Git for D2

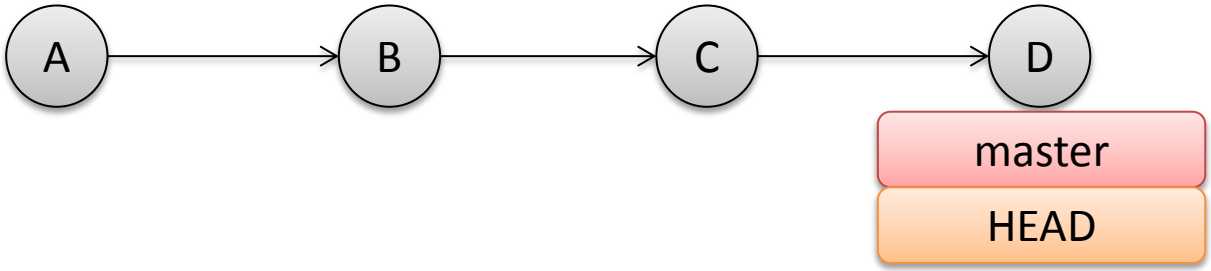
What is Git?

- Git is, at least, a version control tool that keeps track of what changes are made to a project and by whom, so you can revert if something goes wrong.
- More than that, though, it helps with group collaborative work by giving people the power to “branch” the project and work on their own version, without interfering with the main copy.

How does Git work?

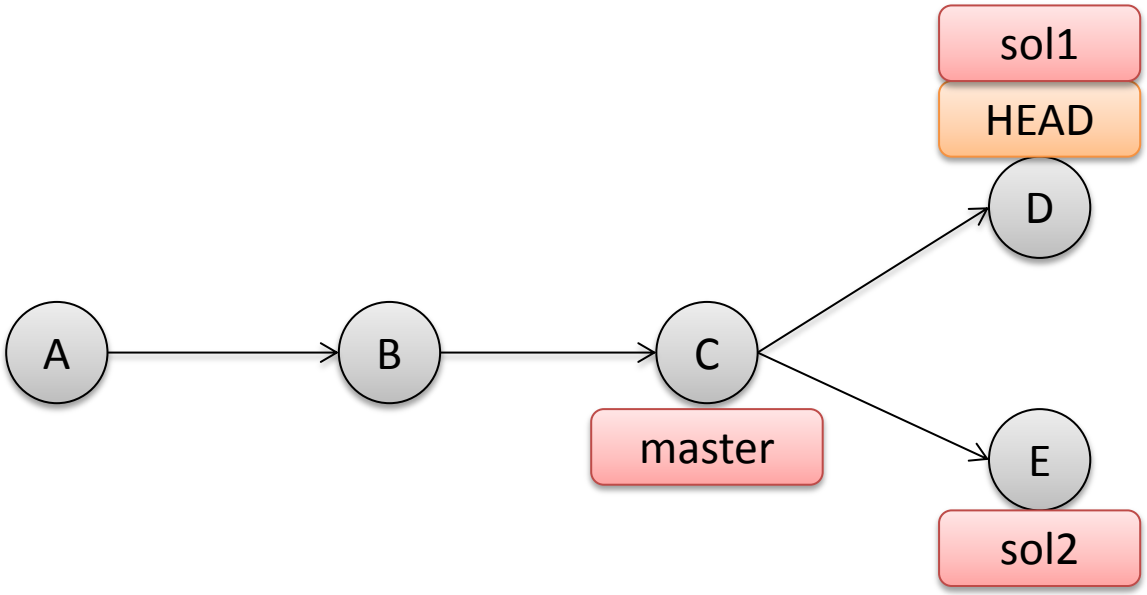
- The basic object in Git is a commit. A commit is a snapshot of the project database relative to a previous commit.
- For instance if we had two snapshots of the project, “A” and “B”, and the only thing that changed was that we added the number “1” to the end of a file, then the “B” commit would simply say “I’m A, except with an extra 1 at the end of this file”

- This creates a string of commits, all referring to one another.



