CVS to git Transition Guide

for Cold Turkey Quitters

Jonas Jusélius

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

Rev 0.3:
`git clone http://git.jonas.iki.fi/git_guides.git`
Corrections and improvements are most welcome via e-mail using the `git format-patch` *facility.*

Contents

1 Preliminaries

This guide attempts to document one way of migrating both repositories and users from CVS to git. I'm sure there are many other equally good, or better, ways of doing this. I have called my scheme the Cold Turkey scheme, since it involves freezing CVS in one go, and moving to git without attempting any kind of soft transition period operating both CVS and git in parallel.

This is not a git manual. There is a lot of decent documentation available on the web, and all git commands are very well documented, so there is no need to repeat them here. This document explains step-by-step how to move your local changes in your CVS repositories to the newly created git repository. It assumes that you have a recent version of git installed on your system, and that you have familiarised your self with the basic git operations by reading the available git documentation at http://git.or.cz.

The first part of this guide is intended for repository administrators, and documents step-by-step how to freeze the CVS archive and convert it to a git archive.

The second part is intended for the individual developer, documenting how to set up git for personal use, and how to safely migrate all uncommitted changes from CVS to git.

1.1 Recommended reading

If you have not done so yet, I recommend you start by reading the following two short tutorials:

Git for CVS users :
http://www.kernel.org/pub/software/scm/git/docs/cvs-migration.html

The git tutorial :
http://www.kernel.org/pub/software/scm/git/docs/tutorial.html

Git in a Nutshell :
http://git.jonas.iki.fi/gitweb.cgi?p=git_guides.git;a=tree

Furthermore, the following documents may be of use:

Everyday git with 20 commands or so :
http://www.kernel.org/pub/software/scm/git/docs/everyday.html

Git user's manual :
http://www.kernel.org/pub/software/scm/git/docs/user-maunal.html

1.2 About this document

This guide was written as part of moving the ACES2 (http://www.aces2.de) program package from CVS to git. As such it reflects the practical choices made for this particular environment.

Finally, I'm sure there are errors and/or semi correct statements in this document since I'm not a true git expert (yet). All suggestions and corrections are welcome!

1.3 Conventions

I have used the following conventions in this document: Commands are written in `typewriter` and anything the user should replace with his/her own options have been enclosed in brackets, e.g. `<mybranch>`.

2 Before you start

This section is intended for administrators and users alike, and is based on some of the experience of migrating a bunch of CVS users to git. It seems that one of the main problems is for CVS users to grasp the distributed nature of git, i.e. the difference between a commit and a push. Users can become very confused, and think that git is impossibly complicated to use and understand.

The very nature of git makes it much more powerful than CVS. Git is allows for a lot more flexibility, leading to a different workflow than under CVS. Changing two both the source code revision software *and* the working paradigm at the same time is a recipe for trouble. It's not git itself that is causing the confusion, rather the new way how to use it really efficiently! To avoid having grumpy and confused users it's better to do one thing at a time. Start by changing from CVS to git, but keep the same working paradigm as before. When the users have become comfortable using git, you can implement more advanced things like restricted branches and multiple remotes.

3 Converting your CVS modules to git

The general scheme used here for setting up the new git repository is based in part on having a central master repository, CVS style, and in part on providing centralised git hosting for the developers. The idea is to provide the developers with as much comfort and fexibility as the like.

The central master server is set up with a master branch, which is supposed to contain the latest, greatest, bleeding edge *working* development version of the code. This is the branch from which the developers should pull regularly, and only the repository maintainers have right to push to this branch. There will also be an experimental branch to which people can push code they want others to test. Furthermore there will be a testing branch which acts as a staging area for code to be included in the master branch. It's simple to have the code in the testing branch pulled once per night and run through a test suite. If all tests pass, the repository maintainers can merge with the master branch.

In addition to these standard branches, every user has the right to create and update their own branch on the master server. One "official" branch per developer is probably enough for most users. This simple setup allows users to regularly fetch all remote branches and communicate code sideways if they need to. If one branch is not enough, or if a developer has special requirements, each developer is allowed to have his/hers own repository on the server. This gives maximum flexibility to the users without losing too much transparency. The git web interface shows all repositories in a directory tree, so that everyone can follow the development cycle in all repositories.

