Chapter 23

Source code control

Source code control systems, also called *revision control* or *version control* systems, are a way of storing software, where not only the current version is stored, but also all previous versions. This is done by maintaining a *repository* for all versions, while one or more users work on a 'checked out' copy of the latest version. Those of the users that are developers can then commit their changes to the repository. Other users then update their local copy. The repository typically resides on a remote machine that is reliably backup up.

There are various reasons for keeping your source in a repository.

- If you work in a team, it is the best way to synchronize your work with your colleagues. It is also a way to document what changes were made, by whom, and why.
- It will allow you to roll back a defective code to a version that worked.
- It allows you to have branches, for instance for customizations that need to be kept out of the main development line. If you are working in a team, a branch is a way to develop a major feature, stay up to date with changes your colleagues make, and only add your feature to the main development when it is sufficiently tested.
- If you work alone, it is a way to synchronize between more than one machine. (You could even imagine traveling without all your files, and installing them from the repository onto a borrowed machine as the need arises.)
- Having a source code repository is one way to backup your work.

There are various source code control systems; in this tutorial you can learn the basics of *Subversion* (also called *svn*), which is probably the most popular of the traditional source code control systems, and Mercurial (or `hg`), which is an example of the new generation of *distributed source code control* systems.

23.1 Workflow in source code control systems

Source code control systems are built around the notion of *repository*: a central store of the files of a project, together with their whole history. Thus, a repository allows you to share files with multiple people, but also to roll back changes, apply patches to old version, et cetera.

The basic actions on a repository are:

- Creating the repository; this requires you to have space and write permissions on some server. Maybe your sysadmin has to do it for you.
- Checking out the repository, that is, making a local copy of its contents in your own space.
- Adding your changes to the repository, and
- Updating your local copy with someone else's changes.

Adding your own changes is not always possible: there are many projects where the developer allows you to check out the repository, but not to incorporate changes. Such a repository is said to be read-only.

Figure 23.1 illustrates these actions for the Subversion system. Users who have checked out the repository

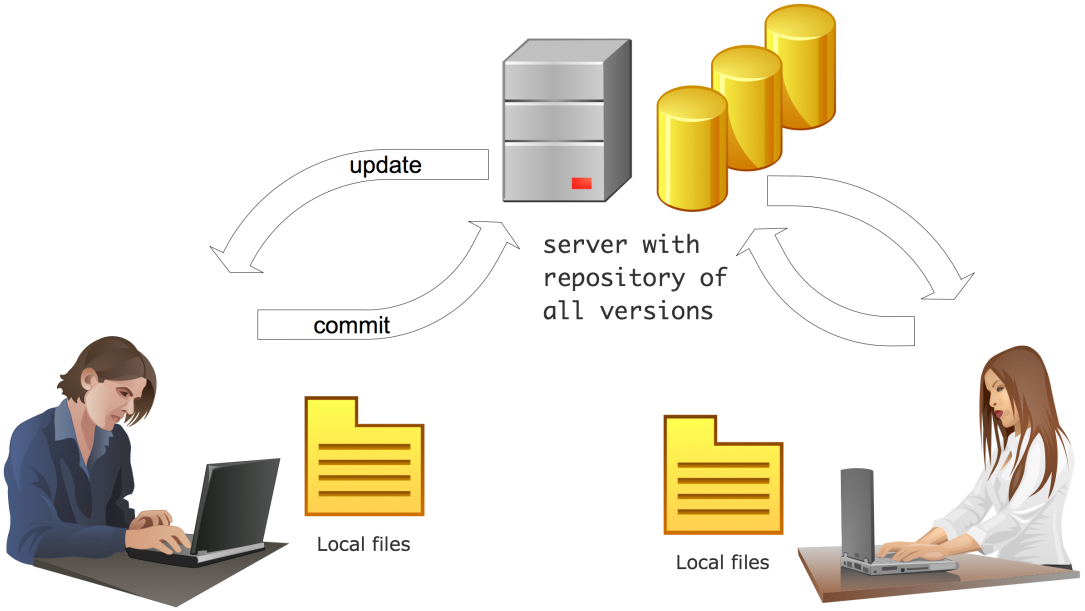

Figure 23.1: Workflow in traditional source code control systems such as Subversion

can edit files, and check in the new versions with the `commit` command; to get the changes committed by other users you use `update`.

