Git in a Nutshell

for normal people (version 0.1)

Jonas Jusélius

<jonas.juselius@chem.uit.no>
University of Tromsø
Department of Chemistry
N-9037 University of Tromsø, Norway

Rev.1:
`git clone http://git.jonas.iki.fi/CVS2git.git`
Corrections and improvements are most welcome via e-mail using the `git-format-patch` *facility.*

About this document

The intent of this document is to explain some of the aspects of working with git that I feel are somewhat poorly explained elsewhere. Most git commands have excellent man pages explaining in excruciating detail the particular command. What I feel is missing is a manual which explains what the commands are for, how the pieces fit together and how to actually work with git on a daily basis. This guide has been written with an audience consisting of the typical academic programmer in mind, and relies implicitly on a repository setup as outlined in the Git to CVS Migration Guide.

Finally a disclaimer. I'm not an expert on git, nor do I have a huge experience working with git. All comments, corrections and suggestions are most welcome!

1 Introduction to git

Git is different from CVS in most respects. When you clone a repository you get the WHOLE repository, everything, not just a working copy like in CVS! `git-clone` is almost the same as doing a

```
$ scp -r myserver:/path/to/myrepo .
```

Within this personal repository you can do *whatever* you like, i.e. create branches, delete branches, tags, and of course commit as much as you ever like. It's only when you push to the master that others can see your changes. In fact, your (cloned) repository can act as a master for someone else! The division into master and client is really quite blurred and artificial in git. When you clone a repository, the new copy will contain administrative information in .git/config about where it was cloned from, and have a slightly modified branch structure (as you will see), but apart from this it's identical to it's parent. And where we humans can't choose nor change who our parents are, git has no problem in changing or removing the parent(s).

2 Branches in git

2.1 Remote branches

The thing that differs most from CVS when working with git is the use of branches. Under git branches are used extensively, and they are essentially free. It costs nearly nothing to create branches, both in terms of time and disk space. There is also no need to tag first and branch second, like in CVS.

When you clone a repository (this is called the 'remote' repository, and is referred to as the 'origin') all the branches in that repo are renamed in your local copy by prefixing the branch name with '`origin/`'. These branches are called 'remote branches' or 'tracking branches', to distinguish them from *your*local branches. `git branch -r` show all remote branches. When you run `git-fetch origin` *all* remote branches are updated to the exact state of the remote repository. Now you can inspect the changes on those branches by either using `gitk`, or a combination of `git-log`, `git-whatchanged`, `git-diff` and `git-annotate`, e.g. to inspect changes on the most important remote branch: `gitk origin/master`. Note that `git-fetch` updates the remote tracking branches, not your working branches! Never, ever, checkout a remote branch directly unless you absolutely want trouble.

2.2 Tracking remote branches

In order to incorporate changes in in remote branches you need to merge your working copy with the remote branch, e.g. (`git-checkout mybranch`), `git-merge origin/master`. If you get a conflict, run `git-mergetool` which will fire up the merge tool of your choice. When you have resolved all issues, run `git-commit` to complete the merge. The `git-pull` command basically does a `git-fetch` and `git-merge` in one go. I recommend you *do not* use it unless you very carefully read the man page first, since it has some pitfalls!

As I mentioned earlier, never work on a remote branch directly! This will cause terrible problems when you try to sync with the remote repository. Instead create a local branch *from* the remote branch and do your work on it instead: `git-branch foobar origin/foobar`; `git-checkout foobar`. After a `git-fetch` run `git-merge origin/foobar` to get the latest updates on the branch (and also `git-merge origin/master` if you want to track the "official" changes). When you feel you have something worth showing to others (or just for backup) you can push your changes on some branch to your personal branch on the master server, i.e. the remote repository, `git-push origin mybranch:myuserid`.

2.3 Daily git

All this might sound a bit confusing, so here are my recommendations for how to work with git until you get used to it: When you clone the master repo, you will get a set of remote branches (`git-branch -r`), and one local working branch called 'master'. This branch is in the exact state of the remote master branch (i.e. origin/master and master are identical). I suggest you keep it this way, and create a working branch for yourself, e.g.

```
$ git-branch work
$ git-checkout work
```

Now work like you normally do, and every now and then (every morning for example) do

```
$ git-fetch origin
$ gitk origin/master
# if you like what you see
$ git-merge origin/master
```

Every time you have made changes which can be considered complete in some sense (you know best), do a commit (it's much better with many small commits than a few big commits as you will see):

```
$ git-status
$ git-add file(s)
$ git-commit
```

2.4 The goal

With so many possibilities for organising both repositories and branches it's important not to forget that the ultimate aim should be to get your beautiful new features merged with the master branch on the master server! It's like the musketeers, "All for one, one for all!".