3.1 Overview

To implement the Cold Turkey (tm) change to git, the following steps must be taken

1. Decide where the new git repositories should reside
2. Create a UNIX group for the developers, one per project if necessary
3. Make the relevant CVS modules read-only
4. Convert the CVS repositories to git
5. Set permissions on files and directories
6. Set up the 'update' hook to enforce access restrictions
7. Lay back and enjoy a good days work

3.2 Staging for the switch

Decide where you want the new repositories to reside, create the necessary UNIX groups and set the permissions. Now is also a good time to fix permissions on files. Source files should have `chmod 644`, directories and scripts should probably be `chmod 755`.

```
$ mkdir /path/to/git
$ chgrp gits /path/to/git
$ chmod 775 /path/to/git
$ chmod +s /path/to/git
$ chmod +t /path/to/git
```

Then change to your CVSROOT and make the module(s) read-only. Make sure that the lock and history files remain in writable locations. These are configured in $CVSROOT/CVSROOT/config.

```
$ cd $CVSROOT
$ chmod -R ugo-w myproj
```

Now CVS commits are disabled and the repository frozen.

3.3 Converting the repository

Converting a CVS repository with all branches, tags and the whole history is very easy using `git-cvsimport`. Make sure you have a very recent version of git and cvsps installed, to avoid unnecessary problems.

```
$ cd /path/to/git
$ git-cvsimport -k -i -d /path/to/CVSROOT -C myproj myproj
```

To create a bare, shared repository we need to rename the git directory, fix the permissions and edit the git config file

```
$ mv myproj/.git myproj.git
$ rmdir myproj
$ chgrp -R gitusers myproj.git
$ chmod -R ug+w myproj.git
$ chmod +s myproj.git
# the hooks/ and info/ dirs are special since the control repository access
$ cd myproj.git; chgrp -R gitadmin hooks info config description
$ find -type d -exec chmod +s {} \;
```

Then edit the config file:

```
[core]
repositoryformatversion = 0
filemode = true
bare = true
sharedrepository = 1
[receive]
denyNonFastforwards = true
```

Finally edit the description file, adding a one line comment comment describing your project for gitweb.

3.4 Access control

To set up per user or group access restrictions, copy the `update`[1] script to the hooks directory, fix the permissions and make sure it's executable. Then copy the `allowed-users` and `allowed-groups` files to the info directory. Fix the permissions and group ownership of these files and edit to your liking. By default they give every developer the right to have one personal branch and any number of personal tags at the master.

3.5 Administrative details

Git has two ways of storing objects in it's database, either as separate files or in a packed format storing only differences between files. The packed format is for many purposes more efficient than the default storage mode. It's therefore a good idea to set up a cron job to traverse all git repositories on the master server and run `git gc` once per night to pack and clean the repositories. Obviously the cron job must be set up for a user which has sufficient permissions in the repository tree. Edit and copy the git-optimize.sh script distributed with this document to your git repository root directory and make sure it's executable. Then set up the cron job to pack the repositories

```
$ crontab -e

1 1 * * * cd /path/to/git; find -name "*.git" -type d -exec \
/path/to/git/git-optimize.sh {} \;
1 3 1 * * cd /path/to/git; find -name "*.git" -type d -exec \
/path/to/git/git-optimize.sh {} \;
```

[1] Distributed together with the source distribution of this document.

As a last point, it's a good idea to make sure that all users have a `umask` of at least 027 to make sure that the files and directories are accessible by all other developers.

3.6 The web interface

Git comes with a very nice and easy to set web interface. Visit http://repo.or.cz/w/git.git/ to see it action. I will not discuss how to set up your web server, since there is ample information available on the web. To install the web interface copy the gitweb.cgi script to a suitable location (e.g. /usr/lib/cgi-bin on some systems) and edit the config file to point to the right repository tree. The cgi-script will automatically pick up all git projects under this tree. Please note that enabling the gitweb interface also enables people to clone your repositories over http. This might or might not be what you want, so it might be a good idea to set up access restrictions on the web server so that only people you want can connect and browse and clone your code. Your call.

