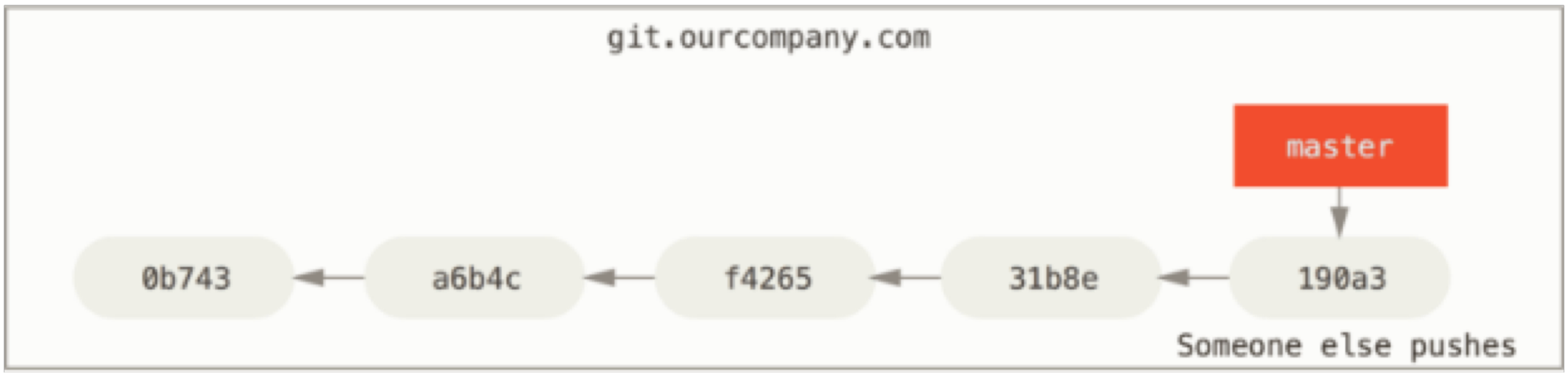
Remote-tracking branches take the form `<remote>/<branch>`. For instance, if we wanted to see what the `master` branch on our `origin` remote looked like as of the last time we communicated with it, we would check the `origin/master` branch.

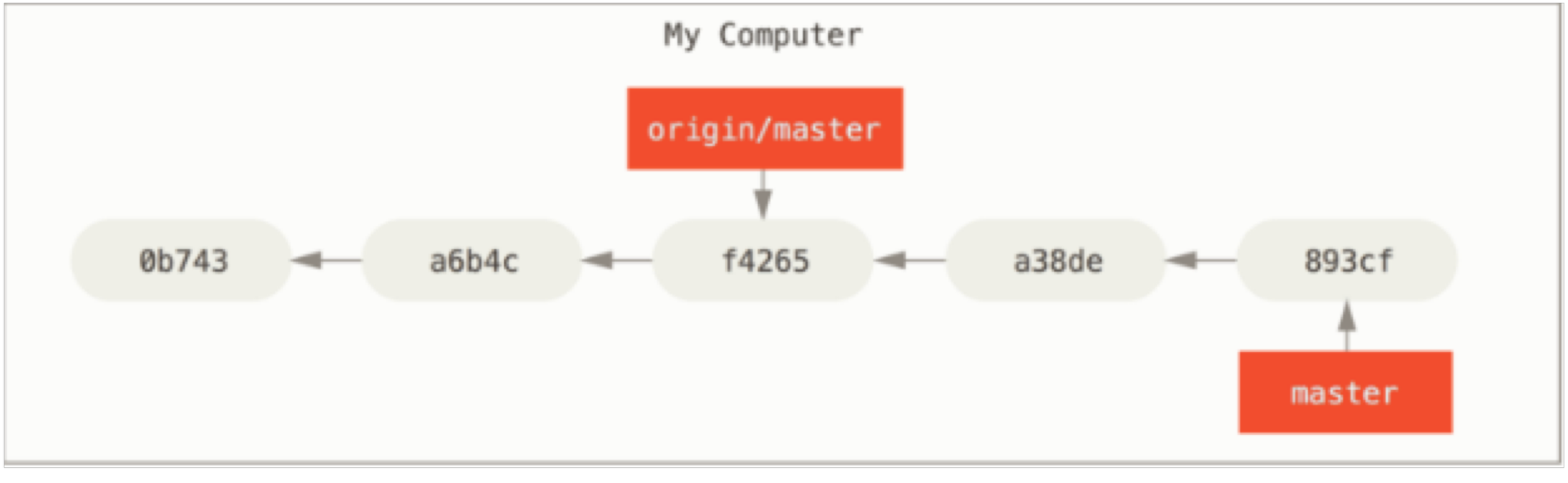
If we were working on an issue with a partner and they pushed up an `iss53` branch, we might have our own local `iss53` branch, but the branch on the server would be represented by the remote-tracking branch `origin/iss53`.

Let's say we have a Git server on our network at `git.ourcompany.com`. If we clone from this, Git's `clone` command automatically names it `origin` for us, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally. Git also gives you your own local `master` branch starting at the same place as origin's `master` branch, so we have something to work from.

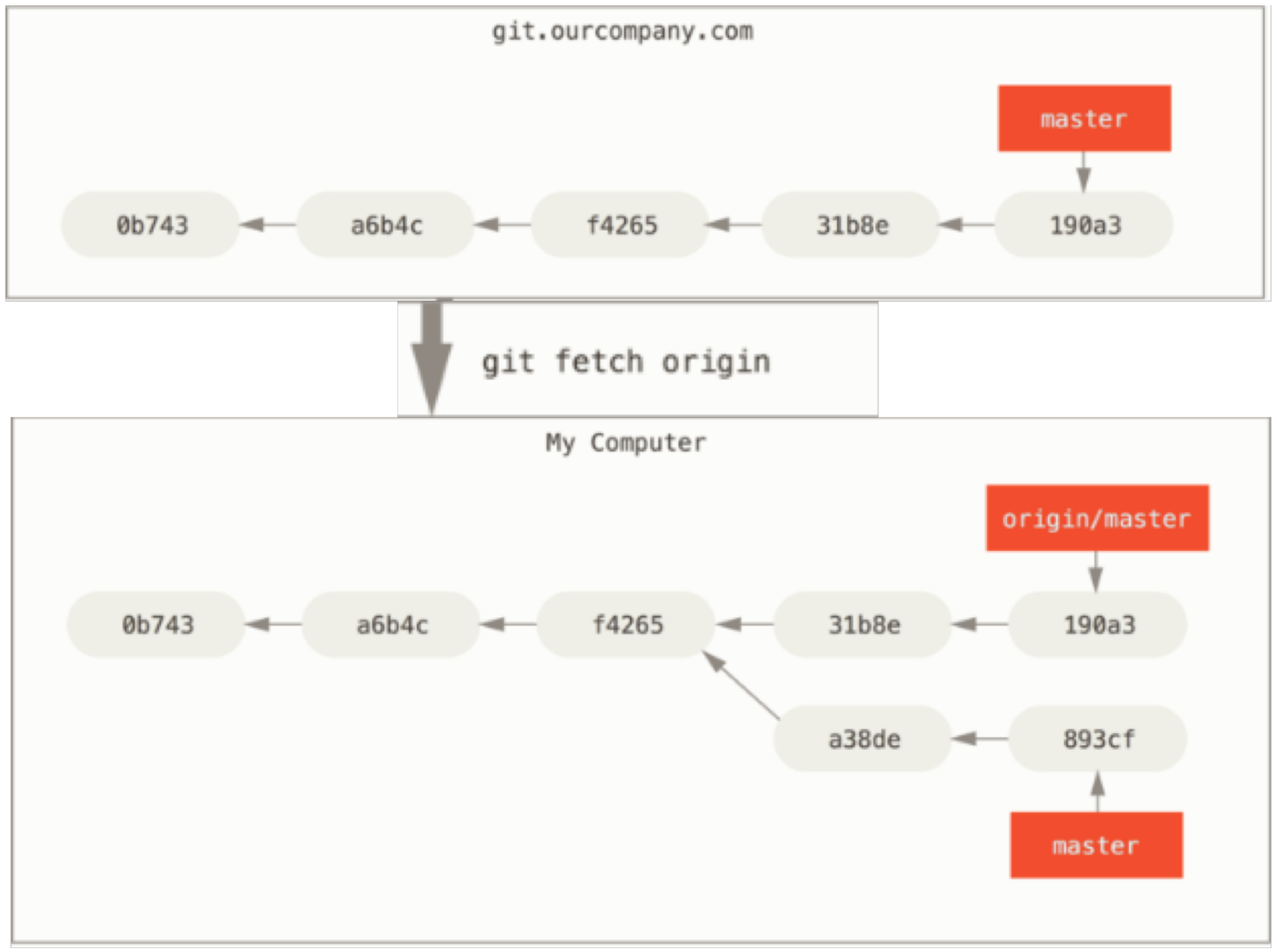

git.ourcompany.com
master
0b743
a6b4c
f4265
git clone janedoe@git.ourcompany.com:project.git
My Computer
origin/master
Remote branch
0b743
a6b4c
f4265
master
Local branch

If we do some work on our local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then our histories move forward differently. Also, as long as we stay out of contact with our origin server, our `origin/master` pointer doesn't move.

