"The core mission of a version control system is to enable collaborative editing and sharing of data."

Subversion guidelines

Jari Häkkinen

`$Revision: 55 $`
`$Date: 2021-08-27 15:43:22 +0200 (Fri, 27 Aug 2021) $`

Contents

This document aims at giving developers a very short introduction to version control systems in general, and subversion in particular. Most of the material is collected from the book "Version Control with Subversion."[1] To be honest, many sentences and paragraphs in this documents are outright stolen from the subversion book.

For a more detailed introduction to revision control systems and its concepts, refer to the subversion book.

1 Introduction

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Version control systems usually manage to keep track of changes and merge them together appropriately. However, there are cases when users do overlapping changes and the system cannot resolve the contradictory changes. This situation is called a conflict, and it's usually not much of a problem. When a user asks his client to merge the latest repository[2] changes into his working copy[3], his copy of file with conflicting changes are somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes, and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once the overlapping changes are (manually) resolved – perhaps after a discussion with the party who made the conflicting changes – the merged file can safely be committed into the repository.

One way to avoid conflicting changes in files is to introduce locking of the repository while someone updates files[4]. This will prohibit concurrent changes of files, and keep conflicts to a minimum, but at the same time it becomes a nuisance since only one person in the team can make changes at the time. Of course, others can also work on the project and wait until they get to lock the repository for their commits, but after that they checked (sometimes without the support of a version control system) that their changes does not contradict other changes made before they got the locking power.

In the end, it all comes down to one critical factor: communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false promise that a locking system will somehow prevent conflicts.

[1] http://svnbook.red-bean.com/

[2] The repository is a database containing all files and directories handed over to subversion control. The repository also stores information about changes, log messages, who did what and when, and so on. The user of subversion will never tamper with the repository directly, but rather use a client for all actions he wishes to perform to subversion controlled files.

[3] The working copy is whatever was checked out from the repository. Different users have their own working copies, and users may even have several different working copies of a repository.

[4] This is how RCS works, or sharing of binary file formats as Microsoft Word

1.1 Branching patterns

For projects that have a large number of contributors, it's common for most people to have working copies of the trunk[5]. Whenever someone needs to make a long-running change that is likely to disrupt the trunk, a standard procedure is to create a private branch and commit changes there until all the work is complete.

Of course, if you need to go back and fix bugs in previous releases of your software, you will need to checkout the revision used for that release, and make necessary changes and store the changes as a branch in the subversion tree.

1.1.1 Release branches

Most software has a typical life cycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality-assurance teams take time to test supposedly-stable versions of the software. Continued work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bug-fix without having to wait for a major new release.

Branching using subversion as suggested by the subversion book:

Developers commit all new work to the trunk. Day-to-day changes are committed to `/trunk`: new features, bug-fixes, and so on.

The trunk is copied to a "release" branch. When the team thinks the software is getting ready for release, then `/trunk` might be copied to `/branches/1.0-stable`.

Teams continue to work in parallel. One team begins rigorous testing of the release branch, while another team continues new work on `/trunk`. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is "frozen" for final testing right before a release.

The branch is tagged and released. When testing is complete, `/branches/1.0-stable` is copied to `/tags/1.0` as a reference snapshot. The tag is packaged and released to customers.

The branch is maintained over time. While work continues on `/trunk` for next version, bug-fixes continue to be ported from `/trunk` to `/branches/1.0-stable` (or the other way around). When enough bug-fixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0-stable` is copied to `/tags/1.0.1`, and the tag is packaged and released.

[5] The trunk is the main development line in the repository, in contrast to branches that exist in parallel to the main development line.

1.1.2 Feature branches

We insist on that `/trunk` and release branches compile and pass regression tests at all times. A feature branch is only required when changes require large numbers of destabilising commits. A good rule of thumb is to ask this question: if the developer worked for days in isolation and then committed the large change all at once (so that `/trunk` were never destabilised), would it be too large a change to review? If the answer to that question is "yes", then the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

To avoid branches to grow to far apart from each others, the feature branches must be kept in sync with the trunk. We require that branch synchronisation is performed regularly (once every week) against the trunk. Remember to write proper log messages to keep track of merges.

