

Versioning systems tutorial

with SVN

Samuel Colin¹

LORIA/DEDALE

1 Introduction to VSes

1.1 A bit of history

In the beginning, software developers wanted to know the differences between two files modified differently by two developers, hence the `diff` and `patch` tools. Then they needed to keep track of all those changes, hence `RCS` the Revision Control System.

Then they wanted to keep track of the history of a whole project, which means keeping track of several files at the same time. Hence they built `CVS` on top of `RCS`. Then developers were happy for a while. In the meantime, software grew bigger and bigger. `CVS` started to be too poor for tracking efficiently some projects, applying modifications, etc. `CVS` did not take into account architectural changes, like renaming parts of a project, moving things around, etc. The best example of this is the Linux kernel. Linus turned to `bitkeeper`, a `VS` (Versioning System) that had more features than `CVS`.

Open-source developers were not happy with this situation:

- Linux was developed with a proprietary tool. Which means Linux depended partly on a commercial company.
- Linux had no convincing free `VS` to be developed with

Then began the explosion of `VSes` (for the record, `Bitkeeper`'s change of licence forced Linux developers to build their own `VS`), see figure 1.

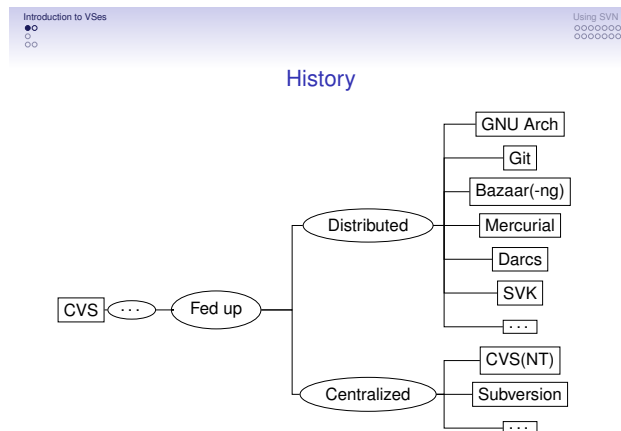


Figure 1: A short history of versioning systems

Versioning systems can be split in two categories:

- Centralized: all local repositories depend on the same central repository. This is a star-shaped kind of connection
- Distributed: each local repository holds the whole history (from the time of the first “connection”) and can act as central repository itself. Depending on the deatures of the VS, the connection is tree-shaped or graph-shaped.

There are also various characterizations for VSEs: how they prevent (or resolve) conflicts, how they keep track of the history,etc. Let us note that Darcs is the only formalized VS, i.e. the ideas behind patches/changesets have been put into a theory.

What does a VS do ?

- Keeps track of modifications: who, what, when, how ?
- Gives a way for several developers to intervene on a same project at the same time
- Gives tools for solving contradicting modifications (conflicts)
- Identifies interesting versions of the project
- Eases more thorough changes of the project (branching)...
- ... and the reuniting with the official version of the project (merging)

A VS is a mean for developers to work on a same project at the same time. The granularity depends on the VS, but usually it is the same as of the `diff` tool, which means the line of text. This means that the optimal data for developing with a VS is text. Binary files can be used but will eventually clutter the history and affect the performances. That is why a good policy is that each committed version shall leave the project in a compilable state.

Usually most VSEs will help keeping track of the changes, so that one can ask details about some change to the developer that contributed it.

Sometimes, especially on the core files of a project, modifications brought by different developers happen on the same files and on the same parts of the same files. When contributing the change, either the VS detects that the changes are similar (very difficult for a computer to detect that) or the VS signals conflicts to the last developer committing offending modifications. Then he will have to solve the conflict, often by speaking to people who contributed modifications contradicting his own modifications. Once the conflicts are solved, the developer uses the tools of the VS to signal a solution has been found and he commits the files.

When developing software it is interesting to identify versions of the project in the VS that correspond to released versions, for instance. It is possible to *tag* such versions with a relevant name (like `software-release-x.y.z`). Each tag will be assigned to given versions of the files being versioned.

Big software projects often require more subtle handling. For instance it happens that critical changes must be made to the functional core, but it is not acceptable to refuse changes from other developers (for instance, bugfixes). Hence with VSEs it is possible to *branch* to build a version that will diverge from the main development, while the critical changes are made. The main *trunk* receives the bugfixes while the critical changes are made to the branch. It is possible to bring the branch the bugfixes through *merging*, i.e. bringing changes applied to one branch to another branch. When the critical changes are done, the diverging branch is *merged* back to the main trunk.