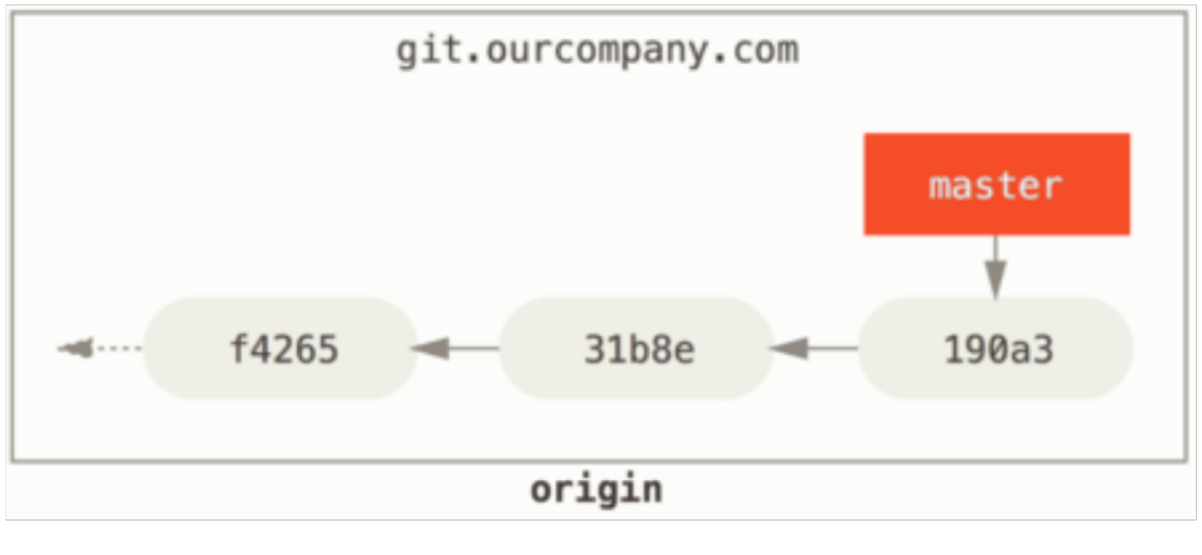


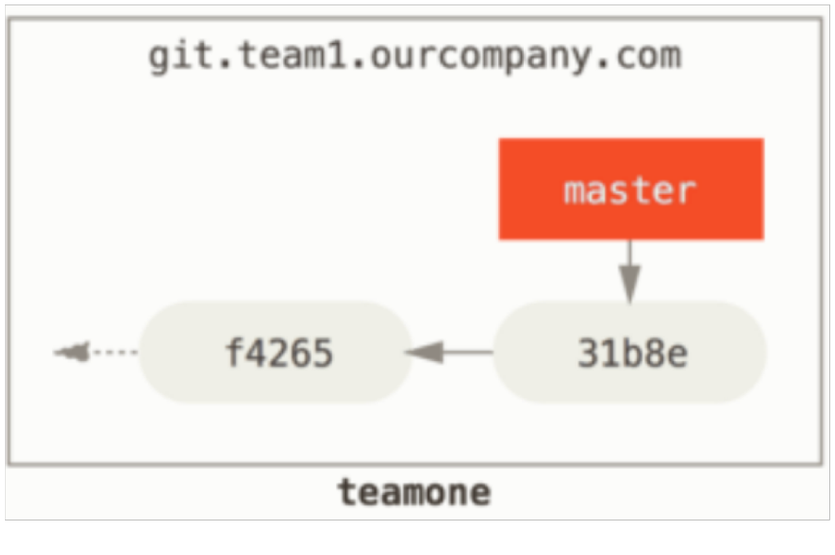


To synchronize our work, we run a `git fetch origin` command. This command looks up which server "origin" is (in this case, it's `git.ourcompany.com`), fetches any data from it that we don't yet have, and updates our local database, moving your `origin/master` pointer to its new, more up-to-date position.

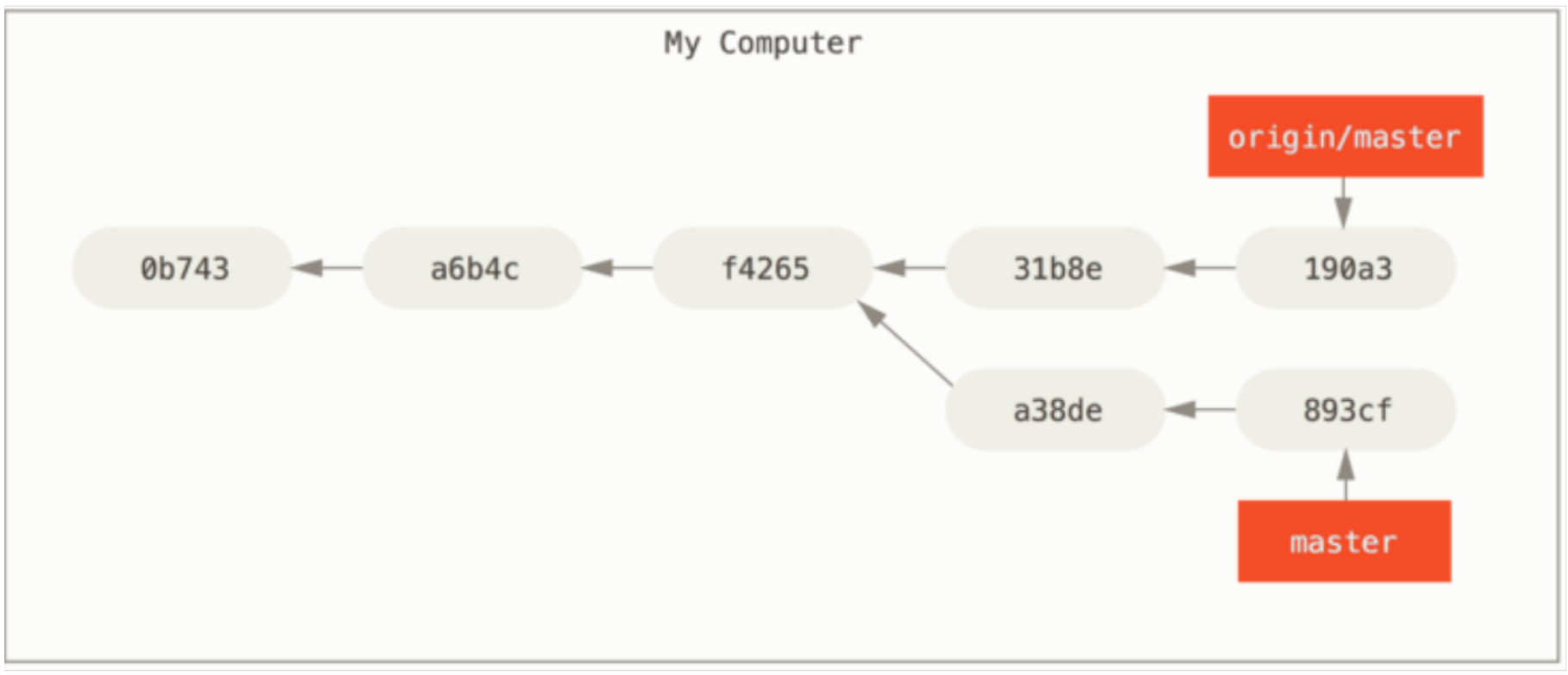


To demonstrate having multiple remote servers and what remote branches for those remote projects look like,

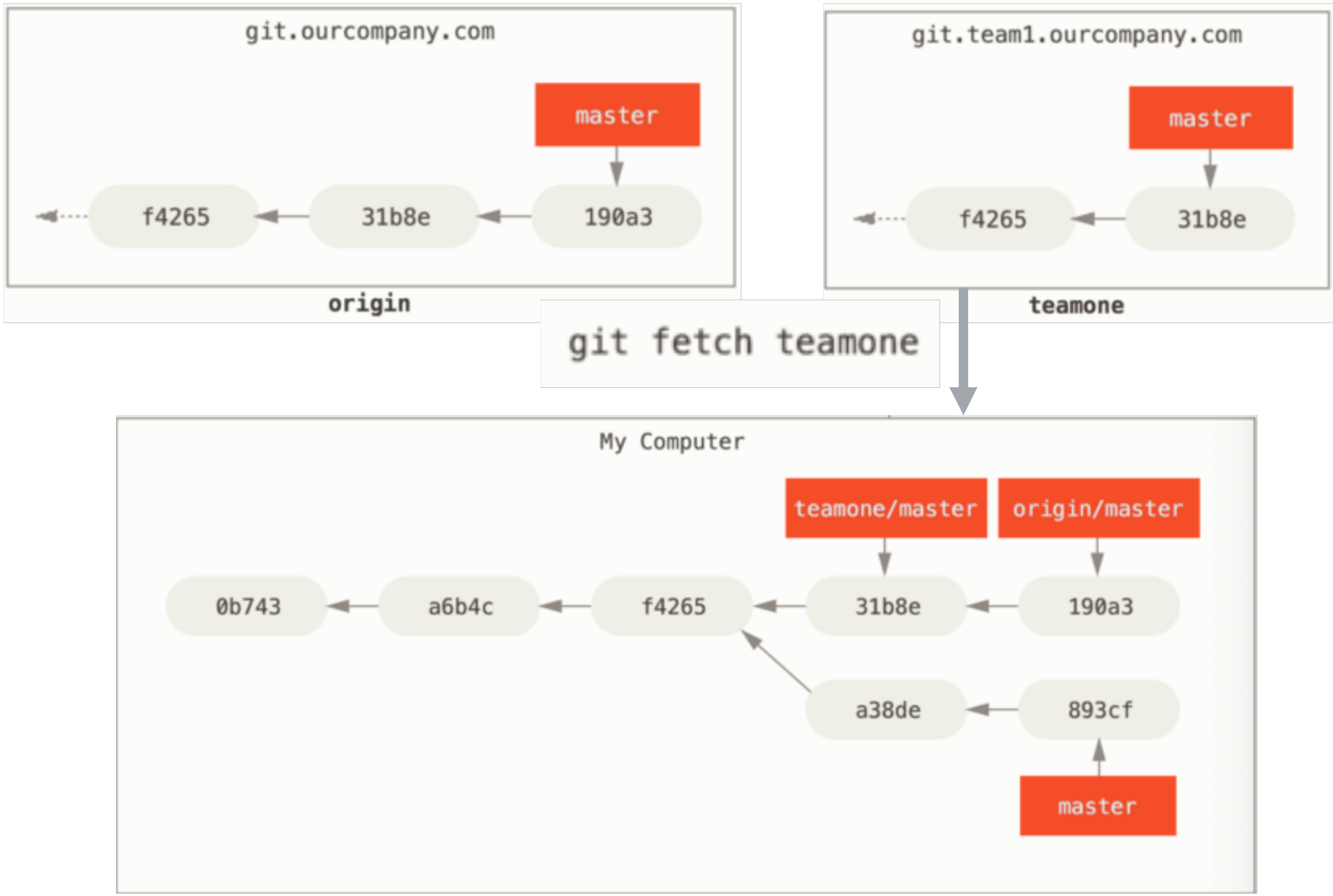





```
git remote add teamone git://git.team1.ourcompany.com
```