When the development branch is kept in sync with the trunk it is straight forward to port the branch back to the trunk since all differences between the branch and the trunk are readily made in the branch. Basically, all that needs to be done is to merge by comparing the branch with the trunk.

1.1.3 Vendor branches

If a project depends on someone else's information, there are several ways to attempt to synchronise that information with the project. Most painfully, one could issue oral or written instructions to all the contributors of the project, telling them to make sure that they have the specific versions of that third-party information that the project needs. If the third-party information is maintained in a subversion repository, one could use subversion's externals definitions to effectively "pin down" specific versions of that information to some location in your own working copy directory (see the section called "Externals Definitions" in the subversion book).

The solution to this problem is to use vendor branches. A vendor branch is a directory tree in the version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that is decided to be absorbed into the project is called a vendor drop.

Vendor branches provide two key benefits. First, by storing the currently supported vendor drop in the version control system, the members of the project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Secondly, because the data lives in the subversion repository, we can store your custom changes to it in-place – there is no more need of an automated (or worse, manual) method for swapping in customisations.

Managing vendor branches generally works like this. Create a top-level directory (such as `/vendor`) to hold the vendor branches. Then the third party code is imported into a sub-directory of that top-level directory. This sub-directory is copied into the main development branch (for example, `/trunk`) at the appropriate location. Local changes are always made in the main development branch. With each new release of the code we are tracking we bring it into the vendor branch and merge the changes into `/trunk`, resolving whatever conflicts occur between local changes and the upstream changes.

1.1.4 Subversion

Each time the repository accepts a commit, this creates a new state of the file system tree, called a revision. Each revision is assigned a unique natural number, one greater than the number of the previous revision[6]. The initial revision of a freshly created repository is numbered zero, and consists of nothing but an empty root directory.

It's important to note that working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. This happens if you commit a changed file, this file will get a new revision number while the rest stays at their current revision. The revision discrepancy will also happen if you `svn update` specific items in your working copy. To bring everything on par with the latest repository revision you must do an `svn update` in working copy root directory level.

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can make a more accurate log message. You may also discover that you've inadvertently changed a file, and this gives you a chance to revert those changes before committing. Additionally, this is a good opportunity to review and scrutinise changes before publishing them.

1.2 Donate Your Changes

[7]After making your modifications to the source code, compose a clear and concise log message to describe those changes and the reasons for them. Then, send an email to the developers list containing your log message and the output of `svn diff` (from the top of your subversion working copy). If the community members consider your changes acceptable, someone who has commit privileges (permission to make new revisions in the subversion source repository) will add your changes to the public source code tree. Recall that permission to directly commit changes to the repository is granted on merit – if you demonstrate comprehension of subversion, programming competency, and a "team spirit", you will likely be awarded that permission.

2 Subversion usage rules

There are not many requirements put on us developers with respect to subversion usage, so we should actually be able to follow them. If you feel that any of these rules are stupid

[6]In contrast, CVS have revision numbers on every file under control.

[7]This subsection is written for developers without write access to the repository.

and as such should be changed, feel free do discuss them at our group meetings. However, until we agree on changing rules, we must obey them.

2.1.1 All checks should work after a commit

Before you make a commit to the trunk, you must make sure that all scripts, compilations, regression tests, documentation generations, or whatever needed in the project works as expected. There is nothing more frustrating than running into someone else's problems when you are about to solve that memory leak you have been looking for since 2pm yesterday. One way to make sure that your last commit did not break things is to actually have another pristine checkout of the project. This pristine working copy should only be updated against the repository for testing after your latest commit. If all checks run okay in this pristine tree you probably did a proper commit from your development working copy.

If you feel that you cannot fulfil this requirement there are two options. i) Create a private branch, fix whatever you are doing, and merge the changes into the main trunk when everything works again. ii) Ask the rest of the development team's permission to leave this rule for a **very** short time.

2.1.2 Do not check in personal debug code into the main trunk

If you must check in personal debug code, create your own branch and keep this branch and the main trunk in sync (see Appendix 1.1 on how to do this).

