Science & Technology Facilities Council

Technical Report
RAL-TR-2012-003

Help I need to choose a revision control system!

DJ Worth

February 2012

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358- 6254

Help I Need to Choose a Revision Control System!

D.J. Worth

April 2011

Abstract

In this report we dive into the daunting looking world of revision control systems, *aka* source code control systems or source code management systems. Keeping track of changes in files is something that most software developers want to do. It allows us to see which files were changed if a bug is spotted or results differ from one version of the software to the next. Attempting to do this with file and directory names is guaranteed to fail (well nearly) and consumes time that could be better used developing the software. The tools we will introduce in this note are designed to help software developers do away with manual control and provide easy access to files and their histories. We will discuss the benefits of revision control, introduce the tools and give use cases for each that illustrate how they can be used.

Keywords: revision control, source code control, source code management,

---

Email: david.worth@stfc.ac.uk
Reports can be obtained from www.softeng.cse.clrc.ac.uk

Software Engineering Group
Computational Science & Engineering Department
Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
Oxfordshire OX11 0QX

Library and Information Services
STFC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
Oxfordshire OX11 0QX

Tel: +44 (0)1235 445384
Fax: +44 (0)1235 446403
Email:library@rl.ac.uk

STFC e-reports are available online at: http://epubs.cclrc.ac.uk

Contents

1 Introduction

As with many areas of software engineering, *revision control systems* (RCS) can seem daunting to the outsider as their use seems *de rigeur* in software engineering projects but their actual use can be obscured by the jargon that surrounds them. Confusion arises from the outset as different people use different names for these systems of tracking changes. Some use the phrase *source code management* (SCM) and others *source code control* (SCC) but they refer to the same thing - a framework for tracking changes to files and directories (folders).
In this report we will first of all attempt to persuade those who are sceptical about adopting an RCS that it is a good idea, If this is successful then the reader can go on to the sections containing details of some of the more popular systems available at present. In each section we give a transcript of a short session with the tool being described showing how each is used from the command line (in Linux). Of course a graphical user interface is easier then having to remember the command line syntax so we give a list of GUIs for each tool which readers can investigate at their leisure and select from according to their predilection for operating system, desktop, integration with other tools, and which looks the coolest.
For the purpose of this report the remote repsoitories are all hosted on CCPForge and so URLs are specific to that host.

1.1 Background

All modern revision control systems track changes to files and some track directories as well, keeping their data in a *repository*. Files can be *checked out* of the repository in order to build the software or make changes to the files and then any changes are *committed* back to the repository with a comment to say why the changes was made.
If two developers change the same file then the second to commit their changes will, in all but the simplest cases, be told that there is a *conflict* and be required to *merge* their changes into the new version of the file created by the first developer. A three-way diff application will be useful at this point, e.g. kdiff3 or gvimdiff.
One way of avoiding having to merge changes too often is to create a *branch* of the source code in the repository for the development task at hand and then merge that branch back into the *trunk* only when the task is complete. The implication here is that the trunk version of the source is the most up-to-date code that works correctly and from which a release could be made at any point and this is considered **best practice**.
Current revision control systems are divided into two camps: *centralised* or *distributed*. A centralised system, as the name implies, has one central repository from which developers checkout code and to which they commit, as described above. In a distributed system there is one original repository but other repositories are cloned from it to a developer's local machine and all commits and other changes are done locally. The changes can be pushed back to the repository from which the clone was made. A cloned repository can be made available to be cloned from by other developers since no one repository has anything that marks it as *the* repository. It is up to the development team to define a process for bringing together the source code for the next version of the software.
A distributed system can be used entirely locally[1], within a user's own directory structure, as it is easy to set up repositories for each project's files in a file system. Moving further into the esoteric, it is possible to checkout from a centralised repository and then run a local distributed system to track the minor changes during a development task before the completed change is committed back to the repository of the centralised system (presumably with the commit log from the distributed system as the commit comment).

[1] Sounds odd but it's true.