4 Migrating your local CVS repository

4.1 Before you start

Before doing anything else, I suggest that you setup some default configuration variables for git. Git uses a number of config files, system wide, per user and per repository (see the git config man page for more info). As a minimum you should set the following options:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "my@email.com"
```

These options are written in /.gitconfig, and are used by git to ensure that your commit messages are sensible, since user names and mail addresses are not necessarily set properly on all machines. In addition you might want to enable the following options as well:

```
$ git config --global color.branch auto
$ git config --global color.status auto
$ git config --global color.diff false
```

In order to save space you can also enable compression

```
$ git config --global core.compression 1
$ git config --global core.loosecompression 1
```

If you want to use some external (graphical) merge tool to resolve conflicts:

```
$ git config --global merge.tool meld
```

Meld is a fantastic merge tool, and I strongly suggest you have a look at it. Other valid possibilities are kdiff3 and xxdiff (amongst others).

4.2 Updating your CVS repository

Since the old CVS server is no longer accepting new modifications, you need to move all of your local modifications under git and then commit them. To do this you need to follow these directions carefully. Before you start I strongly suggest that you backup your working copy!

Update The first thing you must do is to synchronise your working copy with the CVS server by running

```
$ cvs update
```

This makes sure that your files are in the right state. Files with local modifications will not be overwritten nor updated. If you have modified files that meanwhile have been modified on the master you will probably have a number of conflicts. You need to resolve these conflicts before continuing. Edit the files with conflicts and pick the correct pieces of code between the markers.

Diff The next step is to create a patch file to migrate all your changes from CVS to your new git repository. Now that all files, except your locally modified files, are in the same state as on the master you can run

```
$ cvs diff -u -N >migration.diff
```

This file contains all the differences between the master and your working copy.

4.3 Setting up the git repository

Clone Now you are in a position to clone the new git master repository! Change your working directory to where you want to create your new repository and run

```
$ git clone <myserver>:/path/to/repos/myrepo.git
$ cd aces2
```

Optionally you can also specify the name of the new repository.

Branch and checkout Before you do anything else you need to create a new (local) branch which is in the same state as the old master CVS, and switch to the new branch. The new branch will be called migration (or whatever) and must be created from the tag CVS_migration_HEAD:

```
$ git branch migration CVS_migration_HEAD
$ git checkout migration
```

Patch You are now ready to apply the patch you created and bring your own local modifications into your git repository

```
$ patch -p0 < /path/to/myproj/migration.diff
```

The patch should apply without a hitch if you have done everything correctly. You can now run

```
$ git status
```

to show the status of your repository (i.e. modified files, deleted files, files that needs to be added, etc.).

Congratulations! Your repository is now in the same state as before the switch!

4.4 Committing your changes

The following step in the migration process is to commit all your changes in your local repository. Don't worry, you will not mess up anything on the master, or even make your changes public yet. This is a personal commit so far.

Before you add changes (and files) to commit you should probably think closely about how changes are logically related and how they could be grouped into sets, instead of committing everything as one big blob. It's better to have many, many small commits than a few big ones, since this helps picking out particular patch sets and applying them to other branches (this is known as cherry picking). There is a convenient tool to pick, group and commit your changes

```
$ git citool
```

If you rather do everything by hand, repeatedly run

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

to group changes into sets, until all changes have been committed. This is also how you add untracked files to git.

Finally run

```
$ git status
```

to check that you have not missed anything.

You can also just run

```
$ git commit -a
```

to do what you probably should not, i.e. commit everything as one huge commit.

5 Getting started with git

Ok, so now all your changes have been safely committed in your local repository. The big difference to CVS is that commits are not automatically communicated to the master server. How to communicate your changes to the master will depend on how the master has been set up. The two following sections will explain how to get started with git, and how to communicate with the master repository.

5.1 Using git like CVS

Here I will outline how to use git in a CVS-like fashion, without having to bother with branches and other complications. For completeness I have included how to clone a repository. The corresponding cvs commands are included as an example.

1. To get a local copy of the source repository:

   ```
   [ cvs checkout -d :ext:mylogin@myserver:/path/to/CVSROOT myproj ]
   $ git clone mylogin@myserver:/path/to/repository/myproj.git
   ```

2. Updating the source repository to get the latest changes from the master:

   ```
   [ cvs update ]
   $ git pull origin master
   ```

3. Checking in your changes on the master:

   ```
   [ cvs update ]
   [ cvs checkin file1 file2 ...]

