Git: Working with Remotes

Joseph Hallett

January 11, 2023

What's all this about?

Last time we spoke about how to use *Git* to manage changes to local files.

This time

Rather than making *our own* new repo:

- Let's clone someone else's!
- And let's share those changes with others!

What do we mean by *decentralized*?

In the first lecture we said Git was a *decentralized version control system*

- As opposed to a *centralized* one like SVN/CVS

What this means in practice is that your local repo *should* have the complete history of the repo.

- And should be able to function as a master copy of the repo.
- (Again this is a simplification but go with it…)

Let's pretend Alice has a Git repo of their course groupwork in `~alice/coursework/`

- Bob wants to collaborate with Alice and get their own copy

So Bob runs...

```
$ git clone ~alice/coursework ~bob/coursework
Cloning into '~bob/coursework'...
done.

$ cd ~bob/coursework; git log --oneline
af3818c States true facts about this course
7e # make coursework
cc -O2 -pipe    -o coursework coursework.c

 # ./coursework
Softwaer tools is cool!
Hello World!
b311c First draft of the coursework

$ make coursework
cc -O2 -pipe    -o coursework coursework.c

$ ./coursework
Softwaer tools is cool!
Hello World!
```

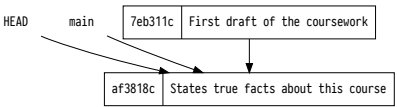
Oh no: a mistake!

Bob can fix that for Alice!

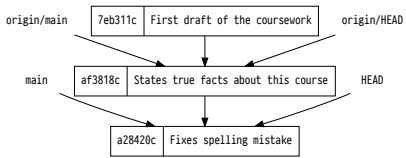
```
$ ed coursework.c
124
4c
    printf("Software tools is cool!\n");
.
wq
124

$ git add coursework.c

$ git commit -m "Fixes spelling mistake"
[main a28420c] Fixes spelling mistake
 1 file changed, 1 insertion(+), 1 deletion(-)
```

So now

On Alice's repo



On Bob's repo



So how do we get Bob's changes back to Alice?

There are multiple ways!

Bob can send Alice their changes patch based

Alice can pull Bob's changes pull based

Patch based approach (Bob's end)

This is how the *Linux Kernel* and many other open source projects manage commits.
Bob starts by preparing a *patch*

```
$ git format-patch origin/main \
      --to=alice@bristol.ac.uk
0001-Fixes-spelling-mistake.patch
```

And sends it to Alice

- (check out `git send-email`!)

```
From a28420cd5c45d06c9a51625d5a03c37bb77e2ca9 Mon Sep 17 00:00:
From: bob <bob@bristol.ac.uk>
Date: Tue, 22 Nov 2022 11:53:04 +0000
Subject: [PATCH] Fixes spelling mistake
To: alice@bristol.ac.uk

---
 coursework.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/coursework.c b/coursework.c
index 2e191f8..8927b2f 100644
--- a/coursework.c
+++ b/coursework.c
@@ -1,7 +1,7 @@
 #include <stdio.h>

 int main(void) {
-    printf("Softwaer tools is cool!\n");
+    printf("Software tools is cool!\n");
     printf("Hello World!\n");
     return 0;
 }
--
2.38.1
```

Patch based approach (Alice's end)

Alice reviews Bob's patch and, if they like it... applies it to their tree.

```
$ git am ../bob/0001-Fixes-spelling-mistake.patch
Applying: Fixes spelling mistake

$ git log --oneline
575dcde Fixes spelling mistake
af3818c States true facts about this course
7eb311c First draft of the coursework
```

The ID is different however to Bob's tree for the latest commit

- (because *Alice* commited it).

If Bob wants to keep IDs in sync with Alice they need to re-`clone`

- (or `git fetch origin`).

Aside

Technically `git am` is actually a couple of git commands rolled into one...

- First it runs `git apply` with the patch to stage all the changes it will make
- Then it runs `git commit` with the commit message also supplied in the patch

There are a *lot* of git commands that are really lower level commands chained together—watch out for them!

