Revision control for TEX documents

An overview

Abstract
Revision control is the management of multiple versions of the same unit of information. Originating in formalized processes in engineering, it was first automated for managing source code for computer software. Since TEX documents are like source code, they lend themselves well to being managed by a revision control system.
Systems like `RCS` and `git` are very suitable for single writers working on their own projects.
More elaborate systems like `CVS` and `subversion` are more suited for groups cooperating on projects. It takes more effort to master them.
For most single users, `git` is the best alternative for multi-file projects, followed by `RCS` for working on single TEX files.

Keywords
revision control, RCS, CVS, subversion, git

Introduction

What is revision control

Revision control is keeping a history of the development of a unit of information.

The first revision control systems were procedures used by draftsmen to differentiate between several versions of drawings or blueprints. Engineering drawings are customarily equipped with a table in the lower right corner, stating the date of a revision, a revision number and a description of the changes since the previous version. While these procedures are helpful, one also had to make a copy of the master drawing before it was changed to keep a full history of the evolution of the drawing.

With the rise of software engineering, similar procedures for tracking the development of computer source code were automated. Programs were written to facilitate storing and retrieving different versions of (plain text) source code files. Changes are usually saved as differences (or diffs) with respect to the previous version, to conserve space.

Since TEX files are plain text files, they lend themselves very well to being managed by revision control systems.

Newer and more sophisticated revision control systems can also handle binary files, like PDF and JPG files efficiently, making them suitable for complex TEX projects.

Modern revision control systems enable several people to work together on a project. This article however will focus on a single person using revision control, since that is envisioned as the most common scenario.

This article focuses on software that is freely available. There are several proprietary systems available but they are usually quite expensive and therefore out of reach of most single users. And as the freely available systems mentioned here are used to manage most of the largest open-source software projects in existence, they are good enough.

Why use revision control

Revision control is primarily useful for maintaining documents that are long-lived and frequently edited. By using a revision control system one can:

- Undo edits, especially deletions.
- Track the history of a document; what has changed and why was it changed. And for documents edited by more than one person; who made the change.
- Work on a single project from multiple machines or with more persons.
- Merge different versions of a document.

Most editors offer an undo option. But this is usually limited to the edits done in the current session. With a revision control system one can easily restore a document or project to an earlier state.

And while it is possible to record the history of a document e.g. in a comment at the start of a file, this requires a lot of discipline and is easily forgotten. Besides, sometimes the record is just not enough; you may want to retrieve an earlier version of a document or project.

A simple system

Probably the most simple system of revision control is regularly copying a file you are working on under a different name, e.g. with the date in the file-

name. So if you are working on a file called `foo.tex` you could store a copy of it at the end of the day as `foo-YYYYMMDD.tex`. The date is embedded in the filename to make it unambiguous and unique. If your project has lots of files, you can wrap them up in a similarly named zip-file or tarball.

This system has the advantage of being simple, since it does not rely on auxiliary programs. As with any other revision control system, it does depend on operator discipline. It is usable on all operating systems.

However, for large projects this system can use an awful lot of storage. One of the author's projects is around 24 MiB[1] in size. Copying that every day would waste a lot of disk space. And it does not facilitate several persons working on a single document or project. Nor does it make it easy to see what has changed between different versions.

RCS

This system was developed in the 1980s by Walter Tichy at Purdue University, as a free and better replacement for the proprietary SCCS.[2] This set of programs deals with single text files. It does not handle binary files well, and it has no concept of a project. It is therefore primarily suited for projects with a limited number of text files. RCS originated as a command-line set of programs for UNIX. It has been ported to almost every UNIX-like system available. A Windows version is also available.[3]

In the following examples the command-line versions of the RCS tools will be used. These examples are not meant to be a replacement for the manual pages that come with the RCS suite, or for the plethora of HOWTO documents available on the Internet.[4,5] A quick search with your favorite search engine for "RCS HOWTO" will turn up a lot of links.

These examples are meant to give you a taste of how a revision control system works. Other systems have different but functionally similar commands.