2 Why Use Revision Control?

The answer lies in remembering the German fairy story of Hansel and Gretel. The children were led into the woods and abandoned to their fate but clever Hansel had left a trail of white stones enabling them to return home.
Think about it .... When you are developing code everything is fine but when you come to test the software and something is broken then how do you retrace your steps towards the last working version to see what went wrong?
If you use a revision control system to track changes as you go then it is easy to retrieve the history of changes to each file making the hunt for the mistake a lot less painful. In addition if you supplied useful comments with each revision then those changes that wont have caused a problem can be quickly dismissed. Going even further, by tagging the set of source files for each release of the software it is straightforward to rebuild the last (presumably) correct version to compare against the new, broken version. Most scientific software does not do anything on its own, it is driven by input data and this data should also be put under revision control so that you can be absolutely sure that the input data has not changed since the last working tests or if it has changed then the revision comment tells you why.
In short revision control can save developers a great deal of blood, sweat and tears when they need to re-visit old versions of files or back out changes that caused the software to break.
There are other fringe benefits to using RCS that should be taken into account.

- A central repository allows for collaborative development.
- A central repository allows use of tools for continuous integration (e.g. with buildbot) or code review for example.
- A local (distributed) repository can be used for keeping track of small changes during a development task so they don't clutter the main repository.

3 RCS in CCPForge

CCPForge (`http://ccpforge.cse.rl.ac.uk`) is a collaborative software development facility for the Collaborative Computational Projects and UK academic scientific software projects. It provides many of the tools that a software development project could require including mailing lists, forums and a release mechanism but the most important is a variety of revision control systems. Each project on CCPForge can choose its own system from those described in the following sections according to the requirements of the project, e.g. there may be an existing repository that can be imported.
It is possible to switch from one system to another though there is no automatic migration of an existing repository to the new format. If this is required then project administrators should contact the CCPForge administrator.
For whichever control system is chosen CCPForge provides browsing of the repository, the commands to check out the code, querying of commits in a specified date range, and a link to the documentation for the system.

4 CVS

The Concurrent Version System, CVS, (`http://cvs.nongnu.org/`) is one of the oldest source code management tools and is still widely used, especially for projects that were started when it was the best system around. It offers version control of files but not directories in a central repository via a client/server idiom and provides some help in merging a file when two people have modified it independently.

CVS keeps separate version numbers for each file (generally in the form x.y) and increments the second digit when a change is committed to the file. It is possible to change the first digit - see the CVS documentation.
In the normal run of events (and it is only those we are considering here) using CVS is very straightforward. It is simply a matter of checking out the code, making the changes and commiting those changes back to the repository. A quick example is given below.

```
$ export CVS_RSH=ssh
$ cvs -d :ext:djw@ccpforge.cse.rl.ac.uk:/cvsroot/singch checkout source/model
cvs checkout: Updating source/model
U source/model/cap.f90
U source/model/condcol.f90
U source/model/dcoef.f90
U source/model/heat.f90
U source/model/inteq.f90
U source/model/singch.f90

$ cd source/model
$ touch tmp.txt
$ cvs add tmp.txt
cvs add: scheduling file ‘tmp.txt’ for addition
cvs add: use ’cvs commit’ to add this file permanently

$ cvs commit -m "Added a file for tutorial purposes"
cvs commit: Examining .
RCS file: /cvsroot/singch/source/model/tmp.txt,v
done
Checking in tmp.txt;
/cvsroot/singch/source/model/tmp.txt,v  <--  tmp.txt
initial revision: 1.1
done
```