It's important to update often from the master repository, partly not to drift too far away from the master branch, and also to incorporate all bug fixes etc. At the same time it's also important to push to the master often and in small increments, since this makes it a lot easier for other developers to stay in sync with *your* work.

3 Cherry picking

There are of course still lots nice tricks we can play with branches. Suppose you want to try out some idea, but you don't know exactly how it will work out. Create a new branch from your current working branch and check it out:

```
$ git-branch foobar
$ git-checkout foobar
```

Now work hard and commit often. If it turns out that everything is good, and you want to keep all changes, merge them with your working branch and push them to the master:

```
$ git-checkout work
$ git-merge foobar
$ git-push origin work:myuserid
```

Now you can delete the foobar branch for ever and all times

```
$ git-branch -d foobar
```

If, on the other hand it turn's out that you had a crackpot idea, and you don't want to see any of it anymore, delete the branch and all changes, commits and everything on the branch will be gone for ever! No trace of it. But what if there are partially useful changes to foobar that you want to keep, before discarding the rest? Well that's when small commits are useful because you can cherry pick! Chekout your work branch, fire up gitk on foobar, and find the commits you like to keep. Every commmit is identified by a long SHA1 hash (a long sequence of numbers and letters). Now cherry pick the commits you want into the work branch:

```
$ git-checkout work
$ gitk foobar &
# find commit, copy the SHA1 hash with the mouse
$ git-cherry-pick SHA1
```

Repeat the cherry pick as many times you like. As you see, git gives a lot of flexibility by using branches. Just be a bit careful in the beginning not to make a mess and lose track of what you are doing.

3.1 Remote repositories

3.2 Stashing

3.3 Resolving conflicts

3.4 Exporting a repository

3.5 Resetting

3.6 Finding bugs

3.7 Odds and ends

git-mv, git-rm, git-grep, git-whatchanged, git-blame

4 Using git for collaboration

Revision management goes well beyond just source code management for a group of programmers. Revision management is useful for most tasks which are expected to evolve with time, like for example manuscripts. Since git is very easy to set up, and supports a wide range of communication protocols, git can be useful for many collaborative tasks. In the following section we will examine how git can be used to collaborate in a highly disconnected environment, where none of the participants have access to a common server or each others machines. This is a typical situation which arises for shorter term projects, like when collaborating on a scientific manuscript. To facilitate this situation git offers a powerful e-mail facility for communication changes.

Suppose you are working on a LaTeX manuscript and you want to have the whole manuscript under revision control:

```
$ cd ~/tex/manus/
$ git init
$ git add manus.tex fig1.ps fig2.ps
$ git commit
```

That's it! Now you can work happily, and remember commit every now and then so that you always can go back in history if you need to.

At the point when you are ready to send the manuscript to your collaborators, you can either clone your repository in /tmp to obtain a clean copy, or just clean out all unnecessary files. Then make an archive of the whole archive and send it by email to your collaborators[1].

```
$ cd /tmp
$ git clone ~/tex/manus
$ tar vfcz ~/tex/manus.tgz manus
$ rm -rf manus
# mail and attach ~/tex/manus.tex
```

[1]If the file is very big it's probably better to provide a (hidden) link to your home page, as many mail servers will not accept excessively large files

Now you and your collaborators continue to work the manuscript. After some time, and a number of commits, it's time to share your changes with the others. The first step is to identify the commits you want to send. The commits can easily be identified by running `git log`. Suppose you have made 3 commits since you last distributed your changes:

```
$ git format-patch -3
```

This creates 3 patch files in the current directory, which now can be attached and sent to your collaborators using your favourite mailer. Alternatively you can use `git-send-email` to do the job.

```
$ git-send-email --subject '[patch] my latest changes' --to foo@bar.org \
--cc raboof@foobar.edu --cc myself@myhost.net *.patch
$ rm *.patch
```

`git-send-mail` can also be configured using `git-configure` to avoid having to write the long command line every time.

When you receive changes from your collaborators by e-mail, just save the mail(s) in your project directory and apply the changes:

```
$ git-am --3way mailfile(s)
$ rm mailfile(s)
```

It might be a good idea to create and switch to a temporary branch before applying the patches, since this gives you better possibility to inspect the changes before merging them with your main branch. Obviously, if there is a conflict it has to be resolved in the normal manner.