2.1.3 Commit often and do it minimalistic

Make sure that you stay up to date with the repository, *i.e.*, commit often and remember to issue `svn update`.

When you decide to make a commit, make it small and to the point. By this we mean that commits should only contain whatever was needed to solve a problem or to add a feature. There is no use in committing trivial unnecessary changes into the repository such as stray blank lines or extra white space characters in the end of a line (CR – CRLF conversions are especially annoying and can be avoided by proper setup of your subversion environment).

2.1.4 Write a log message when committing

Log messages are useful for other developers when they want to know what was done in a commit. You may think that trivial changes can easily be "diffed" and thus need no log message. However, diffs requires more work than reading log messages, and remember, log messages for trivial changes are easy to write.

For projects that use trac[8] for software project management there is a possibility to close or add comments to trac tickets directly when committing changes to the repository. This is done by writing a properly formatted log message, cf. A.20.

2.1.5 Set up proper subversion `config` file

Cut and paste the configuration setup from Appendix F into your subversion configuration file `$(HOME)/.subversion/config`

2.1.6 Use the `$Id$` keyword to identify files

For all text files we modify and maintain, we must add `$Id$` near the top of the file for easy identification of the file and its latest revision change. If you set up a proper subversion configuration file this requirement becomes trivial. The only thing needed then is to add `$Id$` into source files.

2.1.7 Read and understand this document

This should not need to be a rule, but ... There are a few concepts which you may have neglected previously such as branches.

Read the branches section in the introduction and make sure you understand it. If you get confused, ask someone. Note that our release and branching procedure is defined in the branching section.

2.1.8 Project directory structure

In order to make future branching of projects under subversion control seamless, one must plan ahead when creating the project repository. In subversion this means that a proper project directory structure must be set up at the initial import of a project into subversion control.

The directory structure adopted for the project must always contain these three directories at root level

```
/trunk
/branches
/tags
```

The `trunk` directory holds the "main line" of development, the `branches` directory contain branch copies, and the `tags` directory contain tag copies.

This structure is needed for the branching procedure we are adopting (cf. Appendix 1.1), and the recommended structure in the subversion book.

[8] http://projects.edgewall.com/trac/

2.1.9 Do not mess with the `tags` directory

Never make commits in the tag directory, it exists only for keeping track of tags[9].

A Common operations

In true Perl spirit there are more than one way to perform the below tasks. It is also assumed that the subversion repository is residing on a server (here, `svn.example.com` is running a web server with svn support), and *calc* is used as the example project. Furthermore, since we are anticipating branching of the project, we must prepare our repository for branching. Command examples are typed as

```
command --with-option example
```

and

```
sample output is typed like this
```

A.1 Setting up a repository

Setting up a new personal repository is straight forward, do

```
svnadmin create /path/to/svn/repos/project
```

and subversion is now ready accept addition of projects into the repository. If the repository resides on a server, then you need access to server and do some more initialisation steps (see Appendix E.1).

A.2 Handing over control to subversion

If the repository is empty (still at revision 0), you start by creating the initial directory structure to be imported into the repository, and (optionally) add files to be imported.

```
mkdir /tmp/myproject/branches
mkdir /tmp/myproject/tags
mkdir /tmp/myproject/trunk
cp ...
```

Import the project with

```
svn import /tmp/myproject http://svn.example.com/calc -m "Initial import"
```

Subversion does not store the root name `/tmp/myproject`, so these are only temporary names.

[9] A tag is a marked point in the evolution of the project, an alias for a specific revision of the repository. For humans, it is easier to remember a tag such as *release-1.0* than revision number like 1255. Moreover, a conscious choice of tag name makes it easy to remember why revision 1255 was interesting.

A.3 Initial checkout of a project

Assuming we want to checkout the main trunk of the project a simple

```
svn checkout http://svn.example.com/calc/trunk calc
```

will do. This will create a sub-directory `calc` containing the latest revision of the project.