Listing 1: A simple CVS session with check out, add a file and commit

5 Subversion (SVN)

Subversion (`http://subversion.apache.org/`), sometimes known by its command line name svn, is a more modern form of centralised source code control, starting as it did with the intent of improving on CVS. To this end Subversion keeps the revision history for directories as well as files, only transmits and stores the differences to files when they are changed, and all commits are atomic, i.e. if the commit for one file fails in a set of changes then no files are changed in the repository.
Subversion revision numbers are different from those in CVS, being a single integer for the whole repository that indicates the number of changes made since development began. The value is incremented on each commit. In this way it is easy to mark a release of a set of files by remembering the revision number of the repository at the time rather than having to store a list of the separate version numbers of all files in the release.
Like CVS, Subversion uses a client/server architecture with a central repository that can be accessed remotely either by http: or a special svn: protocol.

```
$ svn checkout --username djw http://ccpforge.cse.rl.ac.uk/svn/softeng/SEG_Notes
A    SEG_Notes/2010
A    SEG_Notes/2010/CCPForge_Upgrade
A    SEG_Notes/2010/CCPForge_Upgrade/CCPForge_Upgrade.pdf
A    SEG_Notes/2010/CCPForge_Upgrade/CCPForge_Upgrade.tex
A    SEG_Notes/2010/Coverage-Example1
...
A    SEG_Notes/List-SEG-Notes.txt
Checked out revision 485.
```

```
$ cd SEG_Notes/
$ touch tmp.txt
$ svn add tmp.txt
$ svn commit -m "Added temporary file for tutorial purposes"
Adding         tmp.txt
Transmitting file data .
Committed revision 486.
```

Listing 2: A simple subversion session with check out, add a file and commit

6 GIT

Git (`http://git-scm.com/`) is different from CVS and Subversion in that it is a distributed system. This means that a repository can be cloned to your local machine and used for day-to-day revision control without the need for a network connection or a local repository can be created prior to pushing it to a remote server. When (or if) changes are ready to be propagated to other developers the local repository is pushed to the one it was cloned from. Good documentation is available at `http://www.kernel.org/pub/software/scm/git/docs/`.
A word of warning from the Git reference tutorial before choosing git just because it's what all the smart kids are using:

> *This first thing that is important to understand about Git is that it thinks about version control very differently than Subversion or Perforce or whatever SCM you may be used to. It is often easier to learn Git by trying to forget your assumptions about how version control works and try to think about it in the Git way.*

```
$ git config --global user.name "David Worth"
$ git config --global user.email "david.worth@example.com"

$ mkdir myproject
$ cd !$
cd myproject
$ mkdir subdirectory
$ touch test1.txt test2.txt test3.txt subdirectory/test4.txt
$ ls
subdirectory  test1.txt  test2.txt  test3.txt

$ git init
Initialized empty Git repository in /path/to/myproject/.git/
$ git add .
$ git commit -m "Initial import"
[master (root-commit) 5b9aa0e] Initial import
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 subdirectory/test4.txt
 create mode 100644 test1.txt
 create mode 100644 test2.txt
 create mode 100644 test3.txt

$ vi test1.txt
$ git diff
diff --git a/test1.txt b/test1.txt
index e69de29..e9b1475 100644
--- a/test1.txt
+++ b/test1.txt
@@ -0,0 +1 @@
+This is the first edit for the file : DJW

$ git add .
```

```
$ git commit -m "Added first line of text"
[master 15d1051] Added first line of text
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git log
commit 15d10519c7115abb6d9167585b54c7e6a009dede
Author: David Worth <david.worth@example.com>
Date:   Mon Jun 6 13:15:18 2011 +0100

    Added first line of text

commit 5b9aa0ee606e3b9d9178b3cd8b0819010a31d013
Author: David Worth <david.worth@example.com>
Date:   Mon Jun 6 13:11:12 2011 +0100

    Initial import
```

Listing 3: A simple git session in which we create files, a local repository, and commit to that repository

The initial git repository for a CCPForge project is empty and in a state which can be pushed to from an existing repository such as the local one created above. This can be achieved as follows, *but only for an empty CCPForge repository.* In a newly created repository there is only one branch and that is called the master and we use that name in the `push` command.

```
Add remote repository (for this local repository only)

$ git remote add ccpforge ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing

$ git push ccpforge master
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 574 bytes, done.
Total 7 (delta 1), reused 0 (delta 0)
To ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing
 * [new branch]      master -> master
```

Listing 4: Give a name to a remote repository. and push changes to it

To get the repository from CCPForge (in new directory) the command is as follows.

```
$ cd
$ git clone ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing myproject
Cloning into myproject...
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 7 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.
```