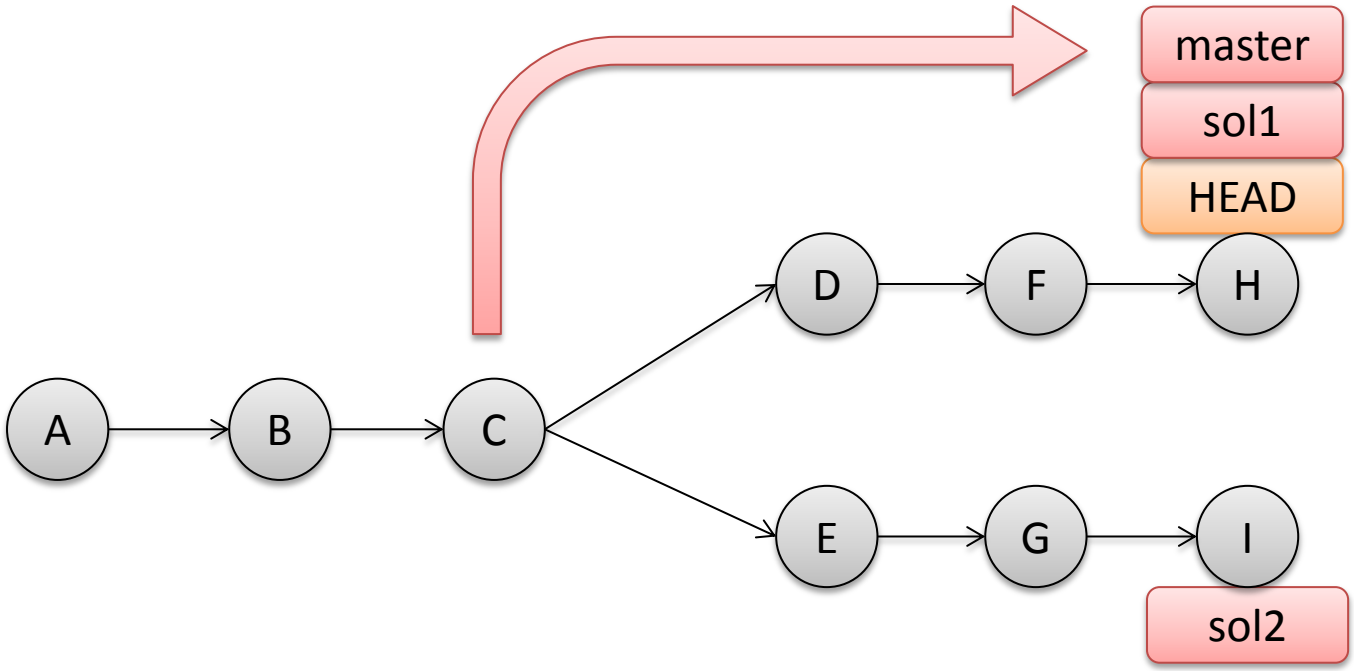
- We also have these labels called “refs”, which keep track of working points in the project. “master” is the default ref, and HEAD is the currently selected ref that you are seeing when you look at the project files.

- This is one of the real strengths of Git! We don't have to keep going straight ahead like this, we can branch.



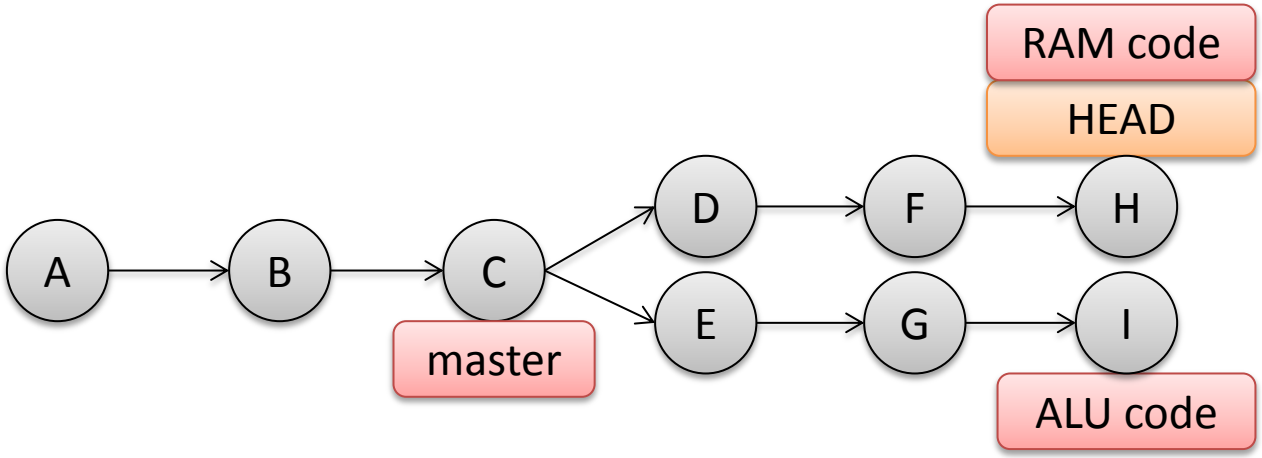
- In this example, we came across a problem in commit C. We create 2 branches from this point, each with a new ref, to explore 2 different solutions. We also need the HEAD ref somewhere, so for completeness sake our current working ref is sol1