A.4 Set up of editor to use for log messages

There two ways to set your favourite text editor for log message editing. Either you define it in the `config` file, or set one of the environment variables SVN_EDITOR, VISUAL, or EDITOR to point at your preferred editor. `svn` will use the first one it finds in the order we described the different possibilities here.

A.5 Undoing changes

If we realize that a change committed to the repository is unwanted we can reverse these by reversing the order of revisions;

```
svn merge -r 303:302 http://svn.example.com/calc/trunk
```

Check the changes, and commit if you are happy with them. Magic!

A.6 Undoing `svn add` (and `svn delete`)

Sometimes we forget that adding items to subversion control is a recursive action, and as consequence end up with a huge import of files. The easiest way out of this is to use the revert sub-command

```
svn revert item
```

where item is whatever you added (or item can be empty, and everything in the current directory will be reverted). Note, this will actually revert all uncommitted changes not just whatever you added.

A.7 Resurrecting deleted items

For this the best solution is to use `svn copy` as

```
svn copy -r 807 http://svn.example.com/calc/trunk/real.c ./real.c
```

The added bonus with this is that subversion will also remember the file history.

A.8 How do I make subversion ignore items when issuing `svn status`

You can also set what files `svn` should ignore when running `svn status` in the config file (see Appendix F).

A.9 Can I add a symbolic link to subversion control

Yes, subversion handles symbolic links (in operating systems where these make sense) automatically, just treat them as any other item.

A.10 Creating a branch

The simplest way to create a branch is to make a copy of the current trunk HEAD (or another revision in the tree) by issuing

```
svn copy http://svn.example.com/calc/trunk \
http://svn.example.com/calc/branches/my-calc-branch \
-m "Creating a private branch of /calc/trunk."
```

This will create a shallow copy, meaning that subversion creates links within the repository until files are actually changed. In consequence branches are inexpensive, and you should not be afraid to create branches. You have to check out the new branch to get access to it. The changes needed to the above command when creating a branch from a tag is straightforward.

You can also create branches within your checked out project (see more in the svn book). There are no real branching in subversion, there are different directories with files that share a common history. Branches is an attribute that the users of subversion attach to the branches directory.

A.11 Porting changes between branches

Sometimes you want to copy changes between branches and this is is accomplished by merging different revisions. Issuing

```
svn merge -r 343:344 http://svn.example.com/calc/trunk
```

will perform a local update of your branch using the difference between the revision specified. You should review the changes before committing them into your repository branch. If something went wrong or is unsatisfactory, use `svn revert`. However, revert may not be able to perform well in some cases.

A.12 Merging a stable branch into the main trunk

Assuming we have a branch that only contains change sets from the branch itself, *i.e.*, no change sets have been ported from trunk to the branch. Now we want all changes made to the development branch to be ported into the main trunk. To accomplish this we need an up to date working copy of the main trunk. The procedure is

```
cd calc/trunk
svn update
At revision 405.
svn merge http://svn.example.com/calc/branches/my-calc-branch
--- Merging r336 through r405 into '.':   ...examine the diffs, compile,
test, etc...
svn commit -m "Merged my-calc-branch changes into the trunk."
Committed revision 406.
```

where some output has been removed. Note that only changes committed into the repository are available for porting. At a later point when more development has occurred in the stable branch, we want to port these changes as well. This is equally easy since the repository remembers which change sets have already been merged into the main trunk, and the the procedure is simply

```
cd calc/trunk
svn update
At revision 513.
svn merge http://svn.example.com/calc/branches/my-calc-branch
--- Merging r406 through r513 into '.':   ...examine the diffs, compile,
test, etc...
svn commit -m "Merged my-calc-branch changes into the trunk."
Committed revision 514.
```

A.13 Keeping a feature branch in sync

Having a feature branch, it is recommended to keep it in sync with the trunk, *i.e.*, port changes in trunk into the feature branch.

```
cd branch-working-copy
svn merge http://svn.example.com/calc/trunk
```

The repository knows when the branch was created from the trunk and will only merge change sets added to the trunk after the branch was created. After some time there has been some significant development in trunk and you want to port these to the feature branch. This is as easy as before

```
cd branch-working-copy
svn merge http://svn.example.com/calc/trunk
--- Merging r354 through r501 into '.':
```

and you do not need to worry that changes will be ported twice. The repository remember which change sets have already been merged to the branch and will not merge these again.