If we now modify the copy of the code checked out above the commit is local and then the changes must be pushed back to the remote repository. Note that instead of an explicit `add` then `commit` we use the `-a` option for the `commit` which adds any changed files for us.

```
$ cd myproject/
$ vi test2.txt
$ git commit -a -m "Added text to test2.txt"
[master 2a35be9] Added text to test2.txt
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 300 bytes, done.
```

```
Total 3 (delta 1), reused 0 (delta 0)
To ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing
   15d1051..2a35be9  master -> master
```

Retrieve this change back in the original directory where initial commit and push were done, again using the master branch.

```
$ git pull ccpforge master
From ssh://ccpforge.cse.rl.ac.uk/gitroot/git-testing
 * branch            master     -> FETCH_HEAD
Updating 15d1051..2a35be9
Fast-forward
 test2.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

This has shown how we can work with a single branch but the usual (and some may say *correct*) way to work is to create a branch in the repository for the development task and then merge that branch back into the master branch. A simple demonstration follows.

```
Start with a fresh copy of the repository
$ git clone ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing myproject
$ cd myproject

List the current (local) branches
$ git branch
* master

Create a branch called djw_new
$ git branch djw_new
Check the branches now. * = current branch
$ git branch
* djw_new
  master
Switch to the new branch so we can work in it
$ git checkout djw_new
Switched to branch 'djw_new'

Make some changes to file contents and commit the change.
$ vi subdirectory/test4.txt
$ vi test2.txt
$ git commit -a -m "Make some changes for branch/merge testing"
[djw_new 7602bc5] Make some changes for branch/merge testing
 2 files changed, 20 insertions(+), 1 deletions(-)

If necessary (for collaboration perhaps) you can push this new branch to the
    repository you cloned from (the origin) with
$ git push origin djw_new
Counting objects: 9, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 518 bytes, done.
Total 5 (delta 0), reused 0 (delta 0)
remote: fatal: This operation must be run in a work tree
To ssh://djw@ccpforge.cse.rl.ac.uk/gitroot/git-testing
 * [new branch]      djw_new -> djw_new

Switch back to the master branch and see that that branch does not have the
    changes
$ git checkout master
Switched to branch 'master'
$ vi subdirectory/test4.txt

Now merge changes from djw_new to master. Remember to pull any changes from the
    remote repository
$ git pull
```

```
$ git merge djw_new
Updating 5bf63e0..7602bc5
Fast-forward
 subdirectory/test4.txt |   17 ++++++++++++++++
 test2.txt              |    4 +++-
 2 files changed, 20 insertions(+), 1 deletions(-)

Once the changes are merged we can delete the branch with
$ git branch -d djw_new
```

Listing 5: Simple branch/merge example

7 Bazaar

Bazaar (`http://bazaar.canonical.com/en/`) is another distributed revision control system and works in much the same way as git. The simple session below follows the 5 minute tutorial at `http://doc.bazaar.canonical.com/bzr.2.3/en/mini-tutorial/index.html`.

```
$ bzr whoami "David Worth <david.worth@example.com>"
$ bzr whoami
David Worth <david.worth@example.com>

$ mkdir myproject
$ cd !$
cd myproject
$ mkdir subdirectory
$ touch test1.txt test2.txt test3.txt subdirectory/test4.txt
$ ls
subdirectory  test1.txt  test2.txt  test3.txt

$ bzr init
Created a standalone tree (format: 2a)
$ bzr add
adding subdirectory
adding test1.txt
adding test2.txt
adding test3.txt
adding subdirectory/test4.txt

$ bzr commit -m "Initial import"
Committing to: /path/to/myproject/
added subdirectory
added test1.txt
added test2.txt
added test3.txt
added subdirectory/test4.txt
Committed revision 1.