- Eventually, when one solution works out, we can move the master head up to meet it

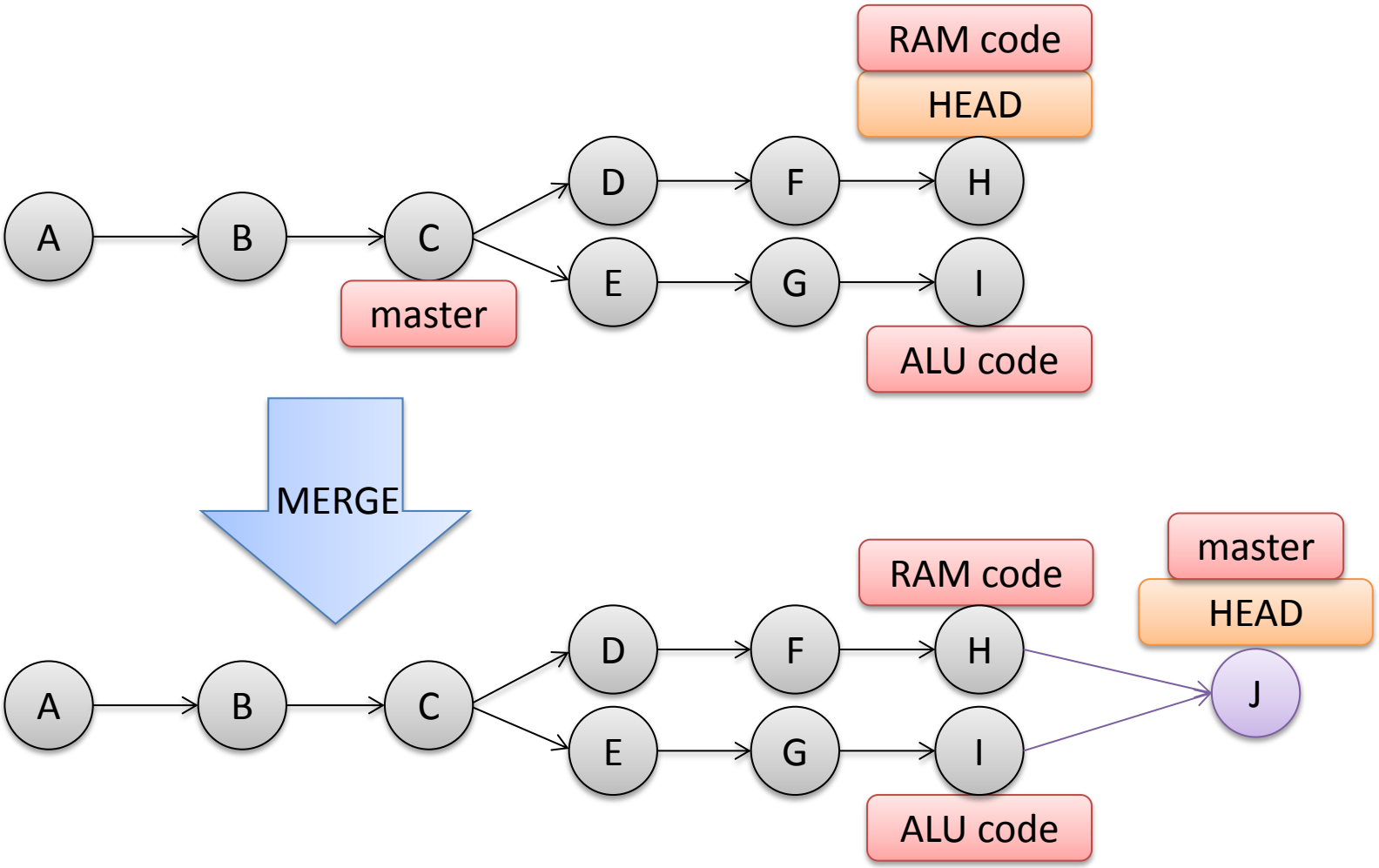


- This is pretty usual in git, and is one of the nice benefits of branching.