VSEs also have secondary abilities that depend more on the VS itself. We won't speak about that here. Now how goes by the day of someone using a versioning system ?

1.2 Typical work with a VS

The typical workflow

- Checkout of the repository (*I want to contribute*)
- Loop:

- Update (*what did others change ?*)
- Modify
- Update + conflicts resolution
- Commit (*I validate my changes*)
- Repeat Loop

A developer wants to contribute. First he has to plug himself to the versioning system of a project. This initial step is called a *checkout*. Then contribution follows the cycle presented previously. The modifications can be anything: adding a file, removing it, renaming it or simply changing its contents.

Then he updates to acknowledge the modifications of other developers and see if it contradicts some of his changes. If it does, he resolves the conflicts (by editing problematic parts, undeleting files, etc). He updates once more to acknowledge the modifications etc. What you see here is that conflict apparition is a showstopper: hence it is crucial that contributors know what they are supposed to modify. With VSes, *communication is key*.

Finally, once no more conflicts has appeared, the developer validates the changes so that they are taken into account in the central repository of the project: he *commits* his changes. Then he goes back to updating, contributing changes, and so on.

1.3 Closing words about VSes

References

- http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

A few last words

- A versioning system is *not* a backup system
- A versioning system does not replace communication
- A best practice is to leave the central repository in a compilable state

A good backup system saves important files regularly. A versioning system keeps a history and a trace of all “working” versions of your files, i.e. files that compile or files the content of which you are happy with. You might as well be working on important files and made serious modifications, as long as it is not committed, it is not backed up¹.

Versioning systems also give a way for working around or resolving conflicts. But conflicts should not appear *in the first place*. Ensure communication happens between all the contributors so that everyone knows who does what. Avoiding conflicts leads to faster development. Merging is also a source of conflict, but this is expected because it corresponds to (temporary) divergent developments reunited later.

Especially with bigger projects, it is crucial that a project is compilable. Indeed, if you contribute only parts of a modification and the project doesn’t compile anymore, the development is slowed down, or event halted if you are the only person developing that part, i.e. you are the only person with the knowledge to put things back on.

2 Using SVN

2.1 A typical work with SVN

Following the workflow depicted in section 1.2, here follow the commands used in the everyday life of the svn-aware contributor:

¹Actually, some persons use versioning systems for backups. But that’s because they want to keep a history of what happens to their files, because they are the only contributors and because they know what they do

The typical workflow: the actual commands

- `svn checkout protocol://directory local-directory`
- `cd local-directory`
 - `svn update`
 - **Modify. Can be:**
 - * `svn add some-file`
 - * `svn remove file`
 - * `svn move file1 file2`
 - * `svn copy some-file`
 - * **Modify files**
 - `svn update` (there might be conflicts, see later)
 - `svn commit`
 - Repeat

`svn help` for details about the commands

The various command names are self-explanatory. The commands for modifying the files are viewed as a svn-aware version of their unix counterparts, i.e. the svn repository is made aware that a file is the same as another because it has been moved, etc. We explain in further detail some of the more cryptic commands:

checkout This command is used to link a local repository (where we will work from now on) to the central repository. The central repository might be accessed through a variety of protocols: local file, http, svn over ssh, etc hence the `protocol://directory` form

update This command synchronises the local files with the central repository, so that we can take into account in our work the modifications of others. It might even modify files we are working on, causing a conflict (what we modify contradicts what another contributor has done) or not. In the latter case, *it might still affect the consistency of the project*, thus we have to ensure we can still compile the project

commit This command updates the central repository with our modifications. If someone was faster than us, it might cause again conflicts.

A lot of the workflow with Subversion revolves around (the possibility of) conflicts.

The typical workflow: cases of conflicts

- On an update: It will look like this

```
C    test.txt
Actualisé à la révision 2.
```

- On a commit: Subversion complains

```
Envoi      test.txt
Transmission des données .svn: Échec de la propagation (commit), détails :
svn: Le chemin 'test.txt' est obsolète dans la transaction '3-1'
```

Solution: do an update

We shall next see how to resolve a conflict.