$ vi test1.txt
$ bzr diff
=== modified file 'test1.txt'
--- test1.txt   2011-05-20 09:36:22 +0000
+++ test1.txt   2011-05-20 09:41:32 +0000
@@ -0,0 +1,1 @@
+This is the first edit for the file : DJW

$ bzr commit -m "Added first line of text"
Committing to: /path/to/myproject/
modified test1.txt
Committed revision 2.
$ bzr log
------------------------------------------------------------
```

```
revno: 2
committer: David Worth <david.worth@stfc.ac.uk>
branch nick: myproject
timestamp: Fri 2011-05-20 10:39:32 +0100
message:
  Added first line of text
------------------------------------------------------------
revno: 1
committer: David Worth <david.worth@stfc.ac.uk>
branch nick: myproject
timestamp: Fri 2011-05-20 10:36:22 +0100
message:
  Initial import

$ bzr push --create-prefix bzr+http://djw@ccpforge.cse.rl.ac.uk/bzr/bzr-testing/
    djw/myproject
HTTP djw@ccpforge.cse.rl.ac.uk, Realm: 'Document repository' password:
Created new branch.
```

Listing 6: A simple bazaar session in which we create files, a local repository and push to a remote repository

Note that the last command needs `--create-prefix` to create the `djw` directory in which directory `myproject` is created. The files and subdirectory in the current working directory end up in directory `/djw/myproject` in the repository. The new branch is call `myproject`.
To get the branch from CCPForge (in new directory) the command is as follows. Note that you cannot get `/djw` as that's not a branch.

```
$ cd
$ bzr branch http://ccpforge.cse.rl.ac.uk/loggerhead/bzr-testing/djw/myproject
Branched 2 revision(s).
```

If we now modify the copy of the code checked out above the commit is local and then the changes must be pushed back to the remote repository.

```
$ cd myproject
$ vi test2.txt
$ bzr commit -m "Test commit to loggerhead branch"
Committing to: /path/to/myproject/
modified test2.txt
Committed revision 3.

$ bzr push bzr+http://djw@ccpforge.cse.rl.ac.uk/bzr/bzr-testing/djw/myproject
HTTP djw@ccpforge.cse.rl.ac.uk, Realm: 'Document repository' password:
Pushed up to revision 3.
```

We can now get these changes back in direcory where the initial commit and branch push were done

```
$ bzr merge http://ccpforge.cse.rl.ac.uk/bzr/bzr-testing/djw/myproject
HTTP ccpforge.cse.rl.ac.uk, Realm: 'Document repository' username: djw
HTTP djw@ccpforge.cse.rl.ac.uk, Realm: 'Document repository' password:
http://ccpforge.cse.rl.ac.uk/bzr/bzr-testing/djw/myproject is permanently
    redirected to http://ccpforge.cse.rl.ac.uk/bzr/bzr-testing/djw/myproject/
 M  test2.txt
All changes applied successfully.
```

8 Mercurial

Mercurial, also known as Hg, (`http://mercurial.selenic.com/`) is another popular distributed revision system that is worth mentioning in this report. It works in very much the same way

as the previous two as can be seen in the following interactive session. A fuller tutorial can be found at `http://mercurial.aragost.com/kick-start/en/basic/`.
First clone a remote repository

```
$ hg clone http://djw45@ccpforge.cse.rl.ac.uk/hg/hg_test
http authorization required
realm: SEGForge Mercurial repositories
user: djw45
password:
destination directory: hg_test
no changes found
updating to branch default
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Now we can create a new file, add it to the local repository and commit the change. Notice the `hg status` shows the change in state of the file as we progress.

```
$ cd hg_test
$ vi README.txt

$ hg status
? README.txt

$ hg add README.txt
$ hg status
A README.txt

$ hg commit -m "Added a first file" -u djw
$ hg status
$
```

Mercurial always wants a user name for commit, hence the `-u djw` but we can set it once and for all in a `hgrc` file. Typically this will be `/.hgrc` and should contain

```
[ui]
username = Firstname Lastname <example@example.net>
```

We can check what we did with

```
$ hg log
changeset:   0:1127890de2aa
tag:         tip
user:        djw
date:        Fri Dec 16 14:36:54 2011 +0000
summary:     Added a first file
```