   $ git pull origin
   $ git commit file1 file2 ...
   $ git push origin master
   ```

   This is where git differs from CVS substantially, committing a file commits it to your local repository, and pushing it transfers it to the master. Note that before you can push anything it has to be committed! A more useful way of working is:

   ```
   ... work
   $ git commit file1 file2 ...
   ... work
   $ git commit fileN dir2 ...
   ... work
   $ git commit -a  # (commits everything)
   ... ok, we are ready with a new feature
   $ git pull origin master
   $ git push origin master
   ```

   If you get a conflict when you pull, please consult the Nutshell guide how to resolve it.

4. Common tasks:

   (a) How to add a file:

   ```
   [ cvs add file1 ... ]
   $ git add file1 ...
   ```

   (b) How to delete a file:

```
[ cvs remove file1 ]
$ git rm file1
```

(c) How to move or rename a file:

```
[ CVS cannot do this ]
$ git mv file1 file2
```

(d) How to restore a file which you have deleted:

```
[ cvs update ]
$ git checkout file1
```

5.2 Git vs. CVS

Git is not very different from CVS for everyday use, although it may at first seem so! There are really only two things that differ dramatically from CVS; your working copy is a full repository, hence the separation of making a commit and communicating the commit(s). The other big difference is that git keeps track of the state of a whole repository, not single files.

It's important to understand that a commit under git is exactly the same thing as under CVS, even if it's local to you until you push. When you push git will update the remote branch (master in this case) to be in the exact same state as your repository. This means that if you have 12 commits which are not on the remote server, then all those 12 commits will be communicated! Under CVS you would have made 12 commits directly to the server, under git you can commit any number of times, and when you think it's appropriate you push them all in one batch to the server.

In the old, comfy realm of CVS you would work, work, work and never commit until you had something you really needed or had to commit, like a bugfix in one file in some module. You can do exactly the same with git! Never commit anything until you have something you really need to commit. Then you do the following:

```
# first make sure we are up-to-date
$ git pull origin master
# commit CVS style
$ git commit file1 file2 ... fileN
$ git push origin master
```

This is identical behaviour compared to CVS, except for the fact that you explicitly need to communicate the changes to your 'origin'.

Of course, now I will argue that working like in the old CVS world is not very good. For one thing, it's important to commit often, since this way you get a much better work log/history and it's makes tracking down bugs a lot easier. Furthermore, when you accidentally delete a file, you can always get a reasonably new version back (if you have no daily backups of your work area).

Now the problem arises; you need to publish a small fix, but you have a lot of commits, and the code is not ready for the public yet. This is one place where branches are very convenient:

- If you have not fixed the problem yet, create a new temporary branch (call it fix or whatever), checkout the branch, fix the problem, commit, push to

the master, checkout your working branch, pull the fix from the master, delete the fix branch.

- On the other hand, if you already have fixed and committed the fix, the procedure is almost the same, but with one difference: Create and checkout the fix branch, use 'gitk' or 'git log' to find the SHA1-number of the commit which fixed the problem, then use 'git cherry-pick SHA1-number' to incorporate that commit and nothing else into the fix branch, push to the master, checkout the working branch and delete the fix branch.

5.3 Working with remote servers

In the setup outlined above, every developer has the possibility to cerate their own branch on the master (note that it's not created by default). You can only commit branches and tags which have names that start with your user id.

5.3.1 Creating a remote branch

To create a new branch on a remote server, git requires a very explicit syntax. After you have created the branch, you can use the same syntax as when operating on local branches. To create a remote branch and push your changes in the migration branch to it, do

```
$ git push origin migration:refs/heads/<muid>
```

where `<myuid>` is your login on the server.

5.3.2 Fetching changes

To follow what others are doing and incorporating their latest changes you need to regularly fetch their changes. git is very flexible in the ways you can do things, so consult the man pages for more info. To download all changes from the master run

```
$ git fetch origin
```

This downloads all remote branches and stores them in your remote tracking branches (use `git branch -r` to see them). `git fetch` does not merge any changes into your working branches. After the git fetch you can examine the changes to the remote branches using e.g. `gitk --all` or `git [log/diff] origin/master`. To update your current branch with e.g. the remote master branch do

