Git introduction
Lecture 7 & 8

Statistics 133: Concepts in computing with data

Instructor: Jarrod Millman
GSI: Karl Kumbier

Summer 2014

Version control

The basic idea is that instead of manually trying to keep track of what changes you've made to code, data, documents, you use software to help you manage the process. This has several benefits:

- easily allowing you to go back to earlier versions
- allowing you to have multiple version you can switch between
- allowing you to share work easily without worrying about conflicts
- providing built-in backup

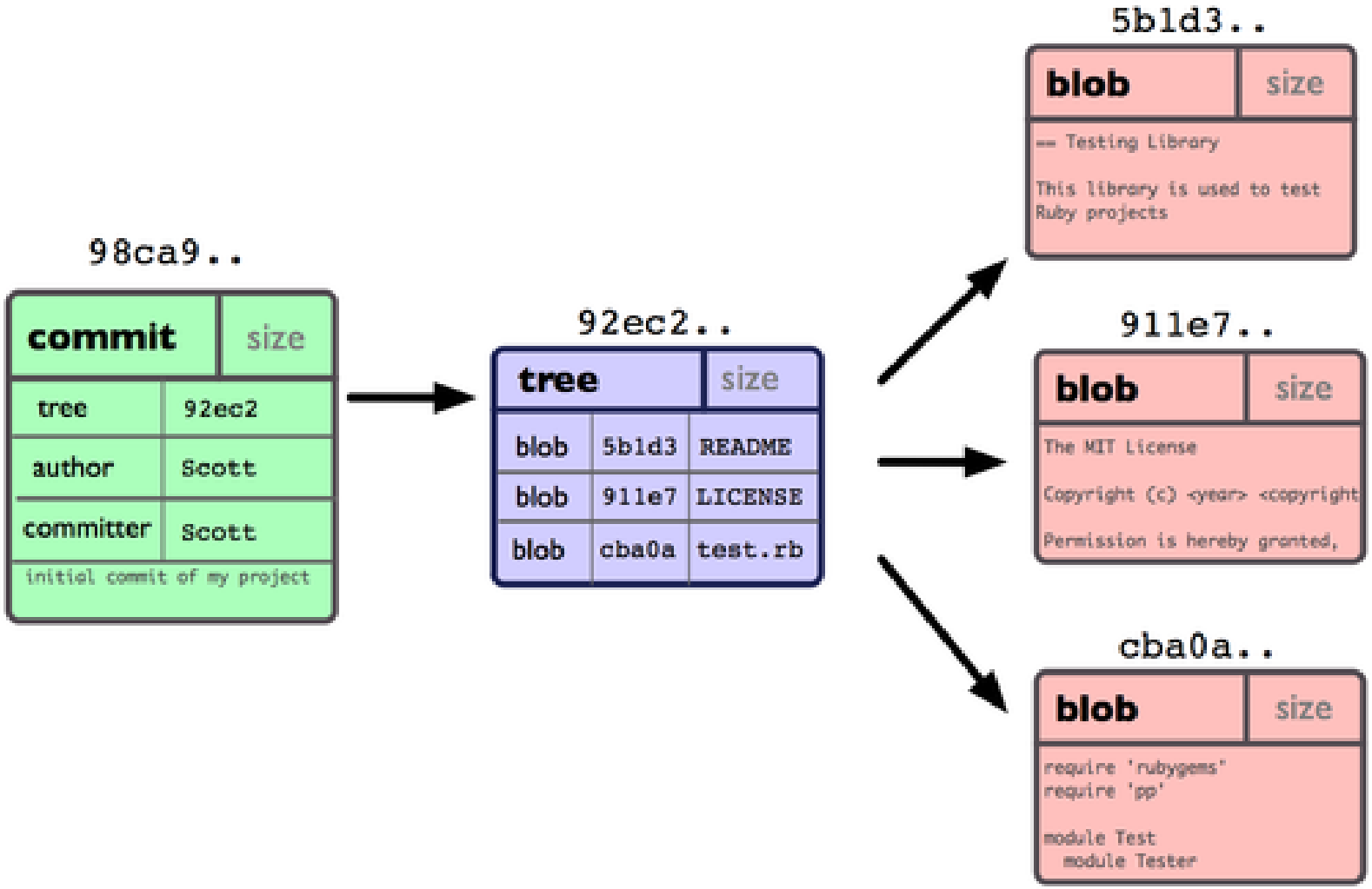
At a basic level, a simple principle is to have version numbers for all your work: code, datasets, manuscripts. Whenever you make a change to a dataset, increment the version number. For code and manuscripts, increment when you make substantial changes or have obvious breakpoints in your workflow.