One of the uses of committing is that you can roll your code back to an earlier version if you realize you made a mistake or introduced a bug. It also allows you to easily see the difference between different code version. However, committing many small changes may be confusing to other developers, for instance if they come to rely on something you introduce which you later remove again. For this reason, *distributed source code control* systems use two levels of repositories.

There is still a top level that is authoritative, but now there is a lower level, typically of local copies, where you can commit your changes and accumulate them until you finally add them to the central repository. This also makes it easier to contribute to a read-only repository: you make your local changes, and when you are finished you tell the developer to inspect your changes and pull them into the top level repository. This structure is illustrated in figure 23.2.

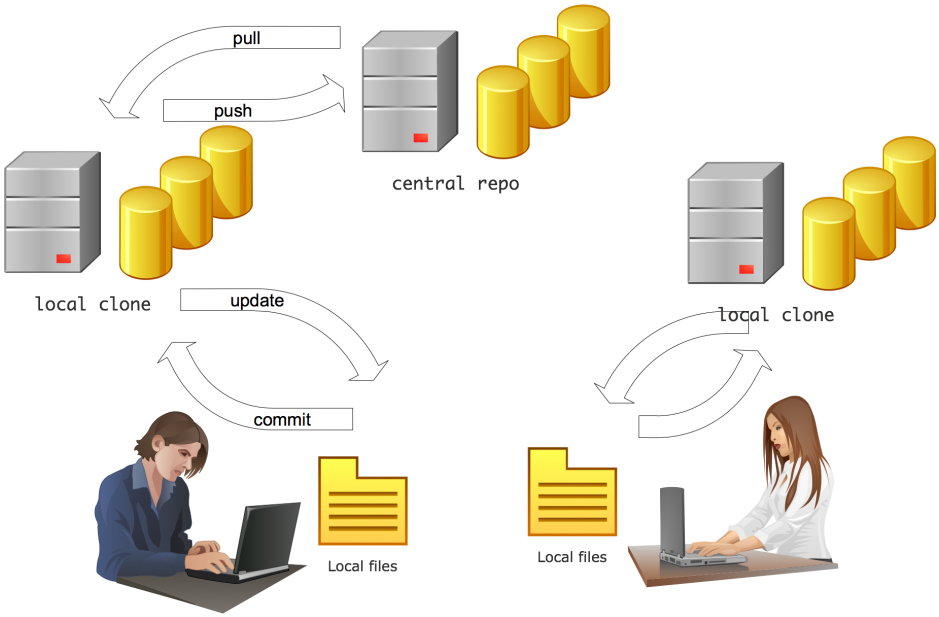

Figure 23.2: Workflow in distributed source code control systems such as Mercurial

23.2 Mercurial (hg) and Git

Mercurial and *git* are the best known of a new generation of *distributed source code control* systems. Many commands are the same as for subversion, but there are some new ones, corresponding to the new level of sophistication. Mercurial and git share some commands, but there are also differences. Git is ultimately more powerful, but mercurial is easier to use at first.

Here is a translation between the two systems: https://github.com/sympy/sympy/wiki/Git-hg-rosetta-s

Minimal introduction to git: http://rogerdudler.github.io/git-guide/

This lab should be done two people, to simulate a group of programmers working on a joint project. You can also do this on your own by using two clones of the repository, preferably opening two windows on your computer.

Best practices for distributed version control: https://homes.cs.washington.edu/~mernst/advice/version-control.html

23.2.1 Create and populate a repository

Purpose. In this section you will create a repository and make a local copy to work on.

mercurial	git
`hg clone <url> [ localdir ]`	`git clone <url> [ localdir ]`

First we need to have a repository. In practice, you will often use one that has been previousy set up, but there are several ways to set up a repository yourself. There are commercial and free hosting services such as http://bitbucket.org. (Academic users can have more private repositories.)

Let's assume that one student has created a repository `your-project` on Bitbucket. Both students can then clone it:

```
%% hg clone https://YourName@bitbucket.org/YourName/your-project
updating to branch default
0 files updated, 0 files merged, 0 files removed,
    0 files unresolved
```

or

```
%% git clone git@bitbucket.org:YourName/yourproject.git
Cloning into 'yourproject'...
warning: You appear to have cloned an empty repository.
```

You now have an empty directory `your-project`.

Exercise. Go into the project directory and see if it is really empty.