A.14 Merging a feature branch into the main trunk

Assuming that you followed the convention of keeping your feature branch in sync with the trunk, all that is needed is

```
cd trunk-working-copy
svn update
At revision 1910.
svn merge --reintegrate http://svn.example.com/calc/branches/my-branch
--- Merging differences between repository URLs into '.':
```

Please note the use of the *reintegrate* option, which is critical. It tells subversion that your feature branch is a mixture of trunk change sets and branch change sets. With the *reintegrate* you have asked subversion to carefully find the branch change sets and only port these.

Now when your feature branch has been merged into trunk it is recommended to remove the branch because the branch is no longer usable for further work. It would not be capable of correctly pick up change sets from the trunk via an `svn merge`, nor can it be properly reintegrated into trunk again. If you need to work further on the feature branch it is better to destroy it and create a new branch.

If you have been a bad boy and as such, out of sync. You must start by porting all changes made to the trunk into your branch and then you are ready to perform the merge into the trunk.

A.15 Removing a branch

When a branch has become obsolete this can be removed with

```
svn delete http://svn.example.com/calc/branches/my-calc-branch \
  -m "Removing obsolete branch of calc project."
```

A deleted branch can be resurrected if needed.

A.16 Creating a tag I

If we want to create a snapshot of the trunk exactly as it looks like in the HEAD revision, just make a copy of it

```
svn copy http://svn.example.com/calc/trunk \
  http://svn.example.com/calc/tags/release-1.0 \
  -m "Tagging the 1.0 release of the 'calc' project."
```

assuming you followed the guidelines and created a tags directory in you initial import.

Use *-r* if you prefer to specify a revision to tag.

Remember rule 2.1.9; Do not mess with the `tags` directory. If you need to change a *tagged* revision of the project, you need to make a branch using the `tags` structure as template for the branch (see item A.10).

A.17 Creating a tag II

You can create a tag with your current working copy status

```
ls
my-working-copy/
svn copy my-working-copy http://svn.example.com/calc/tags/mytag
```

Why is this here? There is one (at least) interesting use for this feature. Sometimes there are situations where you have a bunch of local changes made to your working copy, and you'd like a collaborator to see them. Instead of running svn diff and sending a patch file (which won't capture tree changes), you can instead use svn copy to "upload" your working copy to a private area of the repository. Your collaborator can then either checkout a verbatim copy of your working copy, or use svn merge to receive your exact changes.

A.18 How to list the set of tags in a project

This is straightforward use of the `svn list` command

```
svn list http://svn.example.com/calc/tags
```

Remember rule 2.1.9; Do not mess with the `tags` directory. If you need to change a *tagged* revision of the project, you need to make a branch using the `tags` structure as template for the branch (see item A.10).

A.19 Help, my favourite repository has moved

Sometimes the location of a repository changes, but the contents of the repository stays the same, *i.e.*, the URL to the repository is changed. A repository move is easily fixed with the `svn switch` command, this will change the URLs in your working copy. No file contents are changed, and your work is untouched and can be committed to the new location after switching to the new URL. This is subversion magic. Here is an example where a repository moved from `svn.example.com` to `new.host.org`:

```
svn switch --relocate http://svn.example.com/calc \
  svn://new.host.org/repos/calc .
```

We changed server type and directory path in this example. The Subversion Book warns about spelling errors when changing the URL since it may cause havoc for you, so be careful out there.

A.20 Change trac ticket status through commit log messages

There may be post-commit hooks in the subversion repository that performs specific tasks when changes are committed to the repository. One of these utility programs performs status changes to trac[10] tickets if the log message is properly formatted. The `trac-post-commit-hook` searches commit messages for text in the form of:

```
command #1
command #1, #2
command #1 & #2
command #1 and #2
```

where `command` is one of

```
closes, fixes
  The specified issue numbers are closed with the contents of this
  commit message being added to it.
references, refs, addresses, re
  The specified issue numbers are left in their current status, but
  the contents of this commit message are added to their notes.
```

`closes` and `fixes` are synonyms, and `references`, `refs`, `addresses`, and `re` are all synonyms. More than one command can be issued in a message. An example message is "Fixes #10 and #12, and refs #13. Changed blah and foo to do this or that." This message will close tickets 10 and 12, and add a note to ticket 13.