After creating a new file, it has to be put under control of RCS. This is done with the `ci` command:

```
$ ci foo.tex
RCS/foo.tex,v  <--  foo.tex
enter description, terminated with single '.'
NOTE: This is NOT the log message!
>> Test file.
>> .
initial revision: 1.1
done
```

RCS now asks you to give a description of the file. After giving that, the file is checked in.

A file called `foo.tex,v` has now been created that contains the revision history of the file `foo.tex`. If a subdirectory named `RCS` exists, the revision history file will be put there. Otherwise it will be put in the same directory as the original file. The initial check-in is given the revision number 1.1.

To edit the file, it has to checked out and locked with the `co` command:

```
$ co -l foo.tex
RCS/foo.tex,v  -->  foo.tex
revision 1.1 (locked)
done
```

RCS uses locking to make sure that only one person is editing a file at a given time. This can prove awkward in an environment where multiple persons are working on a project. You cannot check in your changes while someone else has locked the file. And you might undo his changes by checking in yours! Other systems handle this better.

Having checked out the file, you can now use your favorite editor to edit it. Some editors like `emacs` and `vim` are aware of RCS files and have special menus or commands that enable you to check files in and out from the editor.

You can use the `rcsdiff` command to check what has changed in the working file with respect to the last checked in version:

```
$ rcsdiff -u foo.tex
===================================================================
RCS file: RCS/foo.tex,v
retrieving revision 1.1
diff -u -r1.1 foo.tex
--- foo.tex 2007/07/24 16:00:50 1.1
+++ foo.tex 2007/07/24 18:15:08
@@ -1,3 +1,7 @@
-bla
+
+foo
+
+bar
+
 \bye
```

The `-u` option of the `rcsdiff` command selects the so-called "unified diff" format. In the author's opinion this is the easiest to read, because it provides some context to the changes. As you can see, RCS uses lines as the smallest unit. So if one letter changes in a line, the whole line is seen as changed. Lines that are removed are preceded with a "-" in unified diff format, while added lines are preceded with a "+". The line-based difference mechanism used by RCS is what makes it difficult to use with binary files, since non-text files usually do not have meaningful regular line breaks.

If you are finished editing for the day, or if you feel that you have reached a stage in your document that

you want to save, you check in the changes:

```
$ ci -u foo.tex
RCS/foo.tex,v  <--  foo.tex
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.'
>> This is the log message.
>> .
done
```

Using the `-u` option of the `ci` command enables you to go on editing directly. Without it you would have to check out and lock the file again.

CVS

CVS[6] was developed from RCS by Dick Grune in the mid-1980s to handle projects consisting of multiple files in directory trees instead of single files.[7]

Unlike RCS, it can use a client-server architecture and it separates the place where the revision history is stored (called the repository) from the working directory. The server (and the repository) do not have to reside on the same machine as the client (and working directory), making collaboration over a network easier.

The downside is that it is more complicated to work with than RCS. CVS cannot attach a revision to moving or renaming a file or directory. CVS uses a centralized server if multiple persons are working together on a project. For most single-user TEX projects, this will be an overweight solution. The added features of CVS come at the price of added complexity. And since it is based on RCS, it still doesn't handle binary files well.

Many open source software projects use CVS to maintain their code.

Subversion

Advertised as "CVS done right", subversion[8] aims to be a compelling replacement for CVS. This system solves a lot of the problems of CVS; it handles binary files and it can handle file and directory renames. But like CVS it is probably overkill for a small project. And it shares with CVS a steep learning curve, and the need for separate repositories.

Git

Git[9] was developed when the developers of the Linux kernel lost access to the proprietary BitKeeper system[10]. Development was started in April 2005 with version 1.0 being released in December 2005. The Linux kernel development has been managed with git since June 2005.

Unlike RCS, git tracks a directory tree of files. It also handles binary files. But like RCS, a git-managed directory is completely self-contained. It does not require external repositories. It has few dependencies, stores its data in compressed form and is quite fast. For instance, the complete revision history of the 24 MiB project mentioned before which spans two years and 99 revisions is only 12 MiB. It is also a distributed system. Every directory (repository) is self-sufficient, but synchronizing between them is easy. So it is both easy to work with for a stand-alone user, and for a group of people working on the same project.