Expected outcome. There is a hidden directory `.hg` or `.git`

23.2.2 New files

Creating an untracked file

Purpose. In this section you will make some simple changes: creating a new file and editing an existing file

mercurial	git
`hg status [ path]`	`git status [ path ]`
`hg add [ files ]`	`git add [ files ]`
once for each file	every time the file is changed

One student now makes a file to add to the repository:

```
%%  cat > firstfile
a
b
c
d
e
f
^D
```

(where `^D` stands for control-D, which terminates the input.) This file is unknown to `hg`:

```
%% hg status
? firstfile
```

Git is a little more verbose:

```
git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

firstfile

nothing added to commit but untracked files present
  (use "git add" to track)
```

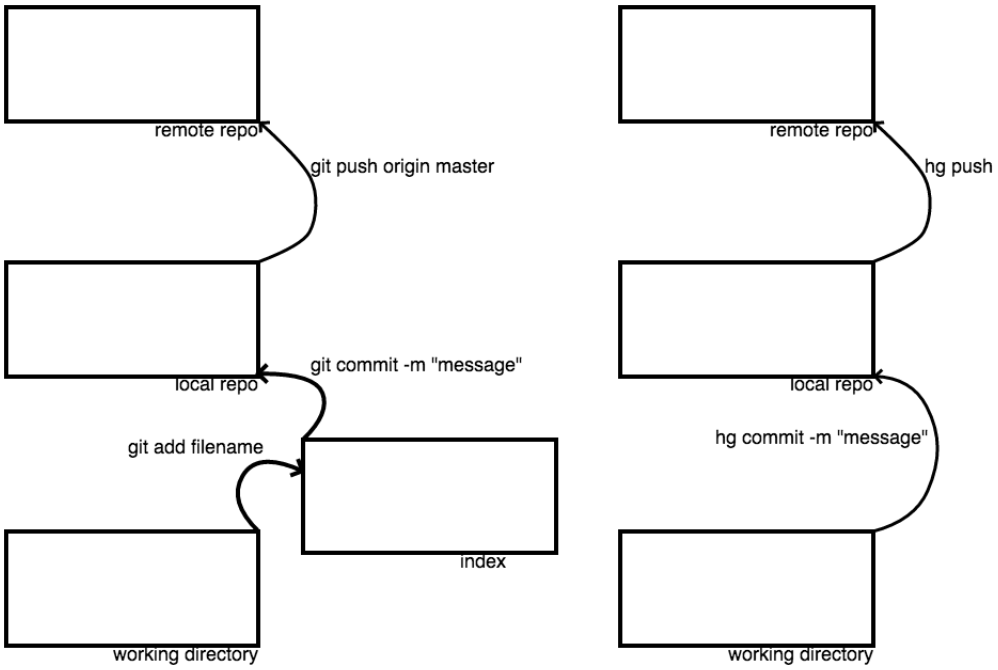

Figure 23.3: Add local changes to the remote repository

Adding the file to the repository We need to declare the file as belonging to the repository; a subsequent `hg commit` command then copies it into the repository.

```
%% hg add firstfile
```

```
%% hg status
A firstfile
%% hg commit -m "made a first file"
```

or

```
%% git add firstfile
%% git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   firstfile
%% git commit -a -m "adding a first file"
  [master (root-commit) f4b738c] adding a first file
  1 file changed, 5 insertions(+)
  create mode 100644 firstfile
```

mercurial	git
`hg commit -m <message>`	`git commit -m <message>`
`hg push`	`git push origin master`

Unlike with Subversion, the file has now only been copied into the local repository, so that you can, for instance, roll back your changes. If you want this file added to the master repository, you need the `hg push` command:

```
%% hg push https://YourName@bitbucket.org/YourName/your-project
pushing to https://YourName@bitbucket.org/YourName/your-project
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
remote: bb/acl: YourName is allowed. accepted payload.
```

In the `push` step you were probably asked for your password. You can prevent that by having some lines in your `$HOME/.hgrc` file:

```
[paths]
projectrepo = https://YourName:yourpassword@bitbucket.org/YourName/my-project
[ui]
username=Your Name <you@somewhere.youruniversity.edu>
```

Now the command `hg push projectrepo` will push the local changes to the global repository without asking for your password. Of course, now you have a file with a cleartext password, so you should set the permissions of this file correctly.