Now, we can run `git fetch teamone` to fetch everything the remote `teamone` server has that we don't have yet.



Pushing

When we want to share a branch with the world, we need to push it up to a remote that we have write access to.

Our local branches aren't automatically synchronized to the remotes we write to — we have to explicitly push the branches we want to share.

That way, we can use private branches for work we don't want to share, and push up only the topic branches we want to collaborate on.

If we have a branch named **`serverfix`** that you want to work on with others, we can push it up the same way we pushed our first branch. Run **`git push <remote> <branch>`**:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Git automatically expands the `serverfix` branchname out to `refs/heads/serverfix:refs/heads/serverfix`, which means, "Take my serverfix local branch and push it to update the remote's serverfix branch."

We can also do `git push origin serverfix:serverfix`, which does the same thing—it says, "Take my serverfix and make it the remote's serverfix."

We can use this format to push a local branch into a remote branch that is named differently. If we didn't want it to be called `serverfix` on the remote, we could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

The next time one of our collaborators fetches from the server, they will get a reference to where the server's version of `serverfix` is under the remote branch `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

It's important to note that when we do a fetch that brings down new remote-tracking branches, we don't automatically have local, editable copies of them. In other words, in this case, we don't have a new `serverfix` branch — we only have an `origin/serverfix` pointer that we can't modify.

To merge this work into our current working branch, we can run **`git merge origin/serverfix`**. If we want our own **`serverfix`** branch that we can work on, we can base it off our remote-tracking branch:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”).

Tracking branches are local branches that have a direct relationship to a remote branch. If we’re on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and which branch to merge in.

When we clone a repository, it generally automatically creates a master branch that tracks **`origin/master`**. However, we can set up other tracking branches if we wish — ones that track branches on other remotes, or don't track the master branch.

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

This is so common that there's even a shortcut for that shortcut. If the branch name we're trying to checkout

(a)doesn't exist and

(b)exactly matches a name on only one remote,

Git will create a tracking branch for you:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

To set up a local branch with a different name than the remote branch, we can easily use the first version with a different local branch name:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

If we already have a local branch and want to set it to a remote branch we just pulled down, or want to change the upstream branch we're tracking, we can use the **`-u`** or **`--set-upstream-to`** option to git branch to explicitly set it at any time.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

If we want to see what tracking branches we have set up, we can use the `-vv` option to `git branch`. This will list out our local branches with more information including what each branch is tracking and if our local branch is ahead, behind or both.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Our `iss53` branch is tracking `origin/iss53` and is "ahead" by two, meaning that we have two commits locally that are not pushed to the server.

Our `master` branch is tracking `origin/master` and is up to date.

Our `serverfix` branch is tracking the `server-fix-good` branch on our `teamone` server and is ahead by three and behind by one, meaning that there is one commit on the server we haven't merged in yet and three commits locally that we haven't pushed.

Finally our `testing` branch is not tracking any remote branch.

It's important to note that these numbers are only since the last time we fetched from each server. This command does not reach out to the servers, it's telling us about what it has cached from these servers locally.

If we want totally up to date ahead and behind numbers, we'll need to fetch from all our remotes right before running this. We could do that like this:

```
$ git fetch --all; git branch -vv
```

Pulling

While the git fetch command will fetch down all the changes on the server that we don't have yet, it will not modify our working directory at all. It will simply get the data for us and let us merge it ourself.

The command **`git pull`** which is essentially a **`git fetch`** immediately followed by a **`git merge`** in most cases.

If we have a tracking branch set up, either by explicitly setting it or by having it created for us by the **`clone`** or **`checkout`** commands, **`git pull`** will look up what server and branch our current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it's better to simply use the **`fetch`** and **`merge`** commands explicitly as the magic of **`git pull`** can often be confusing.

Deleting Remote Branches

Suppose we're done with a remote branch – say our collaborators are finished with a feature and have merged it into our remote's master branch (or whatever branch our stable codeline is in). We can delete a remote branch using the `--delete` option to `git push`. If we want to delete our `serverfix` branch from the server, we run the following:

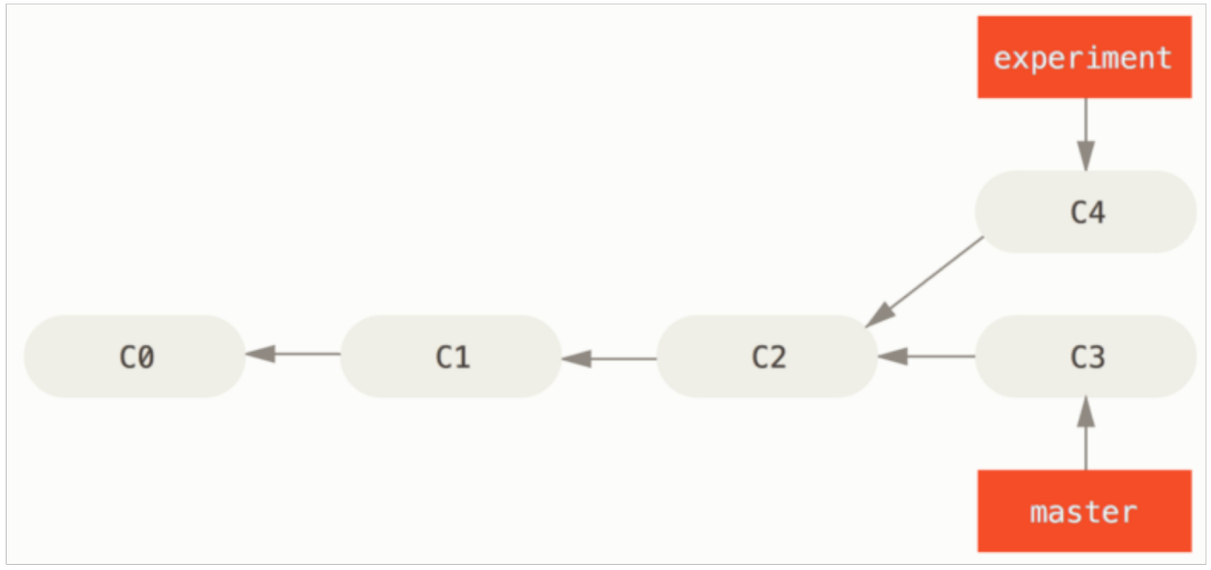
```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

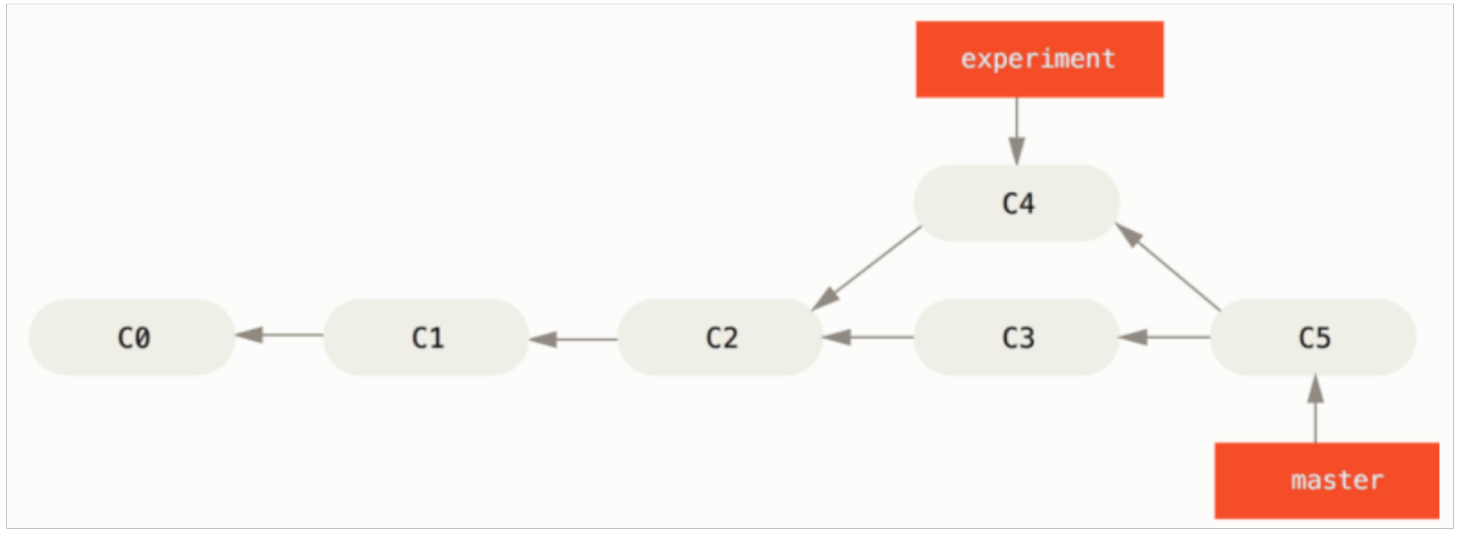
Rebasing

There are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`.

The Basic Rebase

Basic Merging

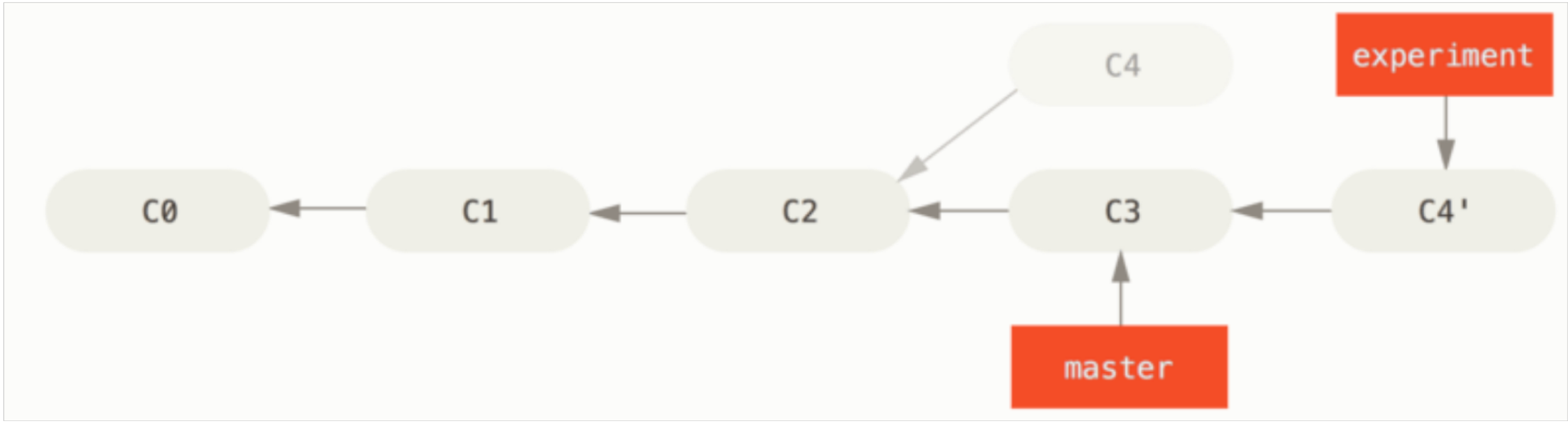




There is another way: we can take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*. With the **`rebase`** command, we can take all the changes that were committed on one branch and replay them on another one.

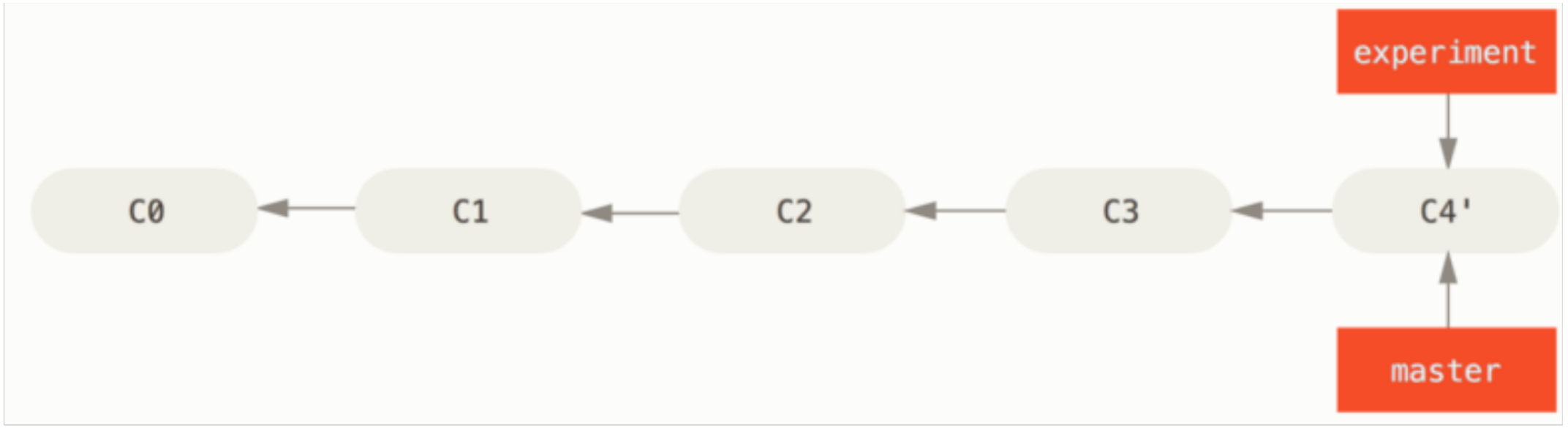
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

It works by going to the common ancestor of the two branches (the one we're on and the one we're rebasing onto), getting the diff introduced by each commit of the branch we're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch we are rebasing onto, and finally applying each change in turn.



We can go back to the `master` branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```



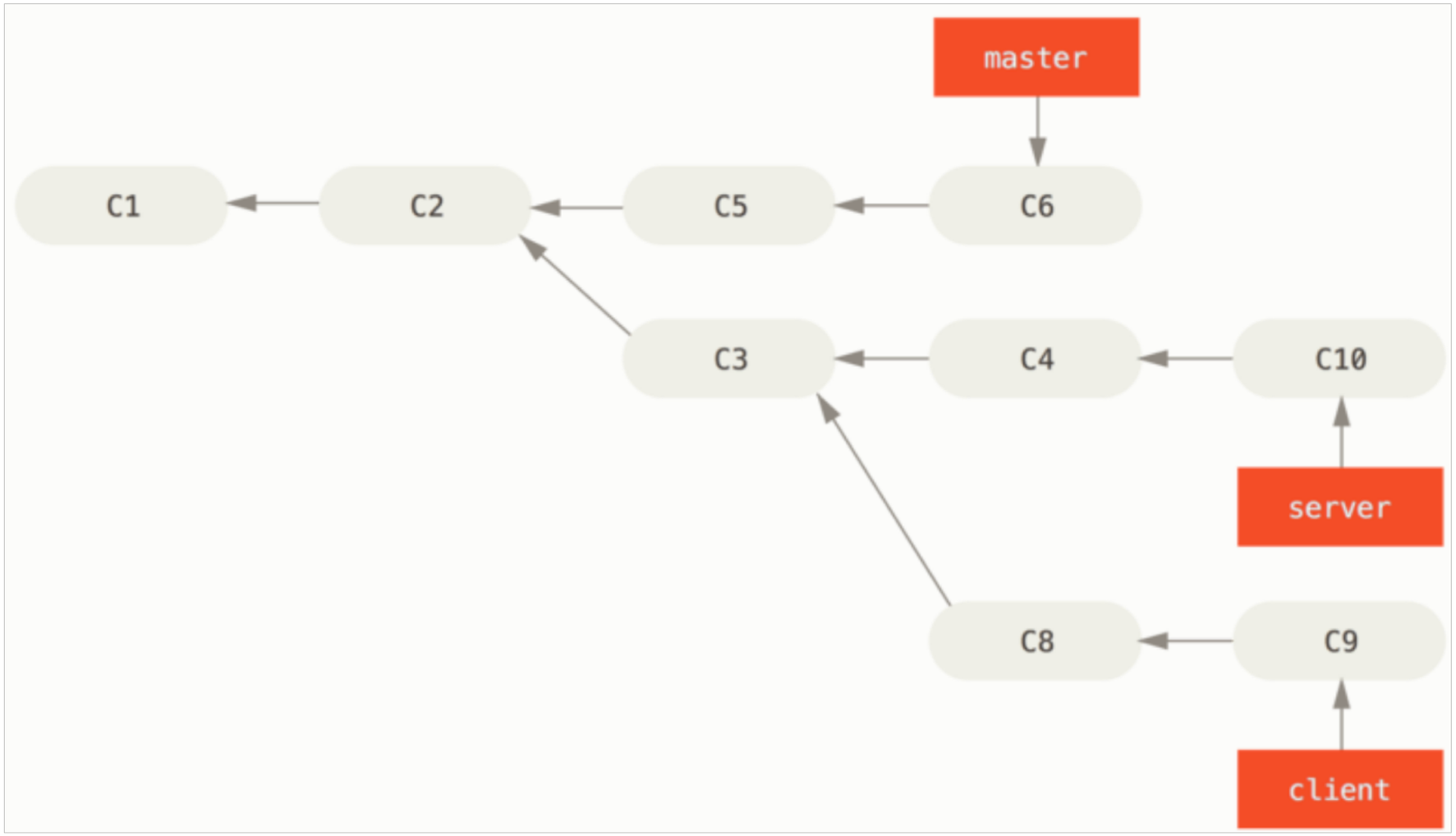
The snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If we examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

We'll do rebasing to make sure our commits apply cleanly on a remote branch — perhaps in a project to which we're trying to contribute but that we don't maintain. In this case, we'd do our work in a branch and then rebase our work onto origin/master when we were ready to submit our patches to the main project. That way, the maintainer doesn't have to do any integration work — just a fast-forward or a clean apply.