The alternative

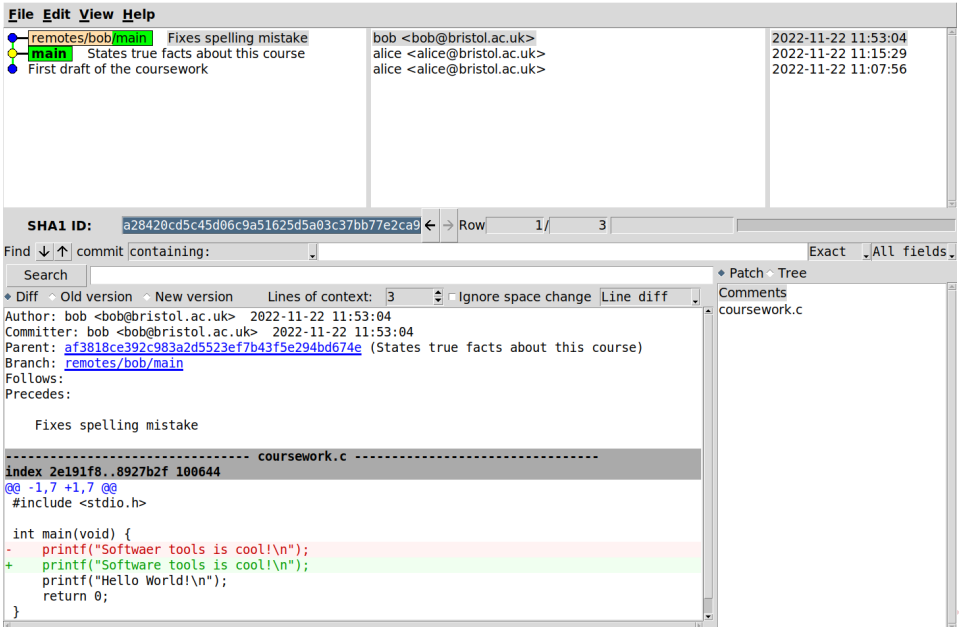
The patch and email-based workflow is *great* when you're working on open-source or when you want to control tightly how integrating other people's work goes.
The alternative is to let Git do the work for you and trust the other person.
Bob tells Alice they fixed a mistake.

- Alice adds Bob's repo as a `remote`

```
$ git remote add bob ~bob/coursework
```

```
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 255 bytes | 127.00 KiB/s, done.
From ~bob/coursework
 * [new branch]      main       -> bob/main
```

Alice inspects Bob's changes...



And if they're happy...

```
$ git pull bob main
From ~bob/coursework
 * branch            main       -> FETCH_HEAD
Updating af3818c..a28420c
Fast-forward
 coursework.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git log | cat
commit a28420cd5c45d06c9a51625d5a03c37bb77e2ca9
Author: bob <bob@bristol.ac.uk>
Date:   Tue Nov 22 11:53:04 2022 +0000

    Fixes spelling mistake

commit af3818ce392c983a2d5523ef7b43f5e294bd674e
Author: alice <alice@bristol.ac.uk>
Date:   Tue Nov 22 11:15:29 2022 +0000

    States true facts about this course
```

Aside

Again the `git pull` command is really a composite. The following is equivalent:

```
$ git fetch bob

$ git merge --ff bob/main
Updating af3818c..a28420c
Fast-forward
 coursework.c | 2 +-
 1 file changed, 1 insertion(+)
```

Also note that this way the Git commit IDs are kept in sync `:-D`.

But what about Github?!

Github gives you a *centralised* remote (called a *forge*):

- You can sign up for an account
- Set up access for users
- And then centrally send commits to everyone with the `git push` command
- Can host a project page and build infrastructure too

Github is owned by Microsoft:

- Some people don't like that
- Some questionable behaviour surrounding AI and Open Source

Alternatives:

`https://bitbucket.org` Owned by Atlassian

`https://gitlab.com` Can self-host

`https://sr.ht` Owned by *Drew DeVault*... costs money

Self host? All you need is an SSH server...

- (Search for *bare repositories* to find out how)

To use a forge

In Bob's repo

```
$ git remote set-url origin \
  git@github.com:alice/coursework

$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

$ git push
Everything up-to-date
```