With git you need to be more explicit, since the ties between your local copy and the 'upstream' repository can be more fluid.

```
git remote add origin git@bitbucket.org:YourName/yourrepo.git
git push origin master
```

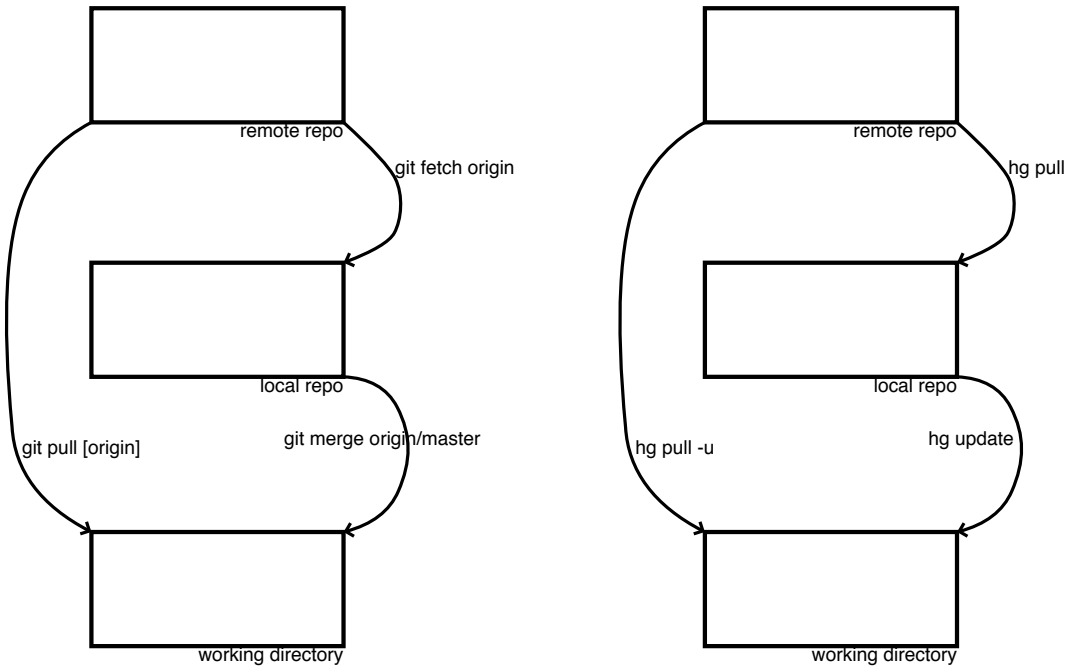

Figure 23.4: Get changes that were made to the remote repository

The second student now needs to update their repository. Just like the upload took two commands, this pass also takes two. First you do `hg pull` to update your local repository. This does not update the local files you have: for that you need to do `hg update`.

Exercise. Do this and check that the contents of the file are correct.

Expected outcome. In order to do the update command, you have to be in a checked-out copy of the repository.

Caveats.

Exercise. Let both students create a new directory with a few files. Declare the directory and commit it. Pull and update to obtain the changes the other mde.

Expected outcome. You can do `hg add` on the directory, this will also add the files contained in it.

Since you will mostly be doing an update immediately after a pull, you can combine them:

```
hg pull -u
```

Git will report what files are updated; for Hg you need to take the changeset number and query:

```
hg status --change 82ffb99c79fd
```

Remark 11 *In order for Mercurial to keep track of your files, you should never do the shell commands* `cp` *or* `mv` *on files that are in the repository. Instead, do* `hg cp` *or* `hg mv`*. Likewise, there is a command* `hg rm`.

23.2.3 Oops! Undo!

One of the reasons for having source code control is to be able to revert changes. The easiest undo is to go back to the last stored version in the repository.

mercurial	git
`hg revert <yourfile>`	`git checkout -- <yourfile>`

23.2.4 Conflicts

Purpose. In this section you will learn about how do deal with conflicting edits by two users of the same repository.