```
$ git merge origin/master
```

5.3.3 Pulling changes

There is a much easier way of updating your branches from the master server:

```
$ git pull origin master
```

This command automatically fetches the master branch from the server and merges it with your current branch. This is more or less the git equivalent of `cvs update`. Note that this only fetches the specified branch from the server. If you want all changes on all branches use the fetch command above.

5.3.4 Push

To publish changes to a branch so that others can see them and alternatively merge them with their branches you must push them to the main server. This is in a sense the git equivalent of a `cvs commit`. To upload a branch and merge it with your personal branch on the server, do

```
$ git push origin somebranch:<myuid>
```

For more details see the git push man page. Note that for a push to work, it must result in a so called fast-forward-merge. Fast-forward means that the files you want to merge on another branch, must have the files in the other branch as parents. Simply stated, the files on the other branch have not changed since the last pull. If the push fails because it's not fast-forward, you must first do a `git pull origin branch` and resolve any conflicts before (re)doing the push.

5.3.5 Tagging

To create a tag, simply run `git tag <tagname>`. To push your tags to the main server, do

```
$ git push --tags origin
```

This will push all your tags to the main server. Note that you are not allowed by the server to create tags unless they are prefixed with your user id, e.g. `<myuid>/mytag`. If you don't want to push all tags, you can use the special syntax

```
$ git push origin tag <myuid>/<mytag>
```

To delete a tag on the master you need to explicitly push an empty tag

```
$ git push origin :refs/tags/<myuid>/<mytag>
```

5.3.6 Deleting a remote branch

If you for some reason want to delete a branch on the remote server, just push an empty branch to the remote branch:

```
$ git push origin :<branch>
```

Since branches come and go, both in `origin` and in your own remote(s) it's a good idea to clean up the remotes once in a while by doing a

```
$ git remote prune origin
```

5.4 Adding a remote repository

If you want to have more flexibility, more branches and tags (on the server) and more freedom generally, you can also set up your own personal repository on the master. git makes it quite easy to work with multiple remote repositories in one go.

To set up your own sub-master repository follow these steps:

```
$ ssh <myserver>
$ cd /path/to/repos
$ mkdir $USER
$ git clone -s -l --bare proj.git $USER/proj.git
$ cd $USER/proj.git
$ git branch -a
# remove any branches and tags you do not care about
$ git branch -D <branch branch...>
$ git tag -d <tag tag...>
# edit description (for gitweb) to your liking
$ vi description
```

The -s and -l options to `git clone` will cause git to set up the repository to use the master's object database as much as possible, so that it will take up very little space.

Now on your local machine, cd to your development repository and register the new remote:

```
$ git remote add <name> <myserver>:/path/to/repos/<myuid>/proj.git
```

Now when you fetch, pull and push use your new remote-tag `<name>` instead of `origin` to the corresponding command. The new development cycle will typically be something like

```
$ git pull origin master
# do some work
$ git push myremote mybranch:mybranch
# or to push all branches automatically
$ git push --all myremote
```

6 Useful git commands

Here is a short list of useful git commands to familiarise yourself with when you feel that you have the basics under control. `man git-<command>` will give you all available information on a particular command.

`gitk` Graphical interface for browsing repositories.

`qgit` Graphical interface for browsing repositories. Better and more advance than gitk, but probably not installed on your system by default.

`git diff` View differences between your working copy and a committed revision (and more).

`git gc` By default git will store all objects in a very primitive and storage inefficient way. This command packs, compresses and prunes unreachable objects in the object database. **Should be used regularly**.

`git stash` If you need to checkout a new branch, but don't want to commit your changes, stash allows you to temporarily save your changes. Only in git 1.5.3 and newer.

`git cherry-pick` Instead of merging two complete branches, only pick the commits you really want from a branch.

`git bisect` Find a commit which introduced some nasty bug. Very helpful.

`git log` View the commit history on a branch

`git archive` Create a tar or zip archive from a branch.

`git mv` Rename files in the repository.

`git rm` Delete files from the repository.

`git annotate` Show who changed what and when in files.

`git format-patch` Generate an patch file of a commit, suitable for sending by e-mail

`git am` Apply a patch from a mailbox