Because of its lightweight nature, directory tracking and handling of binaries, the author now prefers it over using RCS.

An example of working with git follows. In this article it is impossible to showcase the complete functionality of git, since there are more than a hundred and forty git commands. Manual pages for all the git commands, as well as several tutorials and HOWTOs are included in the git distribution[6].

To create a git repository, change to the directory whose contents you want to monitor, and type:

```
$ git-init-db
Initialized empty Git repository in .git/
```

This creates a `.git` subdirectory that git uses to store all data.

The `git status` command shows the state of the repository.

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include
in what will be committed)
#
# Makefile
# rc.tex
nothing added to commit but untracked
files present (use "git add" to track)
```

Now one has to point out which files you want to monitor:

```
$ git add Makefile rc.tex
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file: Makefile
```

```
# new file: rc.tex
#
```

To save this initial state of the files, commit them:

```
git commit -a \
-m "Initial check-in of Makefile and rc.tex."
Created initial commit 9d3de62:
Initial check-in of Makefile and rc.tex.
 2 files changed, 357 insertions, 0 deletions
 create mode 100644 Makefile
 create mode 100644 rc.tex
```

The `-a` flag indicates that all changes should be committed, while the `-m` flag is followed by the commit message.

With the `git log` command one can list the commits:

```
$ git log
commit 9d3de6265ad4fc0df2ca108b7918f6283dc18d59
Author: Roland Smith <rsmith@slackbox.xs4all.nl>
Date:   Thu Jul 26 19:03:57 2007 +0200

    Initial check-in of Makefile and rc.tex.
```

Every commit is identified by a SHA1 hash of its contents. This is also used to check against data corruption. Next is the name and e-mail address of the person who checked in this commit. The date of the commit and the commit message are also listed.

If one compiles the TEX file, some extra files are created that git doesn't know about:

```
$ git status
# On branch master
# Changed but not updated:
#
# modified:   rc.tex
#
# Untracked files:
#
# rc.aux
# rc.log
# rc.pdf
no changes added to commit
```

Normally, we would like git to ignore those files. So we add them to the text file `.git/info/exclude`, and git will ignore them from now on.

The command `git diff` is used here to show the changes in the monitored files since the last commit. A small piece of the diff is shown below. It uses the so-called unified diff output style.

```
$ git diff
diff --git a/rc.tex b/rc.tex
index e3eaefe..3aac05d 100644
--- a/rc.tex
+++ b/rc.tex
@@ -1,5 +1,5 @@
 % -*- latex -*-
-% Time-stamp: <2007-07-26 19:01:53 rsmith>
+% Time-stamp: <2007-07-26 19:20:47 rsmith>
 % Copyright © 2007 R.F. Smith <rsmith@xs4all.nl>
 %
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
@@ -292,10 +292,11 @@ to work with for a stand-al
 the same project.

 Because of its lightweight nature, directory
 tracking and handling of binaries, the
-author now prefers it over using RCS.
+author now prefers it over using RCS. An
+example of working with git follows.
...
```

By supplying `git diff` with one or two commit-id's, you can also show the differences between the named commit and the working files, or between two named commits.

Conclusion

For documents that consist of a single or a small number of TEX files and other text files, RCS is probably the best solution because of its simplicity.

For larger projects, git is a better option. Especially if they contain binary files in e.g. PDF or JPG format that need to be controlled. Or if you want to collaborate with others on a project.

Systems like CVS or subversion will probably appeal to larger organizations that require centralized development.

Notes

1. http://en.wikipedia.org/wiki/MiB
2. http://en.wikipedia.org/wiki/Revision_Control_System
3. http://www.codeproject.com/tools/cs-rcs.asp
4. http://www.madboa.com/geek/rcs/
5. http://www.athabascau.ca/html/depts/compserv/webunit/HOWTO/rcs.htm
6. http://www.nongnu.org/cvs/
7. http://en.wikipedia.org/wiki/Concurrent_Versions_System
8. http://subversion.tigris.org/
9. http://git.or.cz/
10. http://en.wikipedia.org/wiki/Git_(software)

Roland Smith
rsmith@xs4all.nl