Now make another change to the file and get Mercurial to show the difference. We can commit the change without giving a username this time because it is defined in the `.hgrc` file.

```
$ hg diff
diff -r 1127890de2aa README.txt
--- a/README.txt        Fri Dec 16 14:36:54 2011 +0000
+++ b/README.txt        Fri Dec 16 14:47:09 2011 +0000
@@ -1,3 +1,5 @@
 Testing Mercurial version control system

+This file will constantly be updated during testing!
+
 DJW December 2011

$ hg commit -m "Make a second change"
```

To get our work back to original repository we have to push as follows:

```
$ hg push
pushing to http://djw45@ccpforge.cse.rl.ac.uk/hg/hg_test
http authorization required
realm: SEGForge Mercurial repositories
user: djw45
password:
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 2 changesets with 2 changes to 1 files
```

Notice that this single push includes 2 *changesets* corresponding to the 2 commits we made to the local repository. A changeset is simply the collection of changes to all files made between one commit and another or between a clone/update and a commit.
Now we will look at creating and merging branches. To create a branch called experimental

```
$ hg branch experimental
```

With no argument this command simply tells us which branch we are on. Branches are not created in the repository until a commit is made. Now edit the file and commit the branch and finally push it to the original repository.

```
$ hg commit -m "Created experimental branch"

$ hg push --new-branch   # Note the option to create the branch
pushing to http://djw45@ccpforge.cse.rl.ac.uk/hg/hg_test
http authorization required
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
```

The original "main" branch is called `default` and we can switch to that branch using the update command. The edits made to the file in the experimental branch will not be there in this branch. (Check that this is so!)

```
$ hg update default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

We can now merge the changes from the experimental branch into the default one. First we can see what the merge will do before actually doing it with the `--preview` option.

```
$ hg merge --preview experimental
changeset:   2:b962660177e8
branch:      experimental
tag:         tip
user:        djw <david.worth@stfc.ac.uk>
date:        Fri Dec 16 15:06:01 2011 +0000
summary:     Created experimental branch
```

Note the `changeset` value in the output above. We can use this to find out what the actual change was (using diff) and so what will be changed during the merge.

```
$ hg diff -c 2:b962660177e8
diff -r 16699d7a8dda -r b962660177e8 README.txt
--- a/README.txt        Fri Dec 16 14:48:38 2011 +0000
+++ b/README.txt        Fri Dec 16 15:06:01 2011 +0000
@@ -1,5 +1,6 @@
 Testing Mercurial version control system

 This file will constantly be updated during testing!
```

```
+This is an update from the experimental branch

 DJW December 2011
```

There is a line to be added which is what we want so we can go ahead with the merge.

```
$ hg merge experimental
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m "Merged from experimental branch"
$ hg push
pushing to http://djw45@ccpforge.cse.rl.ac.uk/hg/hg_test
http authorization required
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 0 changes to 0 files
```

For all the commands there is help available with `hg help`, e.g.

```
$ hg help init
hg init [-e CMD] [--remotecmd CMD] [DEST]

create a new repository in the given directory

    Initialize a new repository in the given directory. If the given directory
    does not exist, it will be created.

    If no directory is given, the current directory is used.

    It is possible to specify an "ssh://" URL as the destination. See "hg help
    urls" for more information.

    Returns 0 on success.

options:

 -e --ssh CMD        specify ssh command to use
    --remotecmd CMD  specify hg command to run on the remote side
    --insecure       do not verify server certificate (ignoring web.cacerts
                     config)