- We could also be working on two braches that work on different things, like two separate chunks of code



- In this case, we don’t want to just abandon a branch, we want to consolidate them into one branch. This is done with a merge



- Obviously this will run into problems if both branches have edited the same files, so be careful! It's still very useful for adding branches together.

Other stuff:

- You can move the HEAD (your current work-point) between refs using the “checkout” command. Checking out the master ref, for instance, simply moves your HEAD to the master ref.

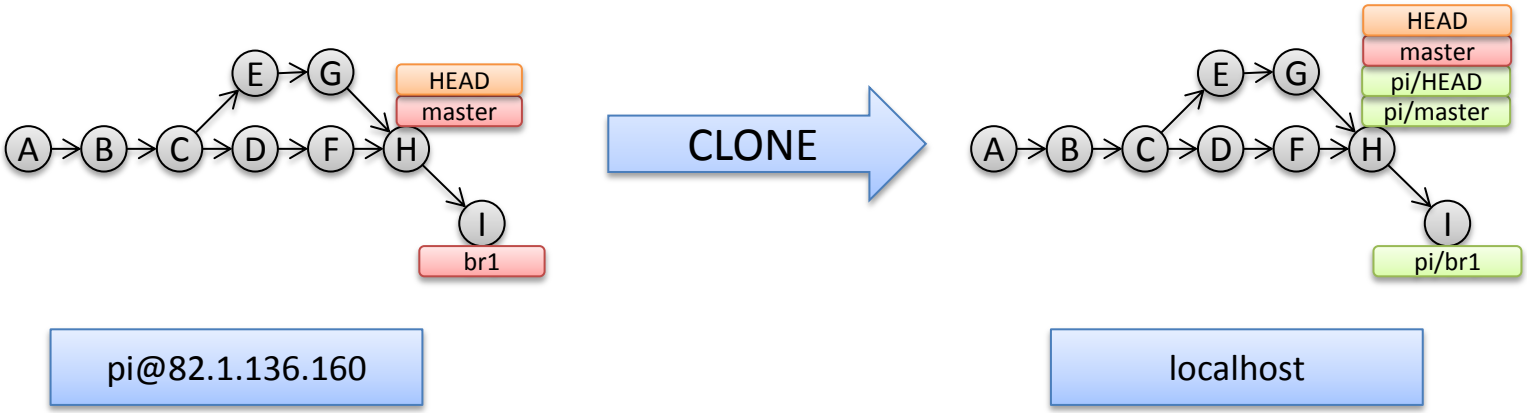
- Before you make a new commit object, you have to add your files to the commit. By default edited files that were already in the last commit will be added automatically, but new files must be added manually. They will then be included in the next snapshot.

- If you merge ref A with ref B and B is simply linearly ahead of A on the same branch, it will simply move ref A to ref B without screwing around with files. This is a good way to move refs ahead that are “lagging” behind.

Git as a team

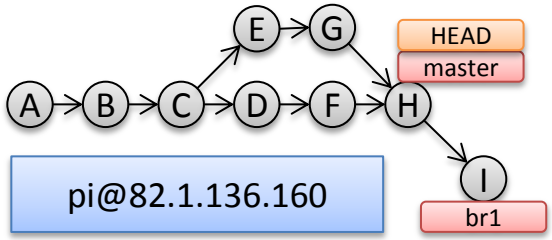
- Everything before is all well and good on your own, but we'll all by working together which is a different beast.
- One *very* important thing to know about git is that when you work, make commits, etc, you are doing all that on your own, local copy. This is a very important distinction and another that makes it very powerful. You can continue doing version control and branching etc even when not connected to the server.

- So the central point-of-access will be the git server running in my house at http://82.1.136.160/D2.git
- The first thing you will do is a *CLONE* operation. This copies the repository exactly onto your computer.

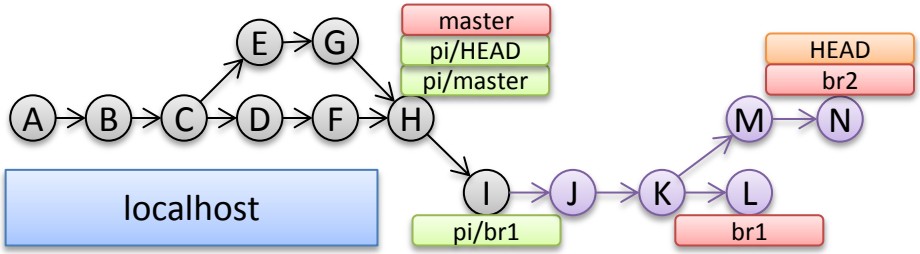


- Notice your cloned copy makes clear where the remote refs are located. They are labelled by <server>/<refname>. In this example the server is called pi.