Version control architectures

- Client-server (e.g., CVS, Subversion)
- Distributed (e.g., Mercurial, **Git**)

Git concepts: commit

a **snapshot** of work at a point in time



Git concepts: repository

a group of **linked** commits (DAG)

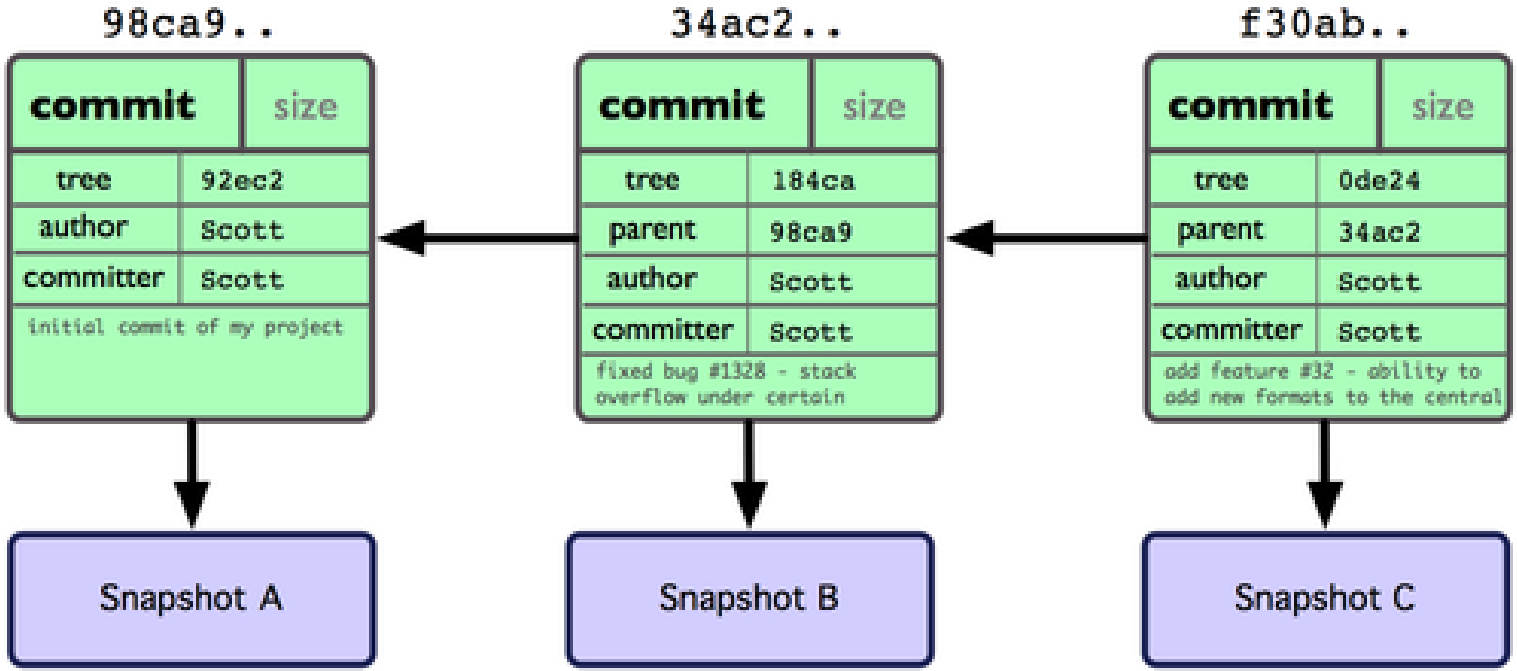


Figure: Credit: ProGit book, by Scott Chacon, CC License.

Git concepts: hash

toy "implementation":

```
library('digest')

# first commit
data1 = 'This is the start of my paper2.'
meta1 = 'date: 8/20/13'
hash1 = digest(c(data1,meta1), algo="sha1")
cat('Hash:', hash1)

# second commit, linked to the first
data2 = 'Some more text in my paper...'
meta2 = 'date: 8/20/13'
# Note we add the parent hash here!
hash2 = digest(c(data2,meta2,hash1), algo="sha1"
cat('Hash:', hash2)
```

Stage 1: Local, single-user, linear workflow

Simply type `git` (or `git help`) to see a full list of all the 'core' commands. We'll now go through most of these via small practical exercises:

git init: create an empty repository

First create an empty repository:

```
cd ~/src
git init demo
```

Let's look at what git did:

```
cd demo
ls -la
ls -l .git
```

git add: adding content to the repository

Now let's edit our first file in the test directory with a text editor... I'm doing it programatically here for automation purposes, but you'd normally be editing by hand:

```
cd ~/src/demo
echo "My first bit of text" > file1.txt
```