use "hg -v help init" to show global options
```

9 GUI Clients

There are many graphical clients to make these revision control systems easier to use for the commandline-phobic or those without access to a commandline. So many clients in fact that there will be no detailed discussion here just a list with current URLs and brief descriptions.

TortoiseCVS CVS commands integrated with Windows explorer context menu plus views of revision trees and graphical display tools for merging and diffs. `http://www.tortoisecvs.org/`.

TortoiseSVN Subversion commands integrated with Windows explorer context menu plus views of revision trees and graphical display tools for merging and diffs. `http://tortoisesvn.net/features.html`.

TortoiseGIT Git commands integrated with Windows explorer context menu plus views of revision trees and graphical display tools for merging and diffs. `http://code.google.com/p/tortoisegit/`.

TortoiseHg TortoiseHg is a Windows shell extension and a series of applications for the Mercurial distributed revision control system. It also includes a Gnome/Nautilus extension and a CLI wrapper application so the TortoiseHg tools can be used on non-Windows platforms. `http://tortoisehg.bitbucket.org/`.

TortoiseBZR A Windows frontend to Bazaar in the style of TortoiseSVN or TortoiseCVS. `https://launchpad.net/tortoisebzr/`.

CvsGui CVS client aplication for Windows, Mac and Linux allows browsing of repository, file operations and can show graph of file history. `http://cvsgui.sourceforge.net/`.

RapidSVN Multi-platform GUI front-end for the Subversion revision system. `http://rapidsvn.tigris.org/`.

Git Extensions Client including Windows explorer integration, Visual Studio plug-in. Can also run on different platforms under Mono. `http://code.google.com/p/gitextensions/`.

msysgit A distribution of git for Windows that includes explorer integration and a gui client `http://code.google.com/p/msysgit/`.

Giggle A GTK based git client able to visualise and browse the revision tree, view changed files and differences between revisions, visualise summary information for the project and commit changes. `http://live.gnome.org/giggle`.

Git-gui Distributed as part of git, git gui a visual client that allows repository browsing and file actions. See `http://nathanj.github.com/gitguide/tour.html` for a guide on its use.

GitX A GUI for Mac OS X featuring a history viewer and commit GUI `http://gitx.frim.nl/`.

Gitg A Git repository viewer for gtk+/GNOME `http://git.gnome.org/browse/gitg`.

Bazaar Explorer Qt based cross-platform client `http://doc.bazaar.canonical.com/explorer/en/`.

QBzr QBzr provides a GUI frontend for many core bzr commands and several universal dialogs and helper commands. `https://launchpad.net/qbzr/`.

Bzr-Gtk and Olive GTK+ Frontends to Various Bazaar Commands. Currently contains dialogs for almost all common operations, including annotate and visualise (log). Olive, the integrated version control application is also part of bzr-gtk. `https://launchpad.net/bzr-gtk/`.

hgview hgview is a simple tool aiming at visually navigate in a Mercurial repository history. `http://www.logilab.org/project/hgview`.

MacHG OSX GUI for Mercurial providing a graphical way to manage a collection of files, to add things to the collection, to save a snapshot of the collection, to restore the collection to an earlier state and in general to work with the files. `http://jasonfharris.com/machg/`.

RabitVCS A set of graphical tools written to provide simple and straightforward access to the version control systems you use. Currently, it is integrated into the Nautilus and Thunar file managers, the Gedit text editor, and supports Subversion and Git. `http://rabbitvcs.org/`.

Push Me Pull You A graphical interface for a distributed version control systems. Currently it contains proof-of-concept support including the distributed systems described in this report.`http://pmpu.sharesource.org/`.

Meld This is a visual diff and merge tool but has support for the revision control systems mentioned in this report. `http://meldmerge.org/`.

There are a number of **Eclipse plug-ins** for revision control systems beyond the included CVS support, including:

Subclipse `http://subclipse.tigris.org/`

Subversive `http://www.eclipse.org/subversive/`

EGit `http://eclipse.org/egit/`

JGit `http://eclipse.org/jgit/`

Bzr-Eclipse `http://wiki.bazaar.canonical.com/BzrEclipse`

QBzr-Eclipse `https://launchpad.net/qbzr-eclipse/`

MercurialEclipse `http://javaforge.com/project/HGE`

10 Conclusion

The simple conclusion is that there is no *best* revision control system. Indeed, there is probably no best system for each individual, just the one they prefer and that is a difficult decision because the systems have such similar command sets.
One thing that can be said is that a distributed system may be better for the following reasons:

- Distributed systems currently have a great deal of momentum so it is worth learning at least one system.
- A distributed system can be used in a centralised way - but that is not enforced.
- You have a local copy of the who repository to work with.
- Branching is easier (more obvious what it means to branch) so you are more likely to use it.

The last word should be to say that revision control is essential for good software development so it is important to choose a system - which ever it turns out to be.

References