Now let's see what happens when two people edit the same file. Let both students make an edit to `firstfile`, but one to the top, the other to the bottom. After one student commits the edit, the other can commit changes, after all, these only affect the local repository. However, trying to push that change gives an error:

```
%% emacs firstfile # make some change
%% hg commit -m ''first again''
%% hg push test
abort: push creates new remote head b0d31ea209b3!
(you should pull and merge or use push -f to force)
```

The solution is to get the other edit, and commit again. This takes a couple of commands:

```
%% hg pull myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)

%% hg merge
merging firstfile
```

```
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

%% hg status
M firstfile
%% hg commit -m ``my edit again''
%% hg push test
pushing to https://VictorEijkhout:***@bitbucket.org/
                      VictorEijkhout/my-project
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 2 changesets with 2 changes to 1 files
remote: bb/acl: VictorEijkhout is allowed. accepted payload.
```

This may seem complicated, but you see that Mercurial prompts you for what commands to execute, and the workflow is clear, if you refer to figure 23.2.

Exercise. Do a `cat` on the file that both of you have been editing. You should find that both edits are incorporated. That is the 'merge' that Mercurial referred to.

If both students make edits on the same part of the file, version control can no longer resolve the conflicts. For instance, let one student insert a line between the first and the second, and let the second student edit the second line. Whoever tries to push second, will get messages like this:

```
%% hg pull test
added 3 changesets with 3 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
%% hg merge
merging firstfile
warning: conflicts during merge.
merging firstfile incomplete!
    (edit conflicts, then use 'hg resolve --mark')
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges
    or 'hg update -C .' to abandon
```

There are now the following options:

1. There is usually a way to indicate whether to use the local or the remote version.
2. There are graphical programs to resolve conflicts. They will typically show you 3 colums, for the two versions, and for your resolution. You can then indicate 'take this from the local version, and this from the remote'.
3. You can also edit the file to resolve the conflicts yourself. We will discuss that shortly.

Both will give you several options. It is easiest to resolve the conflict with a text editor. If you open the file that has the conflict you'll see something like:

```
<<<<<<< local
aa
bbbb
=======
aaa
a2
b
>>>>>>> other
c
```

indicating the difference between the local version ('mine') and the other, that is the version that you pulled and tried to merge. You need to edit the file to resolve the conflict.

After this, you tell hg that the conflict was resolved:

```
 hg resolve --mark
%% hg status
M firstfile
? firstfile.orig
```

or

```
git add <name of that file>
```

After this you can commit and push again. The other student then needs to do another update to get the correction.

Not all files can be merged: for binary files Mercurial will ask you:

```
%% hg merge
merging proposal.tex
merging summary.tex
merking references.tex
 no tool found to merge proposal.pdf
keep (l)ocal or take (o)ther? o
```

This means that the only choices are to keep your local version (type `l` and hit return) or take the other version (type `o` and hit return). In the case of a binary file that was obvious generated automatically, some people argue that they should not be in the repository to begin with.

23.2.5 Inspecting the history

Purpose. In this section, you will learn how to view and compare files in the repository.

If you want to know where you cloned a repository from, look in the file `.hg/hgrc`.

The main sources of information about the repository are `hg log` and `hg id`. The latter gives you global information, depending on what option you use. For instance, `hg id -n` gives the local revision number.

`hg log` gives you a list of all changesets so far, with the comments you entered.
`hg log -v` tells you what files were affected in each changeset.
`hg log -r 5` or `hg log -r 6:8` gives information on one or more changesets.

To see differences in various revisions of individual files, use `hg diff`. First make sure you are up to date. Now do `hg diff firstfile`. No output, right? Now make an edit in `firstfile` and do `hg diff firstfile` again. This gives you the difference between the last commited version and the working copy.

mercurial	git
`hg diff <file>`	`git diff HEAD <file>`
`hg diff -r A -r B <file>`	`git diff A^..B <file>`

Check for yourself what happens when you do a commit but no push, and you issue the above diff command.

You can also ask for differences between committed versions with `hg diff -r 4:6 firstfile`.

The output of this diff command is a bit cryptic, but you can understand it without too much trouble. There are also fancy GUI implementations of hg for every platform that show you differences in a much nicer way.

If you simply want to see what a file used to look like, do `hg cat -r 2 firstfile`. To get a copy of a certain revision of the repository, do `hg export -r 3 . ../rev3`, which exports the repository at the current directory ('dot') to the directory `../rev3`.

If you save the output of `hg diff`, it is possible to apply it with the Unix `patch` command. This is a quick way to send patches to someone without them needing to check out the repository.

23.2.6 Transport

Mercurial and git can use either `ssh` or `http` as *transport*. With Git you may need to redefine the transport for a `push`:

```
git remote rm origin
git remote add origin git@github.com:TACC/pylauncher.git
```