The snapshot pointed to by the final commit we end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it's only the history that is different.

More Interesting Rebases

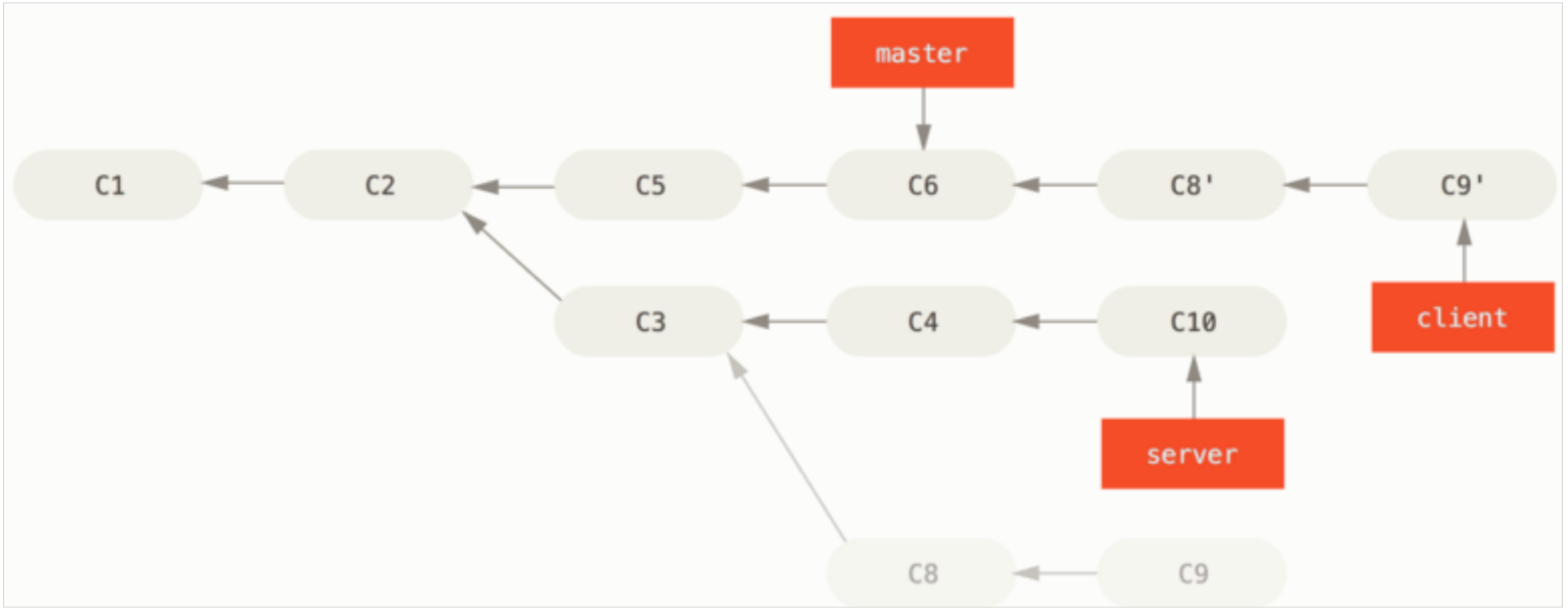
We can also have our rebase replay on something other than the rebase target branch. Take a history:



Suppose we decide that we want to merge our client-side changes into our mainline for a release, but we want to hold off on the server-side changes until it's tested further. We can take the changes on client that aren't on server (`C8` and `C9`) and replay them on our master branch by using the `--onto` option of `git rebase`:

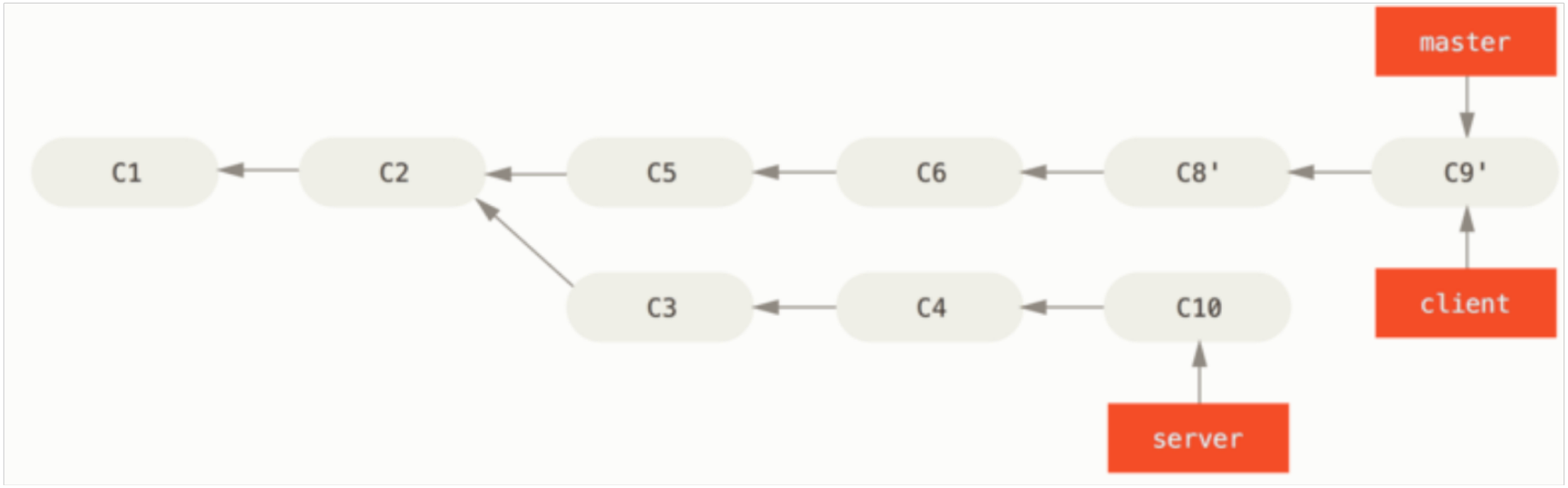
```
$ git rebase --onto master server client
```

This basically says, "Take the `client` branch, figure out the patches since it diverged from the `server` branch, and replay these patches in the client branch as if it was based directly off the `master` branch instead." It's a bit complex, but the result is pretty cool.



We can fast-forward our master branch:

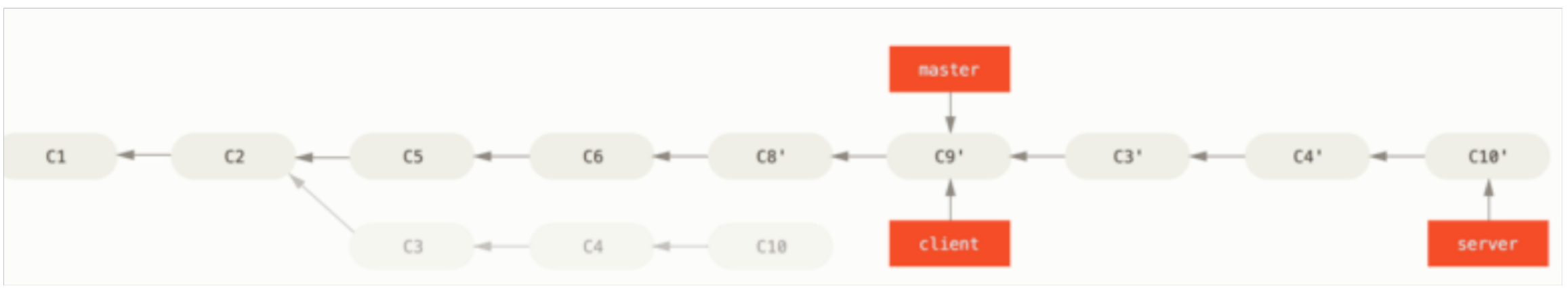
```
$ git checkout master
$ git merge client
```



We decide to pull in our **`server`** branch as well. We can rebase the **`server`** branch onto the **`master`** branch without having to check it out first by running **`git rebase <basebranch> <topicbranch>`** — which checks out the topic branch (in this case, **`server`**) for us and replays it onto the base branch (**`master`**):