In Alice's repo

```
$ git remote -v
remote    git@github.com:alice/coursework (fetch)
remote    git@github.com:alice/coursework (push)

$ git pull remote main
From git@github.com:alice/coursework
 * branch            main       -> FETCH_HEAD
Updating af3818c..a28420c
Fast-forward
 coursework.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

SSH keys

To use a forge you *usually* need to use SSH to authenticate. This means you need to use *keys*:

On Macs/Linux/BSD

To generate a key:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/joseph/.ssh/id_rsa): /home/joseph/.ssh/github
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/joseph/.ssh/github
Your public key has been saved in /home/joseph/.ssh/github.pub
The key fingerprint is:
SHA256:L1Xg2W2I1EgXocqPpuSefRbYlgop/H2x5TXl4EKs0qA joseph@bristol.ac.uk
The key's randomart image is:
+---[RSA 3072]----+
|         .+o+o   |
|         o.B.o   |
|          =.+ o  |
|        ... .o.. .|
|   .   oS=.+ . + |
|    o E o=O o + .|
|     o.o++o* o . |
|     o.=o.= .    |
|     .= .+       |
+----[SHA256]-----+
```

In your `~/.ssh/config` file:

```
Host github.com
User git
IdentityFile ~/.ssh/github
```

Make sure you upload the public `.pub` key not the private one to Github!

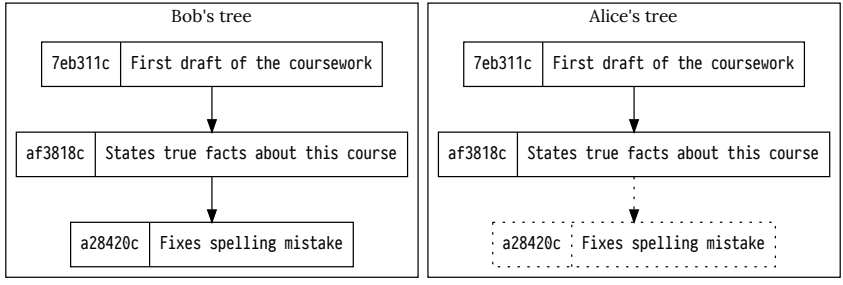
And test with:

```
$ ssh git@github.com
PTY allocation request failed on c
Hi uob-jh! You've successfully aut
Connection to github.com closed.
```

What happens when things go wrong?

When *Alice* pulled *Bob*'s changes earlier they could be *fast-forwarded*.

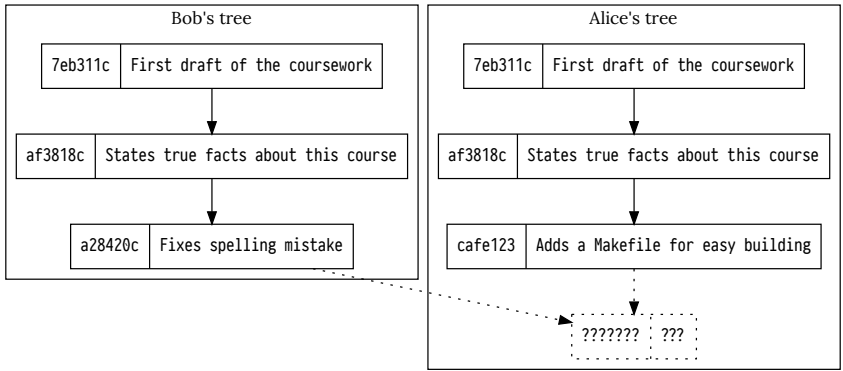
- This means that the changes could be pulled straight across and copied into Alice's tree.



Busy, busy, busy...

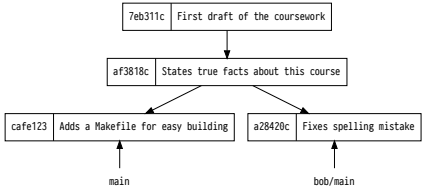
What about if *Alice* has been busy and made some commits of their own.

- The fast forward can't happen now becuase the trees have *diverged*



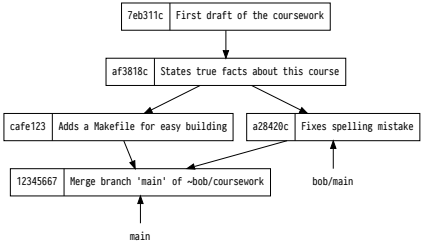
Merging

From Alice's point of view this is what the trees look like




```
$ git merge --no-ff bob/main
hint: Waiting for your editor to close the file...
Merge made by the 'ort' strategy.
 Makefile | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Makefile
```

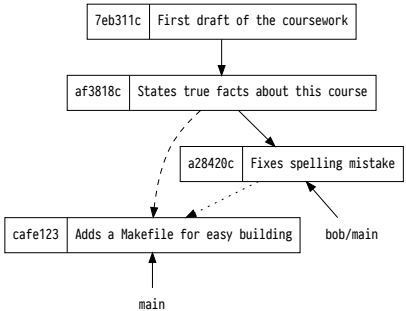
The simplest approach is to do a *merge* and add a commit explicitly merging the changes from both paths of the tree



(But normally it'll be smart and spot that you changed different files and still do the *fast-forward*...)

