Accessing Olsonet git Repositories

June 28, 2010

Introduction

We have given up the CVS repository at cs.ualberta.ca. The primary reason was a security breach affecting some departmental systems and causing a prolonged disconnection of our server from the world, which left us with no easy means to synchronize our changes at a rather critical moment (as it usually happens). Using this incident as a pretext, we decided to switch to *git*. The conversion of the original CVS repository to git was pretty much automatic (git-cvsimport). The single original repository (holding three packages: PICOS, SIDE, and VUEE) has been split into three separate git repositories. The original structure of commits, branches, and tags has been preserved and, as far as I can tell, all the different snapshots/branches that were accessible under CVS are still easily retrievable via git.

This document explains how to access the present (probably temporary) repositories hosted at http://unfuddle.com. Should the location of those repositories change, most of the information in this note is going to remain relevant, except for the URLs of the repositories (and the way to log on to the server). The kind of access offered by unfuddle (via ssh) is typical for git repositories.

Please become acquainted with the basics of git before reading the rest of this document. While I give you some examples illustrating typical commands and scenarios, they shouldn't be used as a replacement for elementary understanding of what you are doing. Note that my familiarity with git is rather superficial at the moment, but I have read a bit and gone through a few exercises. Here are two useful links:

http://progit.org/book/
http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

The first one is a friendly, reasonably short, and quickly absorbed online book on git; the second is the more or less official manual (not quite complete, but very useful if you have to find out about details of some of the more tricky features). And, of course, you can always resort to the (mostly ugly) man pages.

Local prerequisites

Make sure that you have a decent version of git installed on your system. I have done my work so far under Cygwin (git comes standard with the full installation). My version of git, which you can see by executing:

```
git --version
```

is 1.7.0.4.

Execute these commands:

```
git config --global user.name "Your Name"
git config --global user.email "your@email.address"
git config --global core.editor vim
```

They will create a file named .gitconfig in your home directory storing the parameters that you have entered (in some form which you can easily inspect and later edit by hand). Your specified identity will be used to automatically annotate your changes.[1]

[1] Note that all the modifications introduced until now (which have been automatically carried over from CVS) have a dummy author.

The central repository

You have to log on to the account that I have created at unfuddle.com and provide your public ssh key, which will give your git smooth access to the repositories. Point your browser at:

https://pawel.unfuddle.com/

The user name is pkgn (I will send you the password separately). Please move down to *Public Keys* and click on *New Public Key*. Enter a name for the key (e.g., Wlodek's laptop) and paste the key into the *Value* field (do I have to explain this?). You will see that there is already one key there, named MNI, which belongs to me.

Having entered the new key, and saved the changes, click on the *Repositories* tab. You will see the three repositories along with their URLs. Those URLs will allow you to connect to them from your machine. Note that unfuddle offers some features (like web browsing through the repositories), which I haven't had time to explore yet. Please be my guest.

Back to the local system

Fetch the stuff to your local machine. Move to the directory where you want to keep it and execute:

```
git clone git@pawel.unfuddle.com:pawel/side.git SIDE
git clone git@pawel.unfuddle.com:pawel/picos.git PICOS
git clone git@pawel.unfuddle.com:pawel/vuee.git VUEE
```

The last arguments can be full directory paths, so you can execute these commands from anywhere. They will fetch the repositories and checkout the *master* contents of each of them. This will basically get you back to where we were before the mishap, except that this time it will be much easier and faster to view previous versions, experiment with temporary changes, and generally keep track of things. Or so I hope.

For a quick illustration, move to PICOS and do this:

```
git checkout GENESIS
```

After two or three seconds you are inside the GENESIS release. Then do:

```
git checkout master
```

and you are back in the most recent commit of the "main trunk" (using the now obsolete terminology of CVS).

Let us go through a simple modification cycle. Suppose that you are in master, and you have modified some files. When you execute:

```
git status
```

git will show you the files that have been modified and not yet *staged* for addition (this terminology is explained in chapter 2 of the book I mention above) as well as any files (like binaries, for example) that have been created anywhere in the tree, but are not being *tracked*, i.e., they do not belong to the repository.

You can *add* each of those modified files (the same applies to any new files to be included in the repository) with:

```
git add filename
```

or

```
git add -A
```

if all of them are to be tracked (status doesn't report any files that should not be in the repository). Then you can do:

```
git commit
git tag tagname          (e.g., R100629A)
```

Note that up to this point all the changes, including the commit, apply to your copy of the repository. To send them to the central repository, do this:

```
git push --all
git push --tags
```

The second operation is not needed if you haven't introduced any tags that should be copied to the central repository. This pretty much covers the way we used to work with CVS. Here is the transcript of a somewhat more sophisticated way to introduce and test changes:

```
git checkout master
git branch  mytest
git checkout mytest
```

The last two commands can be compressed into a single:

```
git checkout -b mytest
```

The sequence creates a new branch which starts being identical to master. You can now edit/create files, run tests, and so on, without affecting anything at master (to which you can always easily revert). You can also commit the changes (they will be committed to mytest); you can even push them to the central repository (e.g., to show to other people). When you are ready, you do:

```
git checkout master
git merge mytest
```

and the changes will be merged (*fast forwarded*) into master. Let us go through a complete exercise using specific files, branches, and tags. This will pretty much cover all I have ever been doing with CVS (in fact quite a bit more ;-). So we start:

```
cd PICOS
git checkout master
git checkout -b mytest
```

Our objective is to introduce some changes, pretend to test them (keeping master intact for as long as possible), then merge the new changes into master and commit them (tagging the commit in our traditional way).