Now we can tell git about this new file using the `add` command:

```
git add file1.txt
```

We can now ask git about what happened with `status`:

```
git status
```

git commit: permanently record our changes in git's database

Now we are ready to commit our changes:

```
git commit -m "This is our first commit"
```

In the commit above, we used the `-m` flag to specify a message at the command line. If we don't do that, git will open the editor we specified in our configuration above and require that we enter a message. By default, git refuses to record changes that don't have a message to go along with them (though you can obviously 'cheat' by using an empty or meaningless string: git only tries to facilitate best practices, it's not your nanny).

git log: what has been committed so far

To see a log of the commits:

```
git log
```

git diff: what have I changed?

Let's do a little bit more work... Again, in practice you'll be editing the files by hand, here we do it via shell commands for the sake of automation (and therefore the reproducibility of this tutorial!)

```
echo "And now some more text..." >> file1.txt
```

And now we can ask git what is different:

```
git diff
```

The cycle of git virtue: work, add, commit, ...

```
echo "Great progress ..." >> file1.txt
git add file1.txt
git commit -m "Great progress on this matter."
```

git log revisited

First, let's see what the log shows us now:

```
git log
```

Sometimes it's handy to see a very summarized version of the log:

```
git log --oneline --topo-order --graph
```

Git supports *aliases:* new names given to command combinations. Let's make this handy shortlog an alias, so we only have to type `git slog` and see this compact log:

```
# We create our alias (this saves it in git's pe
git config --global alias.slog "log --oneline --
# And now we can use it git slog
```

git mv and rm: moving and removing files

While `git add` is used to add files to the list git tracks, we must also tell it if we want their names to change or for it to stop tracking them. In familiar Unix fashion, the `mv` and `rm` git commands do precisely this:

```
git mv file1.txt file-newname.txt
git status
```

Note that these changes must be committed too, to become permanent! In git's world, until something hasn't been committed, it isn't permanently recorded anywhere:

```
git commit -m "I like this new name better"
git slog
```

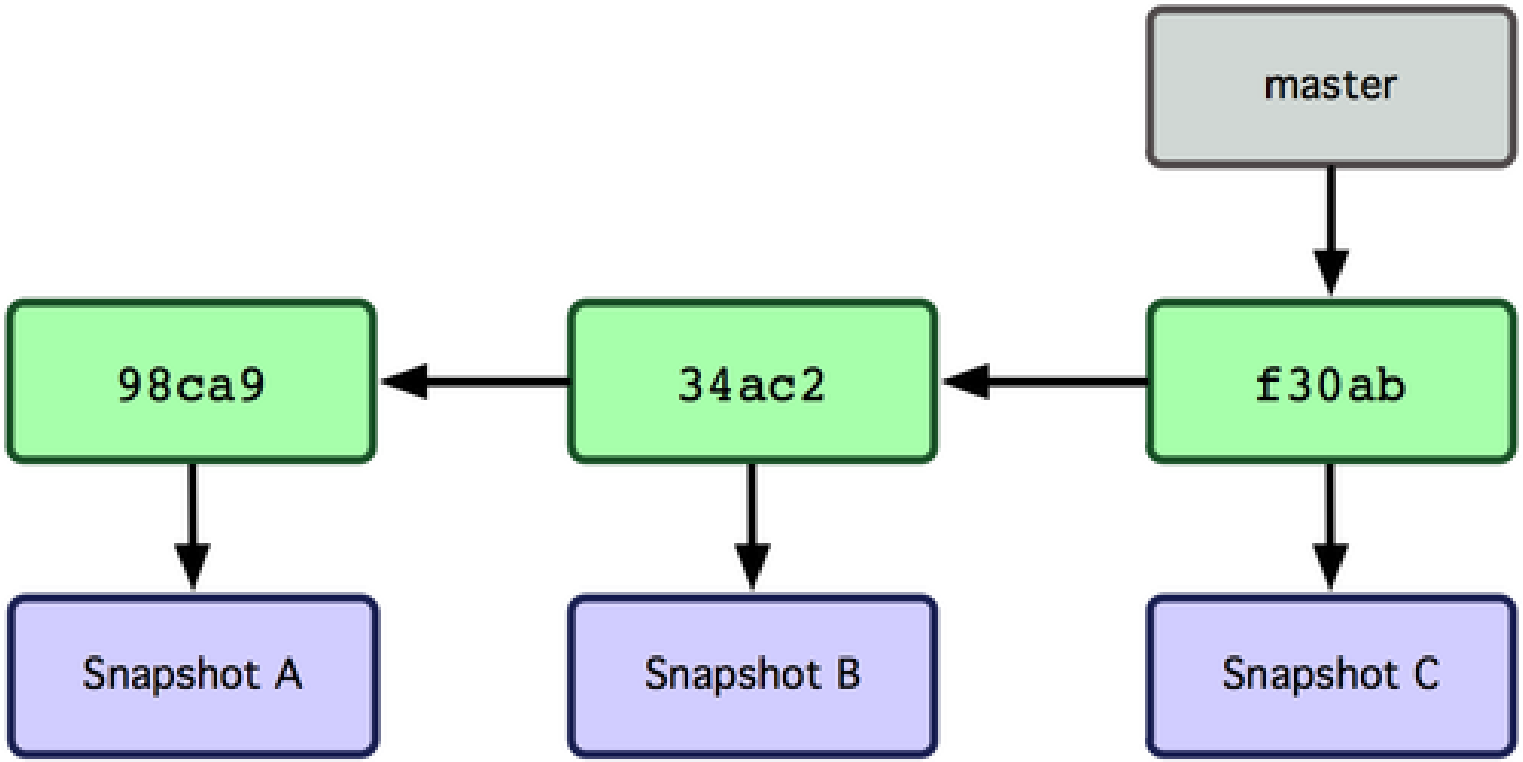
And `git rm` works in a similar fashion.