- At this point you are now free to work on your local copy as you wish! The server will know nothing of it.



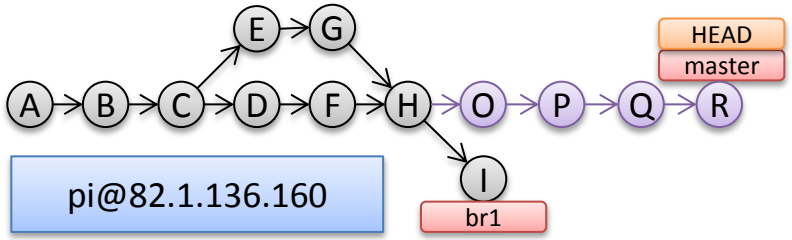
The server, barring anyone else changing it, remains static, blissfully ignorant of your local work



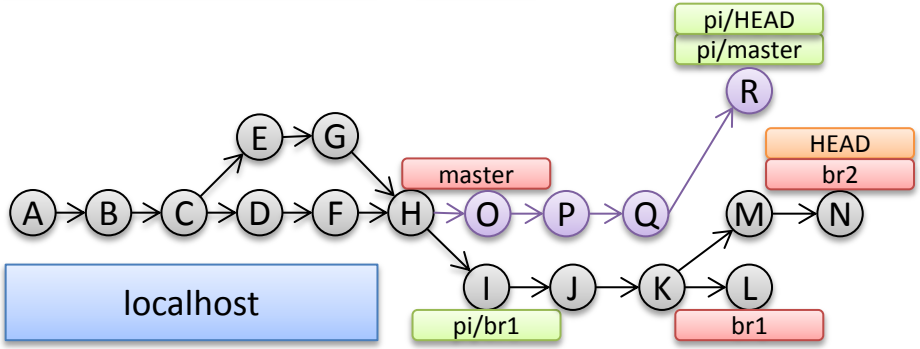
Meanwhile, you may add as much as you like locally!

- The external refs (pi/<whatever>) stay where they are, reflecting the last known state of the server refs. Your local refs move around like usual.

- Let's say that you want to check if the server has been updated since you last checked. You do a *fetch*!



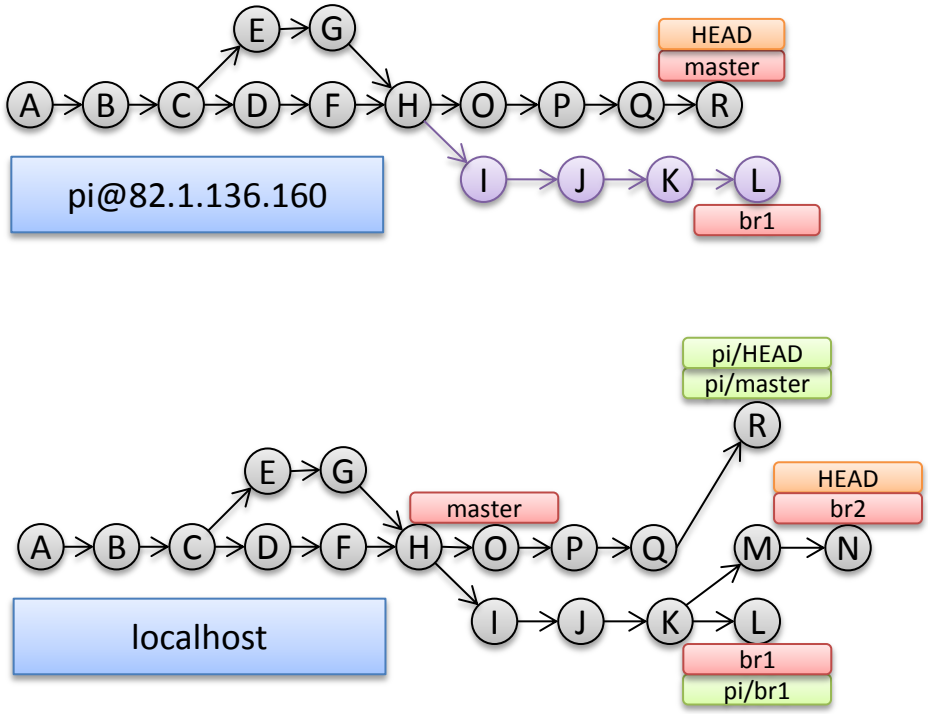
Purple commits are new since we cloned from the server



Novel commits are added to your local tree and the remote refs are updated to reflect the latest state of the server

- Notice that your local refs aren't moved to match the server ones. EG master hasn't been moved to pi/master, br1 hasn't been moved to pi/br1 etc.