While operating within mytest, we move to Apps/VUEE/survey and edit two files: app_peg.cc and app_diag_peg.cc (changing the copyright year to 2010). We compile the praxis for a real node:

```
mkmk WARSAW
make
```

Then we move back to PICOS and add a note at the end of RTAGS. (tagged with R100627A). Now, when we say:

```
git status
```

we get this:

```
# On branch mytest
# Changed but not updated:
#   (use "git add <file>..." to update what will be
#    committed)
#   (use "git checkout -- <file>..." to discard changes in
#    working directory)
#
#       modified:   Apps/VUEE/survey/app_diag_peg.cc
#       modified:   Apps/VUEE/survey/app_peg.cc
#       modified:   RTAGS
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
#    committed)
#
#       Apps/VUEE/survey/.gdbinit
#       Apps/VUEE/survey/Image_peg
#       Apps/VUEE/survey/Image_peg.a43
#       Apps/VUEE/survey/KTMP/
#       Apps/VUEE/survey/Makefile
#       Apps/VUEE/survey/gdb.ini
no changes added to commit (use "git add" and/or
      "git commit -a")
```

Note that the untracked files result from the compilation (they do not belong to the repository). We can ignore them and add the relevant files by hand, e.g.,

```
git add Apps/VUEE/survey/app_diag_peg.cc \
      Apps/VUEE/survey/app_peg.cc RTAGS
```

(this is supposed to be a single line), or remove the untracked files:

```
cd Apps
./cleanup
```

do git status again (for certainty), and say:

```
git add -A
```

to add all the modified files.

A digression: you do not absolutely have to execute git at the root level of the package (where the special .git directory is kept). Apparently, git searches for .git upwards from where you call it. In such a case, the paths of all listed files (like in the above output) will be relative to where you are (so they always can be sensibly pasted as arguments of commands).

The next step is to commit the changes, i.e.,

```
git commit
```

This will open the text editor (the one you configured in as your favorite) asking you to enter a commit message. I recommend to enter the tag you are going to assign to the commit (if any) and a single line (e.g., hinting at the most significant change in the commit). Note that you can in principle enter here a complete description of the changes, in the manner we used to write them into RTAGS, but, IMO, it makes sense to retain RTAGS (for more elaborate notes), using commit messages for brief hints. Those messages can be listed quickly and thus can be useful in locating specific sets of changes at a glance.

So let us enter this:

```
R100627A: Dummy changes testing the new git setup.
```

and exit the text editor. This completes the commit. Now we can assign the tag to it:

```
git tag R100627A
```

The changes (including the commit) have gone into branch mytest. This branch can still be worked on (more commits) without affecting master (we can switch back and forth between master and mytest). When we are ready, we do:

```
git checkout master
git merge mytest
```

which merges mytest into master. In this case the operation "fast forwards" master to our new commit, because mytest descends directly from master (which doesn't have to hold generally). Now, we can remove the temporary branch (which at this time points to exactly the same place as master):

```
git branch -d mytest
```

and we are ready to synchronize the central repository (push the changes into it):

```
git push --all
```

This should be followed by:

```
git push --tags
```

because we have added a new tag to the local repository (and we want it to show up in the remote one). For some reason, tags have to be pushed separately. This is probably because they are not considered as useful as we have made them for us.

Whenever you want to synchronize your personal repository with the central one, you do:

```
git pull
```

The way this operation is performed is quite educational. Your repository maintains a number of pointers (heads) to various branches of the tree. One of them, master, intentionally points to the end of the "main trunk". The above command accepts optional arguments, called *refspecs,* of the form <src>:<dst>, where <src> describes a source head (in the remote repository), and <dst> is interpreted locally. By default (no explicit refspecs), the set of effective resfspecs is described by a pattern in your .git/config file, i.e., in this section:

```
[remote "origin"]
      fetch = +refs/heads/*:refs/remotes/origin/*
      url = git@pawel.unfuddle.com:pawel/picos.git
```

which describes the remote repository associated (by default) with your local repository. The *fetch* pattern says this: for all heads present in the remote repository, copy them to the corresponding remotes/origin/... heads in the local repository. This is what would be accomplished by a default (argument-less) git fetch command.

Note that you can reference from your local repository heads like remotes/origin/master, for example, you can do:

```
git checkout remotes/origin/master
```

which will get you to master as it looked in the remote repository when you last pulled (or fetched) from it. View this as a backup version of master (and this also applies to other branches that you fetched from the remote repository) which you are not supposed to do any development on. These are the branches to fetched into from the remote repository. Needless to say, you can branch them off and do with that stuff whatever you please.

The extra job performed by pull, in addition to fetching the "remotes", is to merge them into your refs/heads, thus, bringing your development heads up to date with the remote repository.