Some special things happen when a conflict is detected:

Conflicts: what has happened ?

svn status leaves us with:

```
?      test.txt.r2
?      test.txt.mine
?      test.txt.r1
C      test.txt
```

Subversion has attempted to merge the modifications from the central repositories, with some modifications contradicting our own: the cases of conflicts. Subversion then produces the following files:

Conflicts: what has happened (cont'd)

What are these ?

test.txt.r2 The version of the central repository

test.txt.mine My version has I attempted to commit it

test.txt.r1 The last version up-to-date with the central repository

test.txt My file, with conflicts pinpointed inside like this:

```
<<<<<<< .mine
Youhou !
=====
Awesome !
>>>>>>> .r2
```

Inside the conflicting file, conflicts are denoted with a special syntax, easy to search with your preferred text editor. The upper part is your modification, the lower part the modification other people did to the same part of the file (obtained from the central repository). We can still commit the other non-conflicting files, but no matter what, Subversion won't commit the files with conflict until we have told him that the conflicts are resolved.

Conflicts: resolving

- Replace the part between <<<<<<< and >>>>>>> with something relevant
- Repeat the previous step for all conflicts inside a file (there might be several !)
- Repeat the previous step for all conflicting files
- `svn resolved conflicting-file(s)`

Finally you can commit your changes ! (and there might be some more conflicts)

That is, in a nutshell, how things happen for the everyday Subversion-aware developer/contributor. You can do other things with Subversion that aren't covered here (yet):

Subversion: other features

- Automatic variables (special variables replaced by the date of last modification, version, etc)
- Tagging (tag a version of specific interest)
- Branching (i.e. forking the development in order not to disturb other developers)
- Merging (contributing back the changes made in a branch)

A few references

- [Collins-Sussman et al., 2007]
- `svn help`
- <http://wiki.loria.fr/wiki/Subversion%40dedale> (see the part with automatic variables)

2.2 Tutorial

This tutorial is meant as a quickstart towards good practices when using a versioning system, using `svn`. It should be easily adaptable to other systems, provided the particularities of the centralized/distributed approaches are taken into account.

Creating a repository, both central and local

- Create the central repository (the tutorial monitor has already done this)
 - `umask 007` (group-based authorization, Unix filesystem)
 - `svnadmin create SVN-test` (path: `/users/dedale/colinssa/pub-dedale/`)
- Checkout out the repository and get ready
 - `cd some-directory`
 - `svn co svn+ssh://login@greny/users/dedale/colinssa/pub-dedale/SVN-test svn-tutorial`
 - `cd svn-tutorial`

How the tutorial will be held

- We will build a LaTeX document presenting each member of the tutorial session
- Because we use various languages and because the session monitor is perverse, we will use Unicode as the input encoding (Emacs users need not worry)
- To each person will correspond a section he/she is responsible of
- The content of each section will be saved in a file
- Each file will be an input for the \LaTeX main file

```
% -*- coding: utf-8 -*-
\documentclass{article}

\usepackage[utf8x]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{pslatex}

\title{SVN tutorial session}
\author{DEDALE}

\begin{document}
\maketitle

\begin{abstract}
\end{abstract}

\end{document}
```

Emacs: \LaTeX → Multifile/parsing → Set master file

Follow this outline

- Write your own section in your own file (find a consistent naming, put `% -*- coding: utf-8 -*-` at the beginning)
- Add it, commit your change
- Reference it in the main file
- Commit your change
- Two persons will write the abstract independently and commit
- One person will add a french(?) translation for each section
- One person will make an introduction presenting the various sections
- One person will spell-check everything
- Get to work !

Don't forget

- `svn update`
- **Modify** (add, move, remove, copy or file content modifications)
- `svn update`
- If there are conflicts:
 - `svn status` to see the conflicting files (C in the left column)
 - Edit the offending file(s)
 - `svn resolved offending-file` for each file
- `svn commit`
- Note: when committing, *use a good commit message*

What happened so far ?

Two things might have happened:

- The repository is now a mess and people are still discussing about how to correct things
⇒ You have not worked enough. Go on !
- The repository is super-duper-clean, and a hierarchy has arisen
⇒ Congratulations, you just understood (if subconsciously) how to get things done with a versioning system

Of interest to those who dislike command-line

There are graphical interfaces to Subversion

- TortoiseSVN
- RapidSVN
- SCPlugin
- SmartSVN
- ...
- See http://en.wikipedia.org/wiki/Subversion_%28software%29

Next time

- Tagging
- Branching
- Merging
- Special properties

References

[Collins-Sussman et al., 2007] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2007). Version control with subversion. website. <http://svnbook.red-bean.com/>.