Note, it is important that #number is used, *i.e.*, ticket:number will not work.

Information on where to retrieve and install the `trac-post-commit-hook` is available in Appendix E.3.

B Subversion related commands

The subversion family of command all have built in help functionality. Issue `command help` to get general information about the command, and a `command help` <subcommand> will print information on the sub-command.

There are a number of commands associated with subversion

`svn` - Subversion command line client tool

`svnadmin` - Subversion repository administration tool

`svndumpfilter` - Filter a subversion repository 'dumpfile'

`svnlook` - Subversion repository examination tool

[10] http://projects.edgewall.com/trac/

`svnserve` - Server for the 'svn' repository access method

`svnversion` - Produce a compact version number for a working copy

C svn sub-commands

Full information on *svn* and its sub-commands can be read in Chapter 9 of the subversion book. Subversion prints nice help information if you issue `svn help` or `svn help subcommand` at the prompt.

There are many different switches for the sub-commands, some of them are quite general such as these

-r number where *number* is a revision number or revision keyword (*HEAD*, *BASE*, *COMMITTED*, or *PREV*). To compare version within the repository itself use *-r number:anothernumber*. A specific date, or a range, can be used. Put the date inside curly braces (many formats are supported whereof we show one below). You can even refer to date specified revisions mixed with numbered revisions.

  HEAD The latest revision in the repository.

  BASE The "pristine" revision of an item in a working copy.

  COMMITTED The last revision in which an item changed before (or at) BASE.

  PREV The revision just before the last revision in which an item changed. (Technically, COMMITTED-1.)

  DATE example: `svn log -r 2005-01-27:2005-02-21`

-v will give more information in many svn sub-commands.

-N Run non-recursively.

Not all `svn` sub-commands are needed in normal use. Here we list them in some sort of groups.

checkout You can check out any sub-directory you want but are recommended to check out the root level directory. Following the recommendation of the svn book the main trunk is in the repository trunk directory. So, you should do

```
svn checkout http://base.thep.lu.se/svn/trunk base
```

The last base tells svn to checkout the main trunk into a sub-directory base. If you omit base, svn will place the checked out files into sub-directory trunk.

update Update (synchronise) your working copy. If a conflict occurs, svn creates temporary files containing different revisions of the conflicting item. You need to resolve the conflict, and use svn resolved to tell subversion that you resolved the conflict.

add Schedule files, directories, or symbolic links to be added to the repository. This command is recursive, use *-N* to prevent recursiveness.

delete Schedule files, directories, or symbolic links to be removed from the repository. (What happens if the directory is non-empty? Need testing.)

copy Copy items and schedule them for addition into the repository. Inherits history information.

move Move items, *i.e.*, perform `svn copy; svn delete`

status Reports all changes made in the working directory. There are many different status codes, please refer to Chapter 9 in the svn book for the list of codes. Recursive. *-v* switch will output information about every item in the working copy. *-u* will report whether items in the working copy is out of date.

diff Prints file changes in unified diff format. Useful for creating patches, `svn diff` > `patchfile`. With no switches, comparison is done against the pristine working copy, *i.e.*, only locally changed files will show differences. Use *-r* for comparison against the repository.

revert Reverts the item to its previous state, *i.e.*, changes are disregarded and item restored, and any (svn) scheduled tasks are reset.

resolved Tell subversion that you resolved a conflict. Issuing this command will remove the temporary files created by svn update when a conflict is spotted by subversion. There are other ways of resolving conflicts but use this command. Subversion will not accept a commit until conflicts are resolved. Note, you must specify item in the command, and be careful, if you do svn resolved, subversion accepts this without checking that you really resolved the conflict and will subsequently accept a commit without making a fuss.