```
$ git rebase master server
```

This replays your server work on top of your master work.

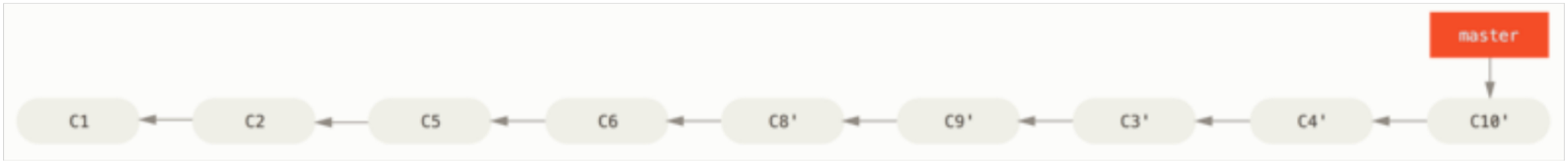


We can fast-forward the base branch (`master`):

```
$ git checkout master
$ git merge server
```

We can remove the `client` and `server` branches because all the work is integrated and we don't need them anymore, leaving our history for this entire process looking like Final commit history:

```
$ git branch -d client
$ git branch -d server
```

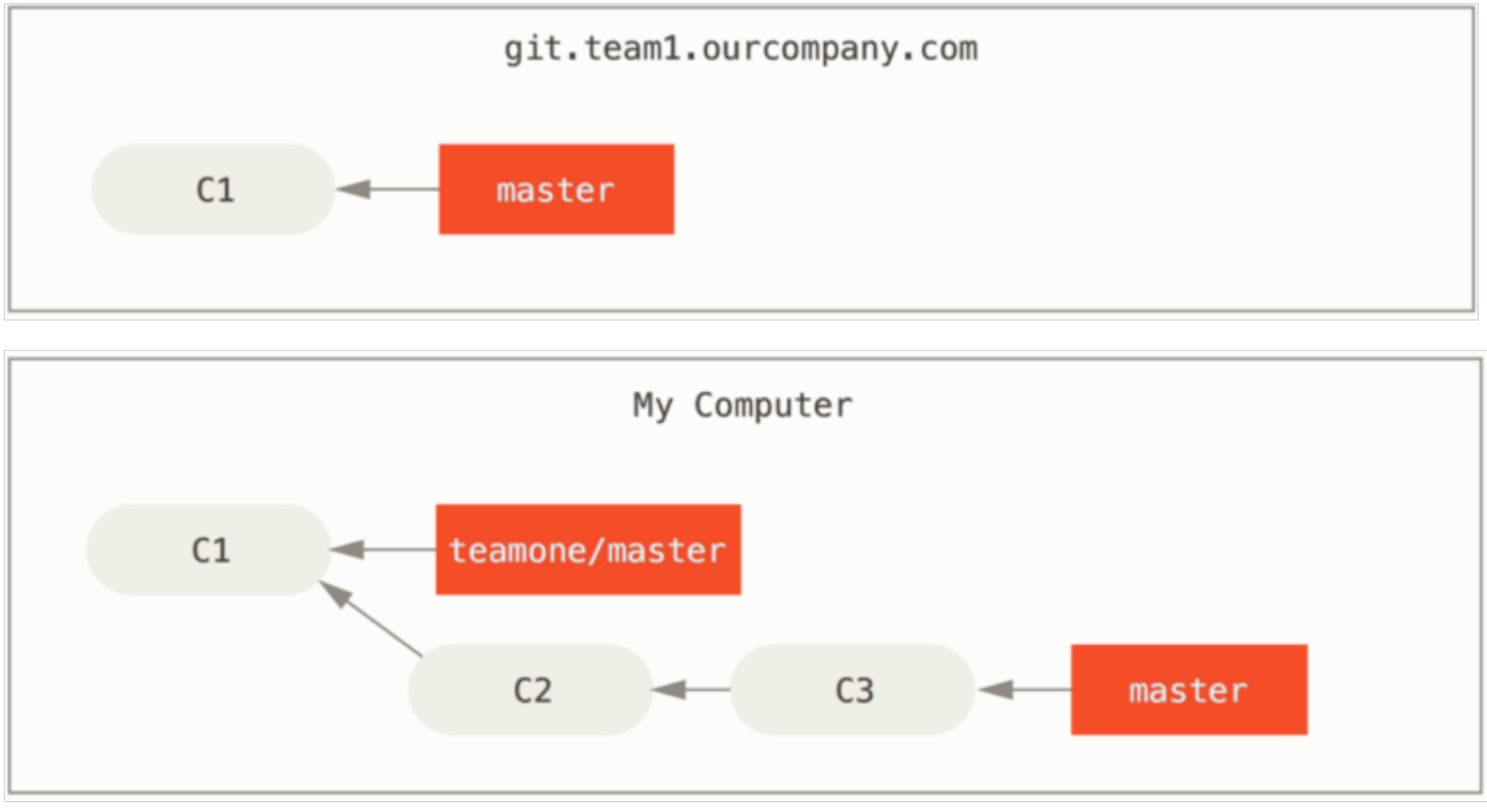


The Perils of Rebasing

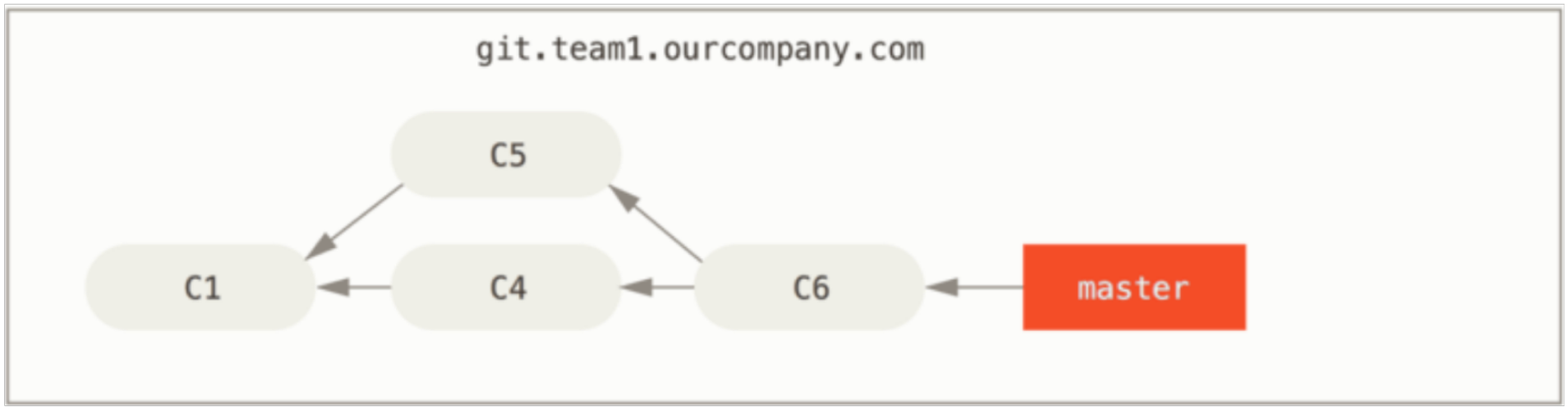
The bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

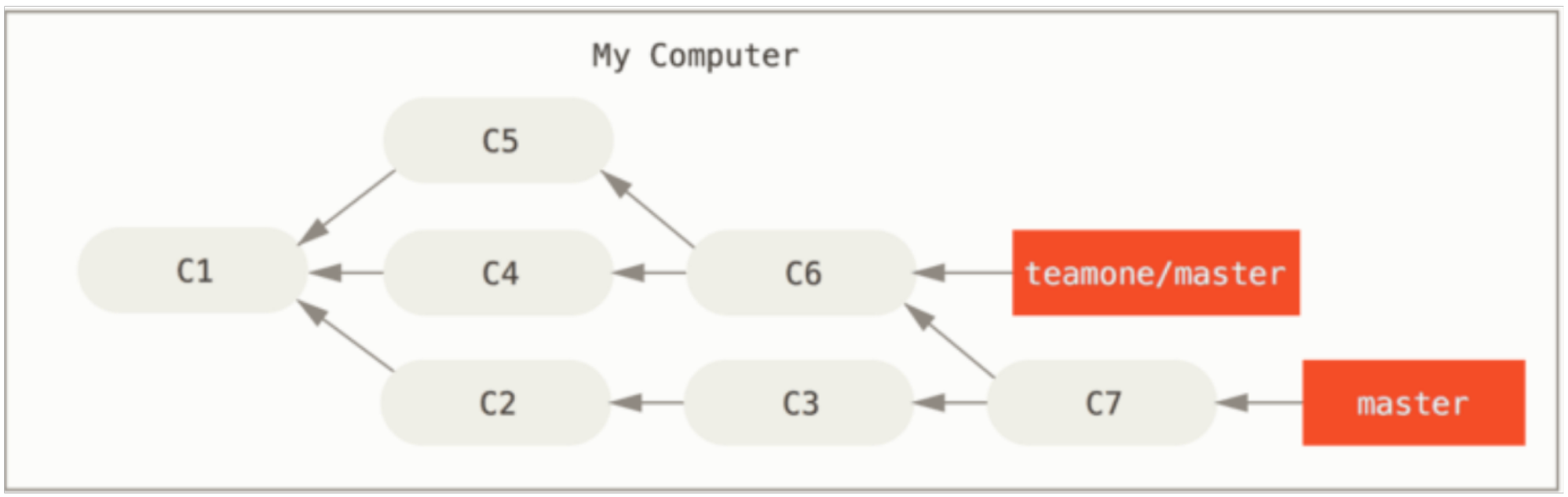
Do not rebase commits that exist outside your repository.

Let's look at an example of how rebasing work that we've made public can cause problems. Suppose we clone from a central server and then do some work off that. Our commit history looks like this:

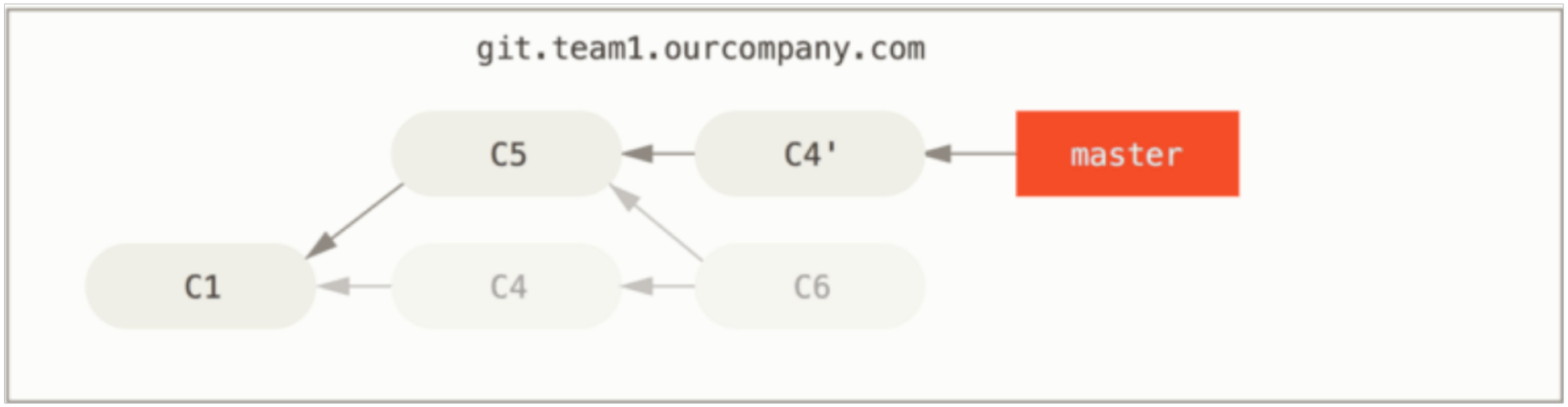


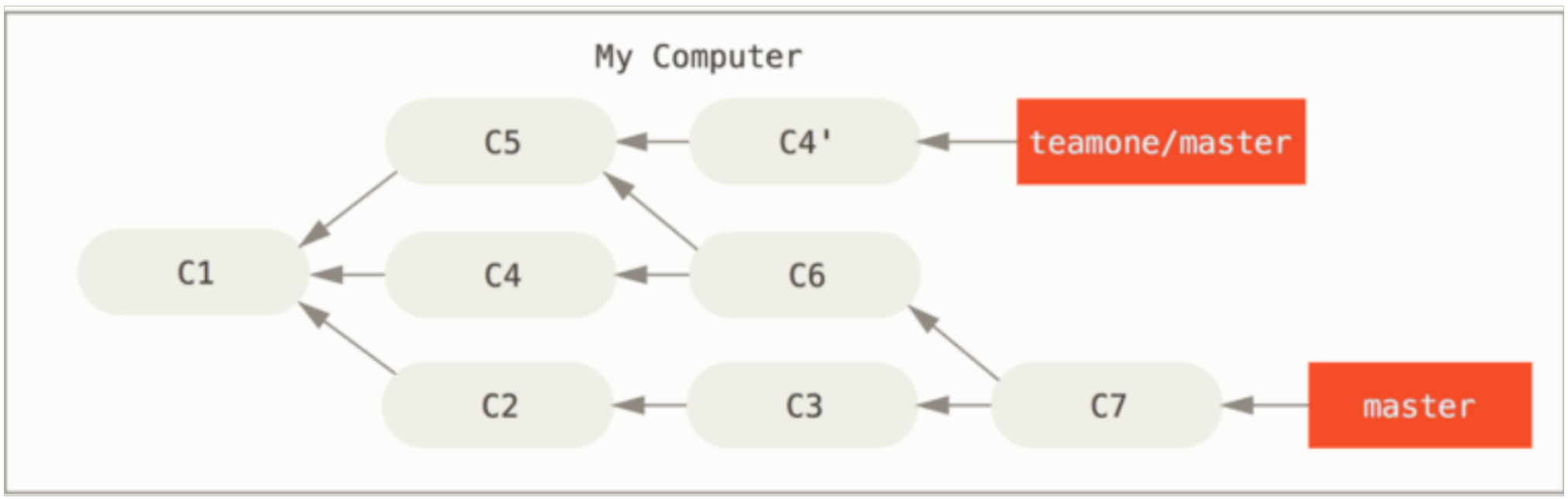
Someone else does more work that includes a merge, and pushes that work to the central server. We fetch it and merge the new remote branch into our work, making our history look something like this:



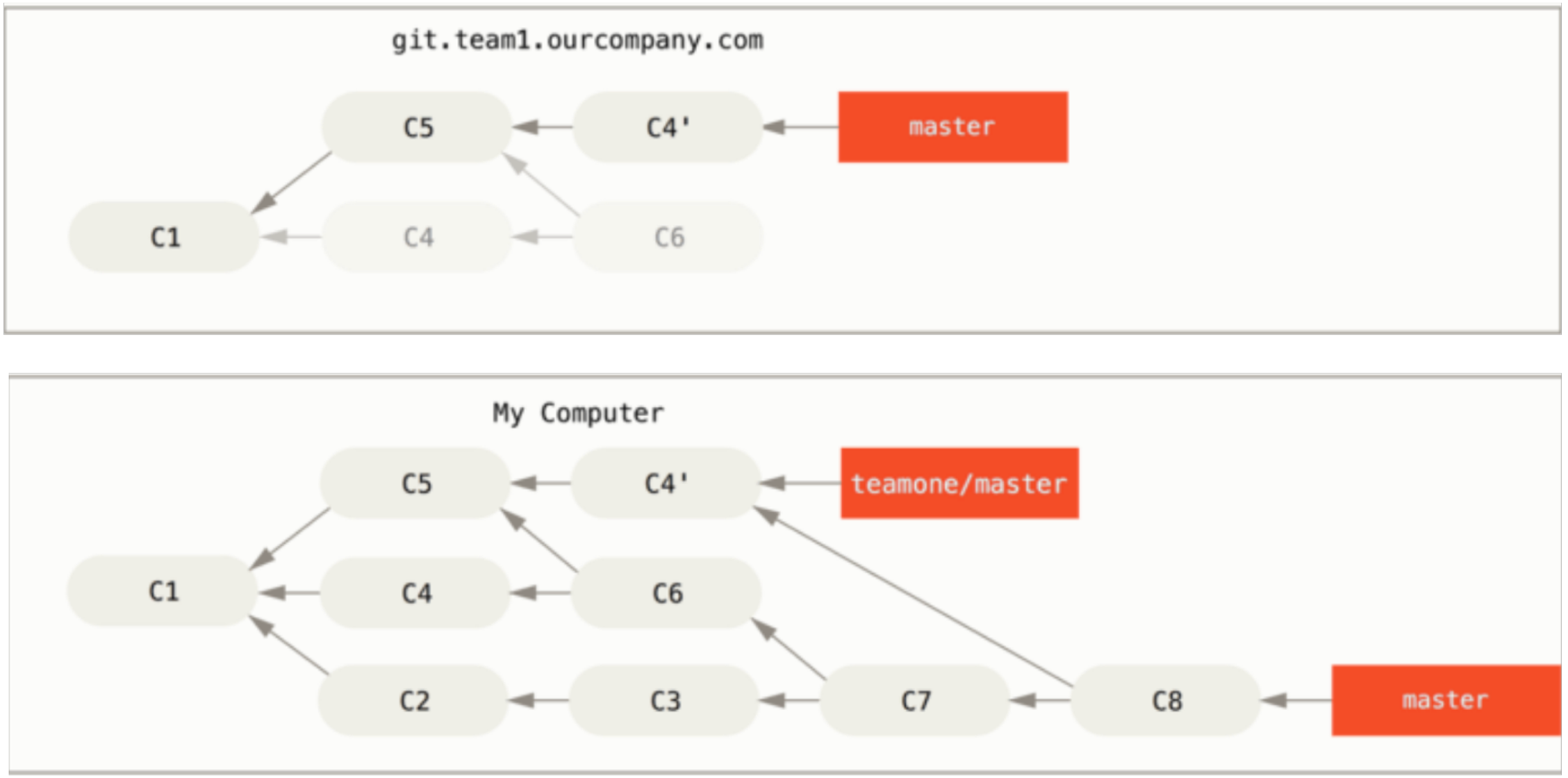


Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. We then fetch from that server, bringing down the new commits.





Now we're both in a pickle. If we do a `git pull`, we'll create a merge commit which includes both lines of history, and our repository will look like this:



If we run a `git log` when our history looks like this, we'll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if we push this history back up to the server, we'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want `C4` and `C6` to be in the history; that's why they rebased in the first place.

Rebase When You Rebase

If we **do** find ourself in a situation like this, Git has some further magic that might help us out. If someone on our team force pushes changes that overwrite work that we've based work on, our challenge is to figure out what is ours and what they've rewritten.

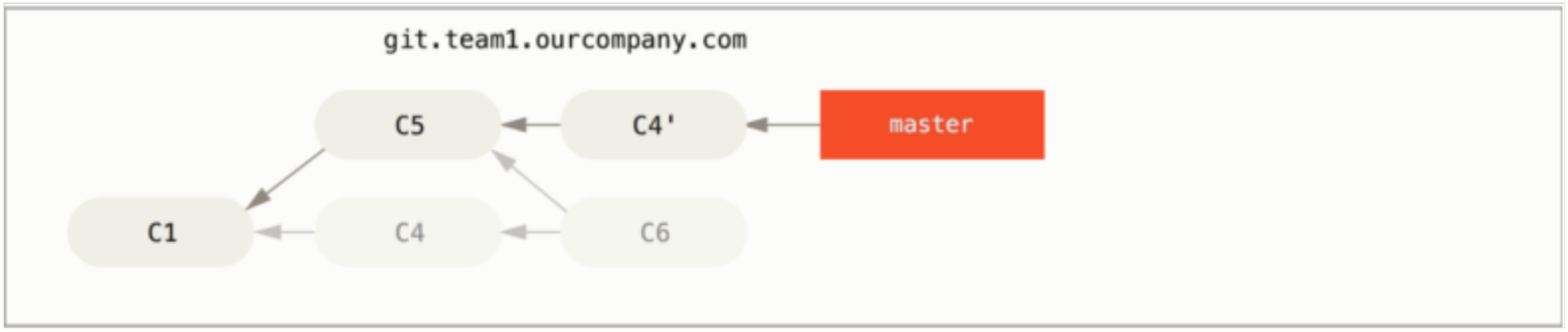
It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a "patch-id".

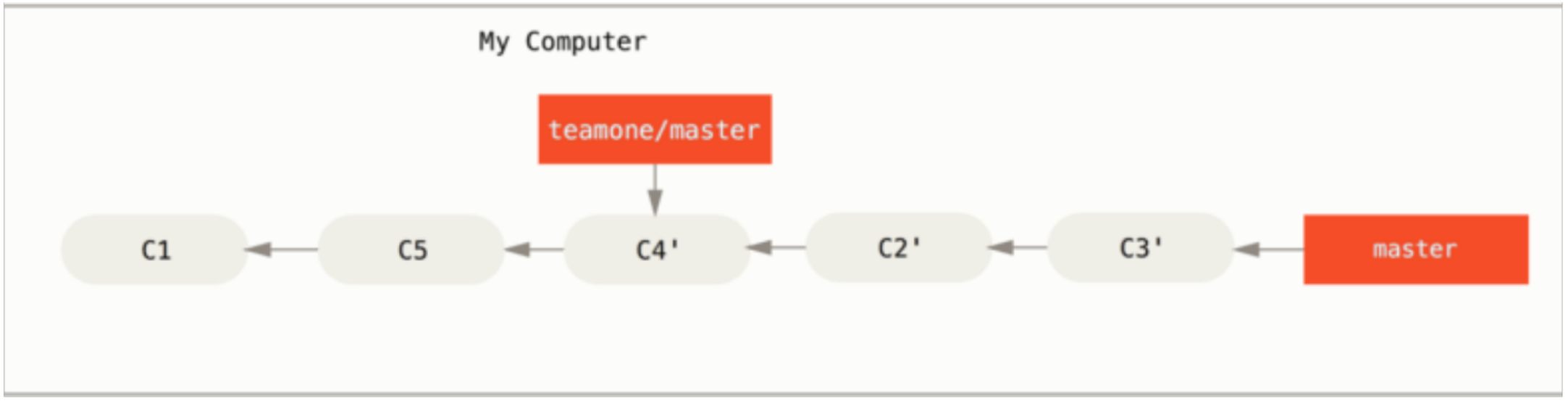
If we pull down work that was rewritten and rebase it on top of the new commits from our partner, Git can often successfully figure out what is uniquely ours and apply them back on top of the new branch.

In the previous scenario, if instead of doing a merge when we're at "Someone pushes rebased commits, abandoning commits you've based your work on" we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')
- Apply those commits to the top of `teamone/master`

So instead of the result we see in “We merge in the same work again into a new merge commit”, we would end up with something more like “Rebase on top of force-pushed rebase work.”





This only works if C4 and C4' that our partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

We can also simplify this by running a `git pull --rebase` instead of a normal `git pull`. Or we could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If we are using `git pull` and want to make `--rebase` the default, we can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.