Optional: Exercise

Add a new file `file2.txt`, commit it, make some changes to it, commit them again, and then remove it (and don't forget to commit this last step!).

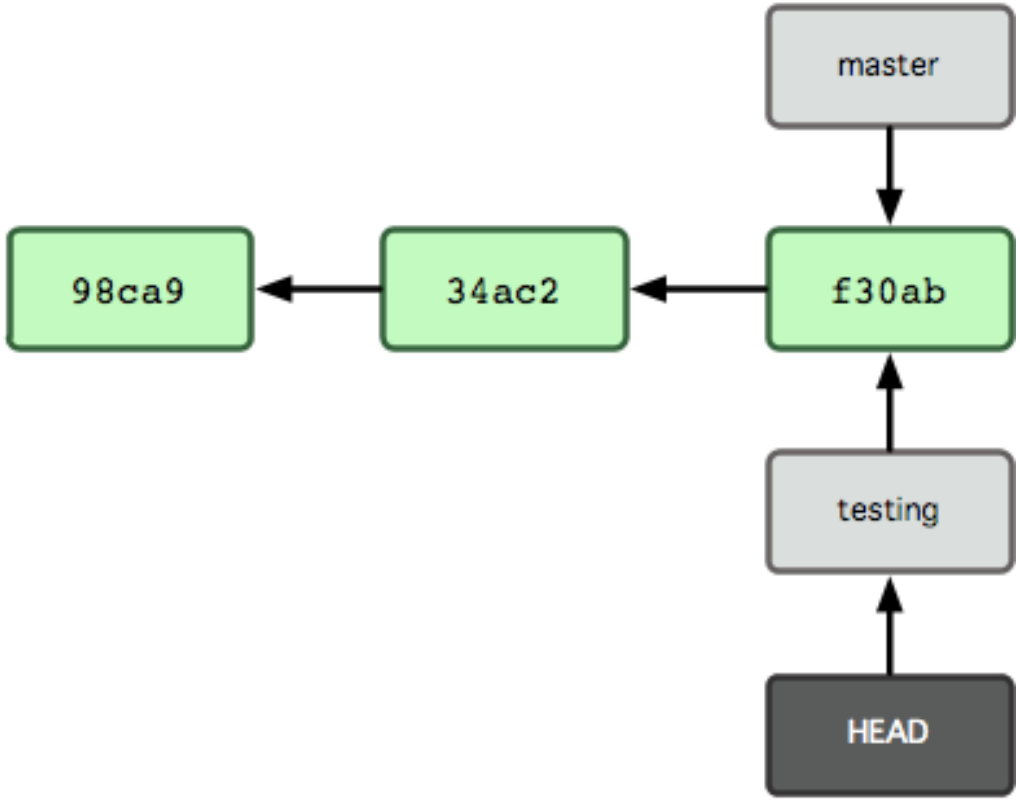
Local user, branching: the concept

What is a branch? Simply a *label for the 'current' commit in a sequence of ongoing commits*:



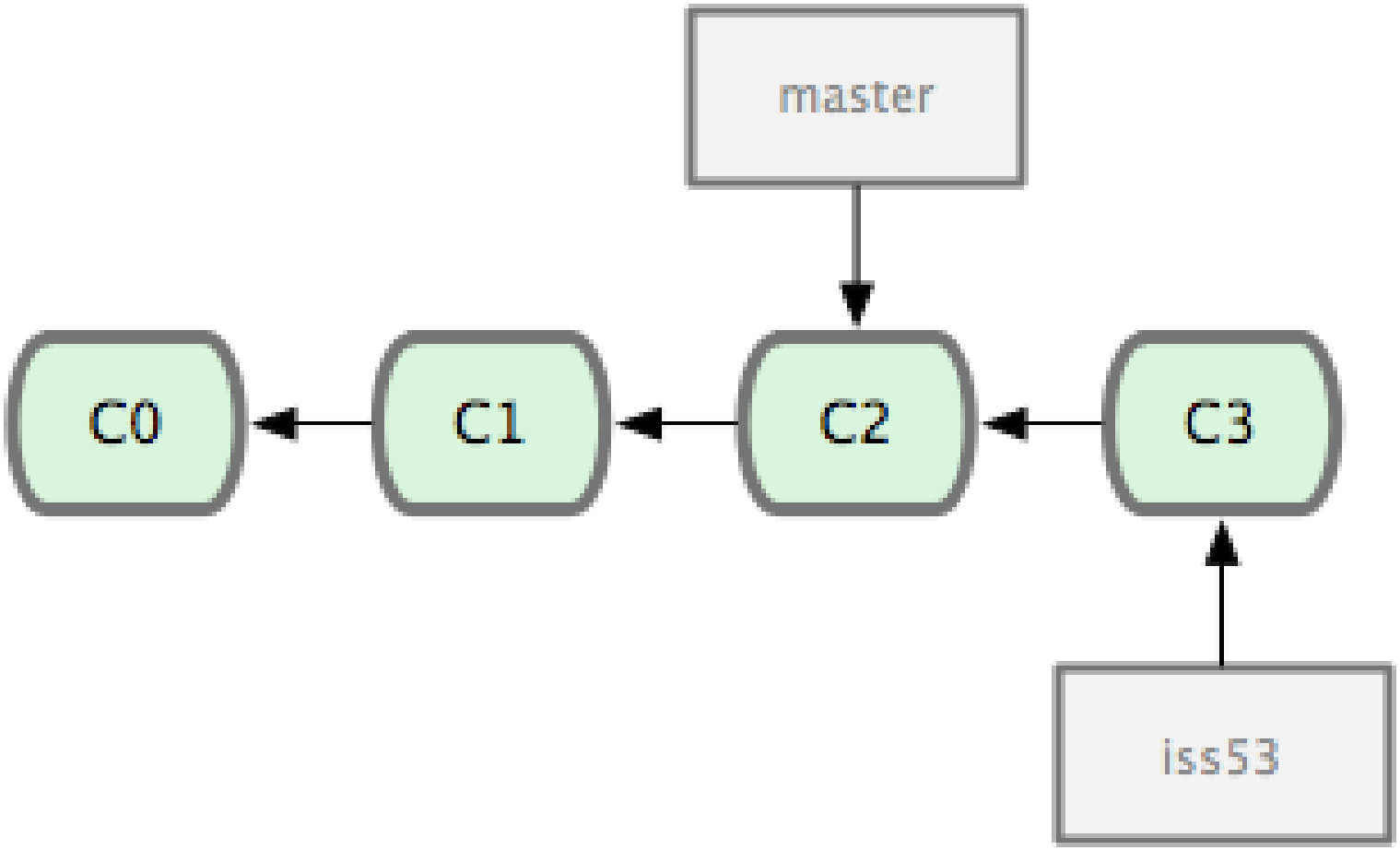
Local user, branching: the concept

There can be multiple branches alive at any point in time; the working directory is the state of a special pointer called HEAD. In this example there are two branches, *master* and *testing*, and *testing* is the currently active branch since it's what HEAD points to:



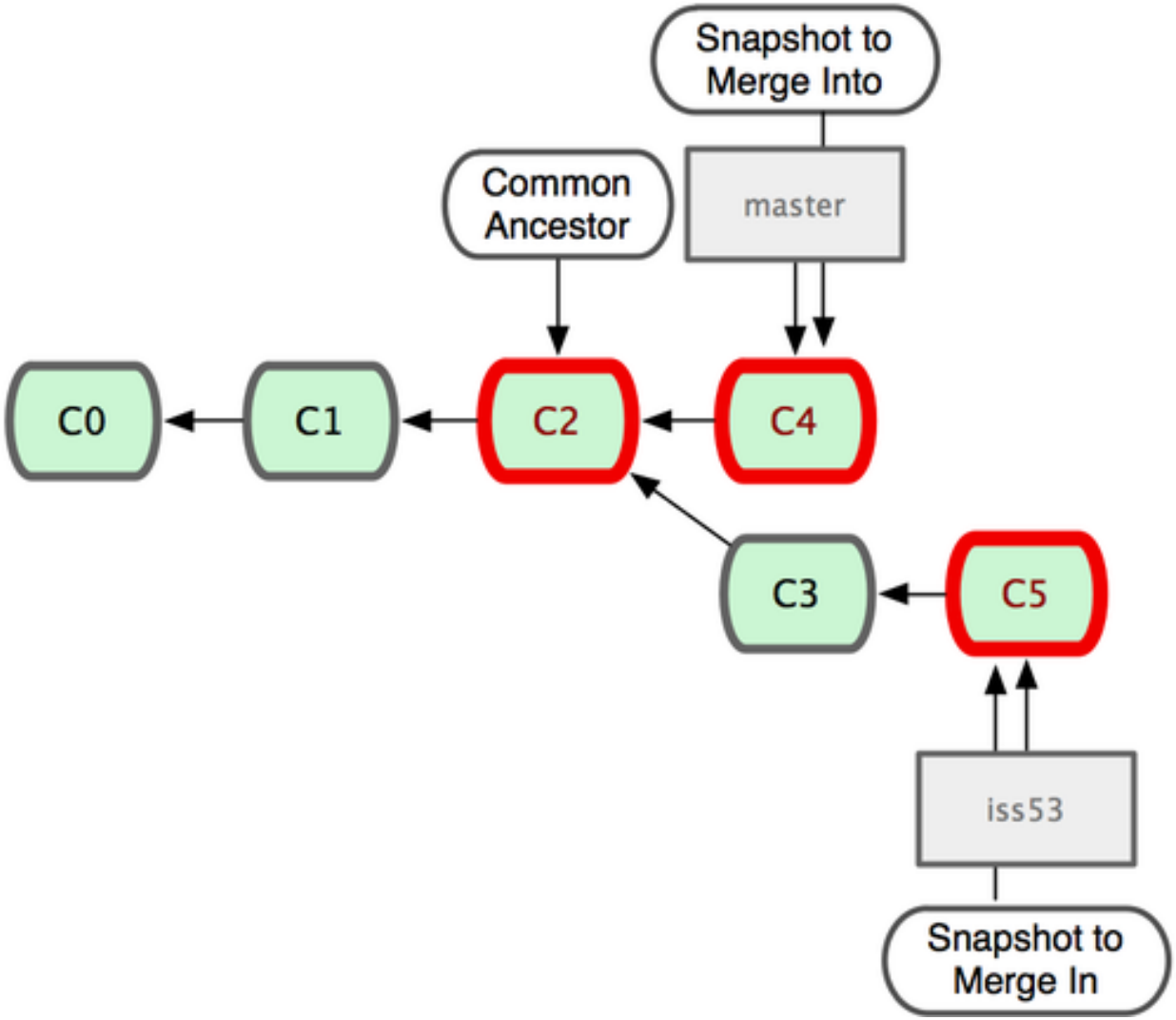
Local user, branching: the concept

Once new commits are made on a branch, HEAD and the branch label move with the new commits:



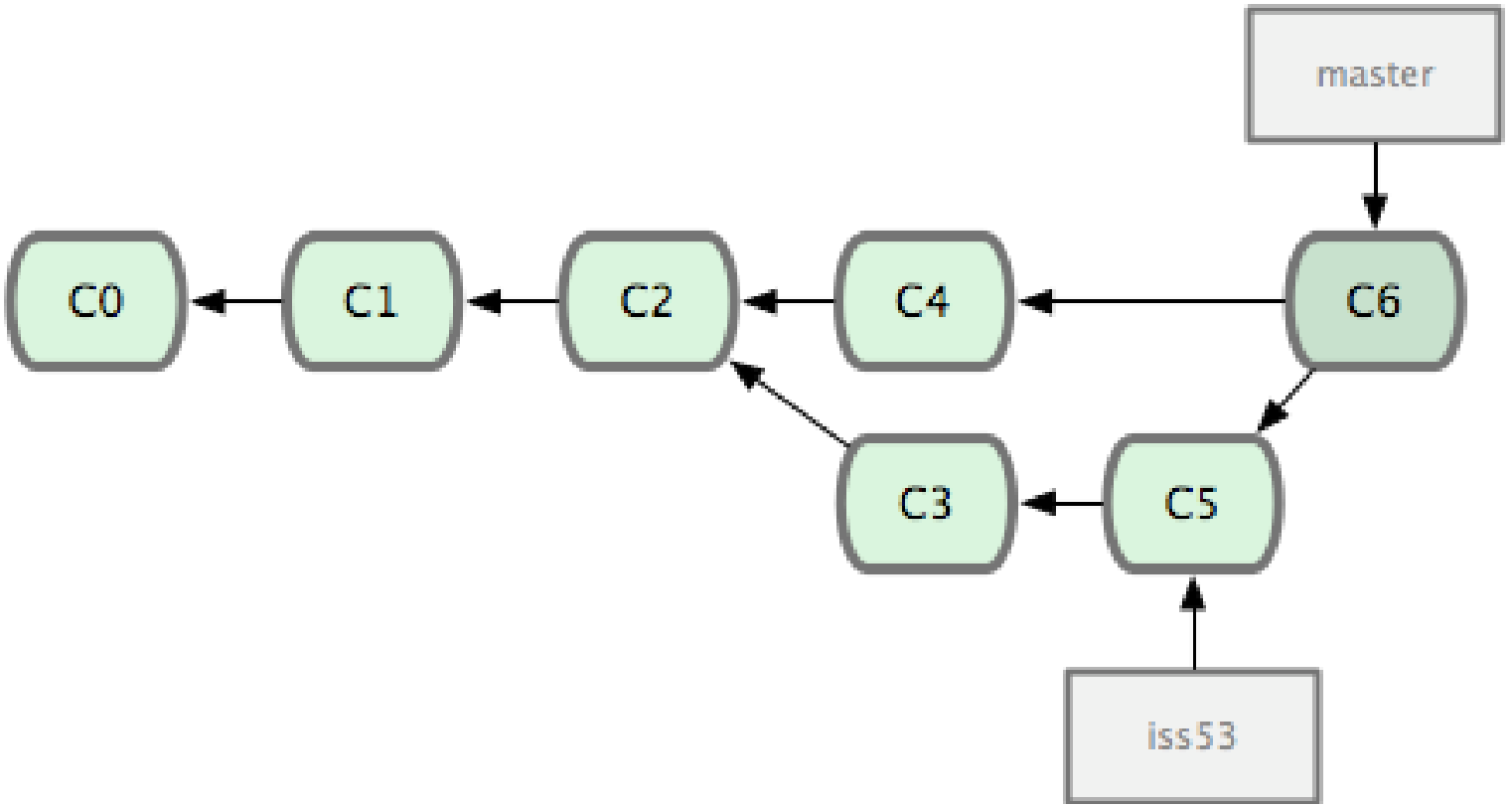
Local user, branching: the concept

This allows the history of both branches to diverge:



Local user, branching: the concept

But based on this graph structure, git can compute the necessary information to merge the divergent branches back and continue with a unified line of development:



Local user, branching: an example

Let's now illustrate all of this with a concrete example. Let's get our bearings first:

```
git status
ls
```

We are now going to try two different routes of development: on the `master` branch we will add one file and on the `experiment` branch, which we will create, we will add a different one. We will then merge the experimental branch into `master`.

Local user, branching: an example

Create and work on an experimental branch:

```
git branch experiment
git checkout experiment
echo "Some crazy idea" > experiment.txt
git add experiment.txt
git commit -m "Trying something new"
git slog
```

Local user, branching: an example

Work on the master branch:

```
git checkout master
git slog
echo "Work goes on in master..." >> file-newname
git add file-newname.txt
git commit -m "The mainline keeps moving"
git slog
```

Local user, branching: an example

Now merge experimental branch:

```
ls
git merge experiment
git slog
```

Using remotes as a single user

We are now going to introduce the concept of a *remote repository*: a pointer to another copy of the repository that lives on a different location. This can be simply a different path on the filesystem or a server on the internet.
For this discussion, we'll be using remotes hosted on the GitHub.com service, but you can equally use other services like BitBucket or Gitorious as well as host your own.

```
git remote -v
```

Since the above cell didn't produce any output after the `git remote -v` call, it means we have no remote repositories configured. We will now proceed to do so.

Using remotes as a single user

Once logged into GitHub, go to the new repository page and make a repository called `test`. Do **not** check the box that says `Initialize this repository with a README`, since we already have an existing repository here. That option is useful when you're starting first at Github and don't have a repo made already on a local computer.