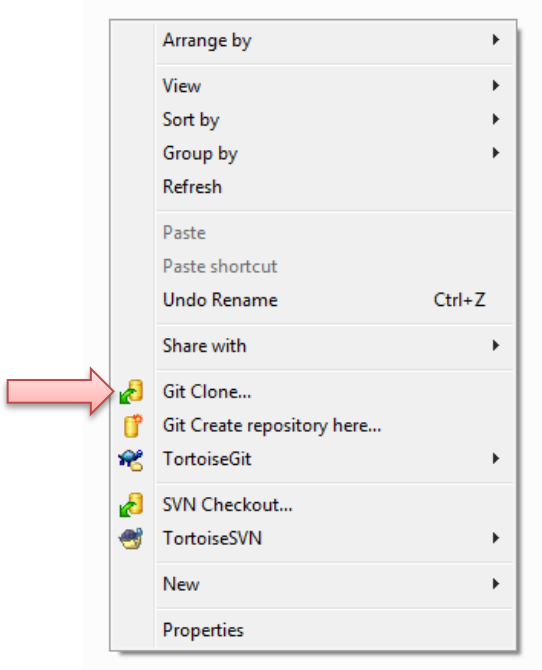
- And what if you want to move your changes back to the server? You *push* a branch. Here we have pushed br1 (to br1 on the server)



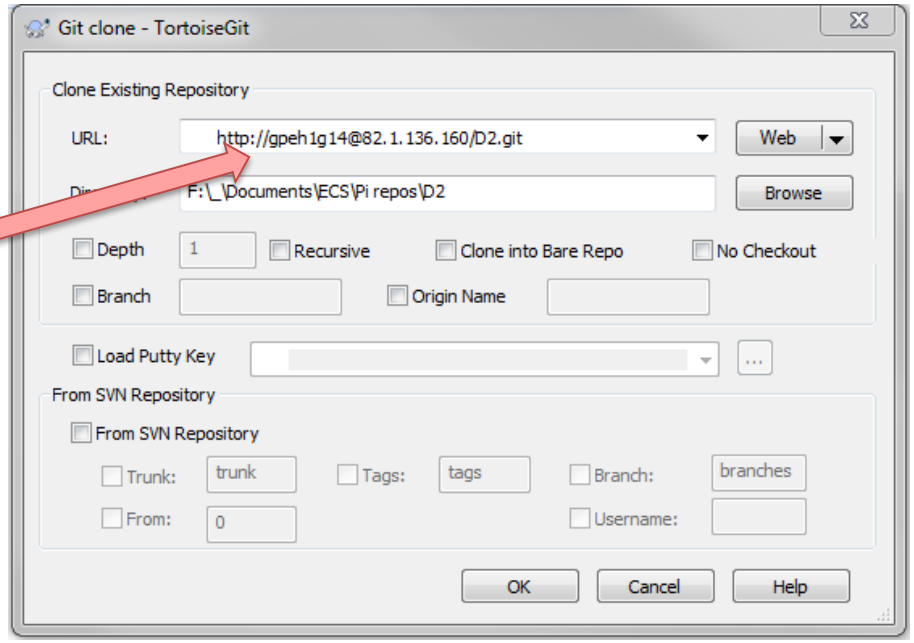
Purple commits have been added to the server. Since we only pushed br1, br2 remains solely on the local machine. Since we pushed to br1, it updated the br1 ref on the server to match with our local copy. We could have pushed to any name we wanted and it would have moved/created the appropriate ref label.

- This is how you can get your changes onto the server

- On Windows, the tool to use is TortoiseGIT (https://tortoisegit.org/). You start by cloning out the repo:



Put your username here



- From there on it should be reasonably self explanatory!

Ze rules

- Generally speaking, do your work in a branch (maybe with your name?). The master on the server should only be merged to on occasion, and you should message me so I can do it on the Pi itself. That way the master is pretty “clean”.
- Comment your commits! Always write something informative so we can track things around.
- If a commit is important, you can tag it. These are more noticeable than commit comments.