If we treat rebasing as a way to clean up and work with commits before we push them, and if we only rebase commits that have never been available publicly, then we'll be fine. If we rebase commits that have already been pushed publicly, and people may have based work on those commits, then we may be in for some frustrating trouble, and the scorn of our teammates.

If we or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

We may be wondering which one is better.

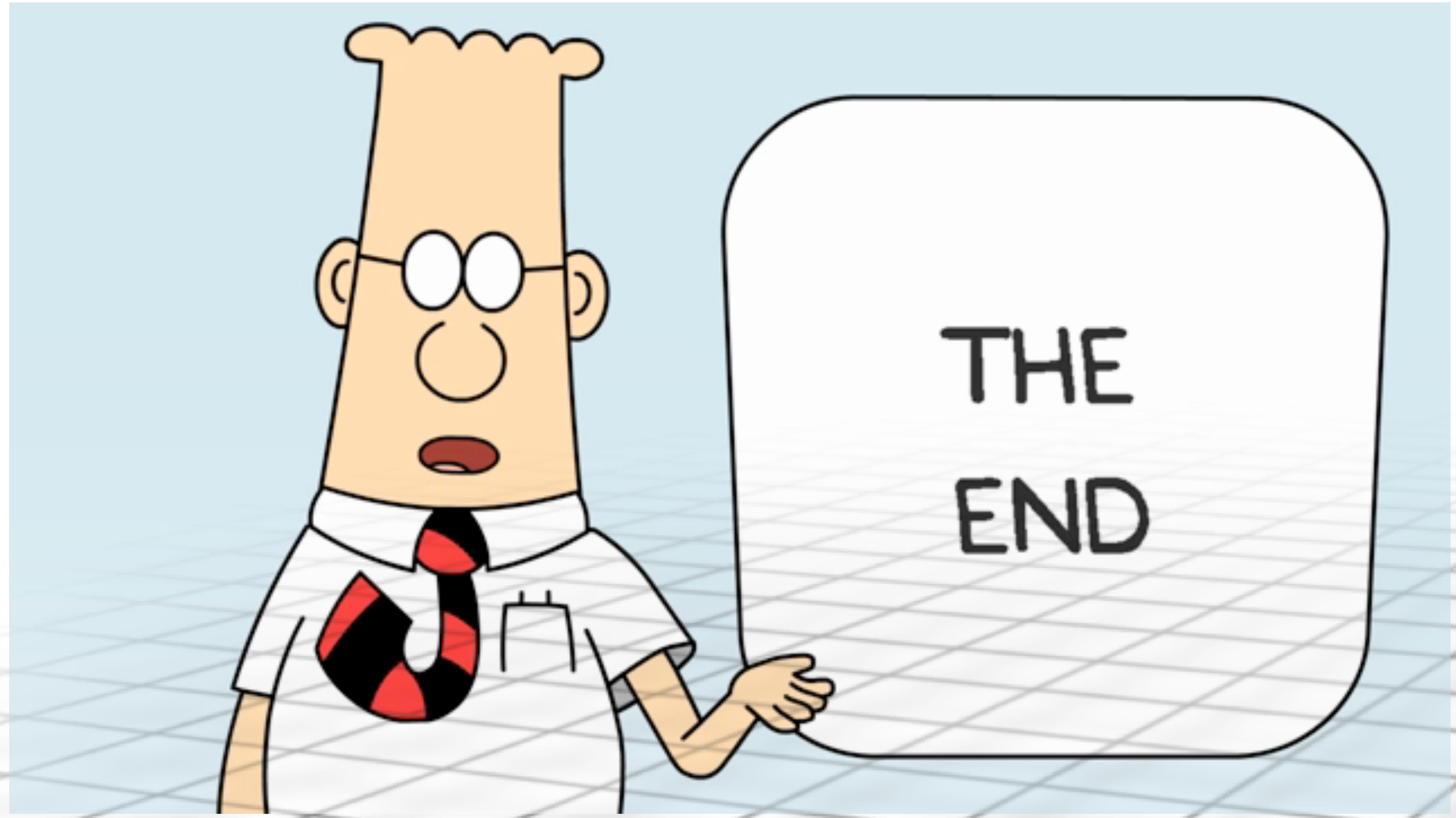
Before we can answer this, let’s step back a bit and talk about what history means.

Our repository's commit history is a **record of what actually happened**. It's a historical document, valuable in its own right, and shouldn't be tampered with. From this angle, changing the commit history is almost blasphemous; we're lying about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made**. We wouldn't publish the first draft of a book, and the manual for how to maintain our software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that's best for future readers.

To the question of whether merging or rebasing is better: hopefully we'll see that it's not that simple. Git is a powerful tool, and allows us to do many things to and with our history, but every team and every project is different. Now that we know how both of these things work, it's up to us to decide which one is best for our particular situation.

In general the way to get the best of both worlds is to rebase local changes we've made but haven't shared yet before we push them in order to clean up our story, but never rebase anything we've pushed somewhere.



출처: metachannels.com

감사합니다