Using remotes as a single user

We can now follow the instructions from the next page:

```
git remote add origin git@github.com:jarrodmillm
git push -u origin master
```

Let's see the remote situation again:

```
git remote -v
```

We can now see this repository publicly on github.

Using remotes as a single user

Let's see how this can be useful for backup and syncing work between two different computers. I'll simulate a 2nd computer by working in a different directory...

```
cd ~/src/
# Here I clone my 'test' repo but with a differe
# to simulate a 2nd computer
git clone git@github.com:jarrodmillman/test.git
cd test2
pwd
git remote -v
```

Using remotes as a single user

Let's now make some changes in one 'computer' and synchronize them on the second.

```
cd ~/src/test2
# working on computer #2
echo "More new content on my experiment" >> expe
git add experiment.txt
git commit -m "More work, on machine #2"
```

Using remotes as a single user

Now we put this new work up on the github server so it's available from the internet:

```
# working on computer #2
git push
```

Now let's fetch that work from machine #1:

```
cd ~/src/demo
git pull
```

An important aside: conflict management

While git is very good at merging, if two different branches modify the same file in the same location, it simply can't decide which change should prevail. At that point, human intervention is necessary to make the decision. Git will help you by marking the location in the file that has a problem, but it's up to you to resolve the conflict. Let's see how that works by intentionally creating a conflict.
We start by creating a branch and making a change to our experiment file:

```
git branch trouble
git checkout trouble
echo "This is going to be a problem..." >> exper
git add experiment.txt
git commit -m "Changes in the trouble branch"
```

An important aside: conflict management

And now we go back to the master branch, where we change the *same* file:

```
git checkout master
echo "More work on the master branch..." >> expe
git add experiment.txt
git commit -m "Mainline work"``
```

So now let's see what happens if we try to merge the `trouble` branch into `master`:

```
git merge trouble
```

Let's see what git has put into our file:

```
cat experiment.txt
```

An important aside: conflict management

At this point, we go into the file with a text editor, decide which changes to keep, and make a new commit that records our decision. To automate my edits, I use the `sed` command:

```
sed -i '/^</d' experiment.txt
sed -i '/^>/d' experiment.txt
sed -i '/^=/d' experiment.txt
```

An important aside: conflict management

I've now made the edits, in this case I decided that both pieces of text were useful, so I just accepted both additions.

```
cat experiment.txt
```

Let's then make our new commit:

```
git add experiment.txt
git commit -m "Completed merge of trouble, fixin
git slog
```

An important aside: conflict management

Note: While it's a good idea to understand the basics of fixing merge conflicts by hand, in some cases you may find the use of an automated tool useful. Git supports multiple merge tools: a merge tool is a piece of software that conforms to a basic interface and knows how to merge two files into a new one. Since these are typically graphical tools, there are various to choose from for the different operating systems, and as long as they obey a basic command structure, git can work with any of them.

Collaborating on github with a small team

Single remote with shared access: we are going to set up a shared collaboration with one partner (the person sitting next to you). This will show the basic workflow of collaborating on a project with a small team where everyone has write privileges to the same repository.
Note for SVN users: this is similar to the classic SVN workflow, with the distinction that commit and push are separate steps. SVN, having no local repository, commits directly to the shared central resource, so to a first approximation you can think of `svn commit` as being synonymous with `git commit; git push`.

Collaborating on github with a small team

We will have two people, let's call them Alice and Bob, sharing a repository. Alice will be the owner of the repo and she will give Bob write privileges.
We begin with a simple synchronization example, much like we just did above, but now between *two people* instead of one person. Otherwise it's the same:

- Bob clones Alice's repository.
- Bob makes changes to a file and commits them locally.
- Bob pushes his changes to github.
- Alice pulls Bob's changes into her own repository.

Collaborating on github with a small team

Next, we will have both parties make non-conflicting changes each, and commit them locally. Then both try to push their changes:

- Alice adds a new file, `alice.txt` to the repo and commits.
- Bob adds `bob.txt` and commits.
- Alice pushes to github.
- Bob tries to push to github. What happens here?

The problem is that Bob's changes create a commit that conflicts with Alice's, so git refuses to apply them. It forces Bob to first do the merge on his machine, so that if there is a conflict in the merge, Bob deals with the conflict manually (git could try to do the merge on the server, but in that case if there's a conflict, the server repo would be left in a conflicted state without a human to fix things up). The solution is for Bob to first pull the changes (pull in git is really fetch+merge), and then push again.

Learning Git

- Git for Scientists: A Tutorial
- Gitwash: workflow for scientific Python projects
- Git branching demo