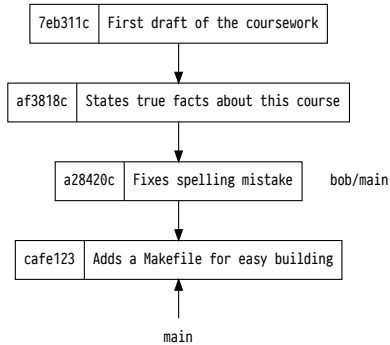
Rebasing

Alternatively Alice could do a bit of time-travelling... Lets *pretend* that Alice's commit came after Bob's:




```
$ git rebase bob/main
Successfully rebased and updated refs/heads/main.
```

Then we can *fast-forward* as before through *Bob*'s change and then replay *Alice*'s new commit after.



Now we get a nice neat straight line tree again!

Merging vs Rebasing

Merging is simpler *conceptually*...

- ...but messy

Rebasing is neater

- ...but complicated and prone to failure
- There are some other neat tricks you can do that make it much better (e.g. `git bisect`)

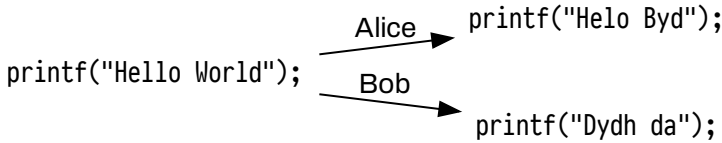
You (and your employers) will have an opinion

- Do that.
- It really doesn't matter
- (I slightly prefer the merge version but I switch back and forth...)

So far easy!

So far our merges have been *easy*.

- Alice and Bob have made edits to different files
- Changes have all been able to be done by Git automatically.
  What happens if Alice and Bob *both change the same lines in the same file*?




```
$ git merge bob/main
Auto-merging coursework.c
CONFLICT (content): Merge conflict in coursework.c
Automatic merge failed; fix conflicts and then commit the result.
```

Lets fix the conflict

Git has discovered there are two sets of changes and it can't work out which is the one to go with...
If we *follow Git's instructions* `coursework.c` looks like:

```
#include <stdio.h>

int main(void) {
<<<<<<< HEAD
  printf("Helo Byd\n");
=======
  printf("Dydh da\n");
>>>>>>> bob/main
  return 0;
```

Fix up the file and then run `git add` / `git commit` when it looks good...

- Don't just delete one side of it.
- ...Seriously... I've seen people fired for that.

```
$ git add coursework
$ git commit
[main 16d3aa6] Merge remote-tracking branch 'bob/main'
```

Wrap up

- Use `git remote` and `git clone` to work with other people
- Use `git fetch` or `git pull` or patch files to get other peoples work
- Use `git merge` or `git rebase` to integrate changes
- Use `git push` to send work back to a forge
- Merge conflicts are a pain but you have to deal with them

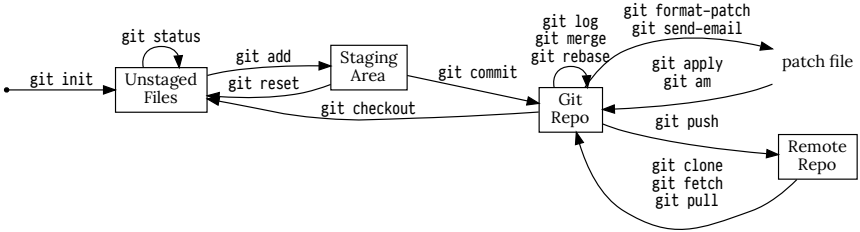
Merge tools

If you find yourself dealing with merge conflicts regularly... there are tools that help you work with them

`https://meldmerge.org` Good tool for dealing with merges

(I use Emacs.)

Commands


git init
Unstaged Files
git status
git add
git reset
Staging Area
git commit
git checkout
Git Repo
git log
git merge
git rebase
git format-patch
git send-email
git apply
git am
patch file
git push
Remote Repo
git clone
git fetch
git pull