commit Commit your changes. Tree changes are performed in the repository when you issue commit. Use *-m* "My message." to supply a log message, or if you prefer to write an essay in a file, you can use *--file filename*. If you omit the message, then an external editor is launched. A commit is refused if you try to commit out of date items.

log Shows you broad information: log messages attached to revisions, and which paths changed in each revision. Verbose information available, *-v*. Recursive.

cat This is used to retrieve any file as it existed in a particular revision number and display it on your screen.

list Displays the files in a directory for any given revision.

cleanup Will cleanup subversion if subversion ended up in an undefined state due to unexpected interruption during some subversion command. Locks can be resolved with this command.

import A quick way to copy an unversioned tree of files into a repository, creating intermediate directories as necessary.

D Item properties

You can set properties (meta-data) on items. Properties are arbitrary name/value pairs associated with files and directories in your working copy. For more information see the subversion book.

The special properties supported by subversion are

svn:eol-style Possible values are native, CRLF, LF, CR

svn:executable is used to define whether a file is an executable or not.

svn:externals Read documentation, this is, I think, a wonderful thing. Use this to make subversion automagically checkout stuff from other locations/repositories within your subversioned structure.

svn:ignore is a nice way to add a property to a directory that contains files not under subversion control to avoid cluttering when using `svn status`.

svn:keywords is used to define whether you want subversion to perform keyword substitution. Where the keyword is inserted in your files is controlled by a keyword anchor, $KeywordName$. Supported keywords are

`HeadURL` or `URL`

`Id`

`LastChangedBy` or `Author`

`LastChangedDate` or `Date`.

`LastChangedRevision` or `Revision` or `Rev`

To set the *svn:keywords* property do something like

```
svn propset svn:keywords "LastChangedDate Author" filename
```

svn:mime-type is used to set the mime-type of a file.

To set or get a property name, use the `svn propset` and `svn propget` sub-commands. To list all properties on an object, use `svn proplist`.

Subversion provides some automatic property setting when you do `svn add` or `svn import`, *e.g.* if subversion thinks you are adding a binary file, the `svn:mime-type` is set to `application/octet-stream` (alas the general binary mime type). Subversion does not try to do smarter, *i.e.*, figure out that a file is a png graphics which should have mime type `image/png`). However, you can affect the way subversion sets properties automatically by changing your `config` file. You can make subversion to recognise a pattern like `*.jpg`, and consequently set the mime property to `image/jpeg`. We supply a set of options to add into your `config` file in Appendix F.

E Repository side stuff

Here we provide an account on some actions that can be made on the server side.

- There are ways to (really) remove items from a repository, i,e, all tracking is lost.
- There are straightforward way to split/merge projects on the server side.
- There is a possibility to add hooks on the server side for different subversion commands.
- There are prepared ways to make subversion to send mail of commit information, and to trigger backups at every commit.
- Log messages can be modified. There are even ways to allow users to modify log messages, but these are disabled by default.
- The administrator should look out for outstanding, non-finished, transaction. Dead transactions should be remove, see more in the book (subsection "Repository cleanup")
- Repository migration is possible (read this as CVS to subversion).

E.1 Setting up a repository

There is a file `/path/to/svn/README` on the repository server. This file contains a short description, with links to project web sites where appropriate, about the projects hosted. Please keep this file updated when projects are added or removed.

Initialise a repository (here *base*)

```
svnadmin create /path/to/svn/repository/base
```

Comment: The addition of files and directory structure to the project is done from the client side as usually after the project is checked out, or a directory layout can be forced onto the project (see below).

E.1.1 Repository access

Using a web server with svn support you set up repository access using your web servers authentication methods. On apache webdav migth be used where two files `passwd` and `access` located in `/path/to/svn/` (file name and directory set in subversion/apache configuration) are used to set up authentication.

Using svnserve the repository access is set up by changing to the repository configuration directory

```
cd /path/to/svn/repository/base/conf
```

Create a `passwd` file (you could copy a template from another existing project). Add users into this file, these will all have read/write privileges. `svnserve.conf` must be setup (again, copy a template), where also (optional) anonymous access is set up.

E.2 Administrator enforced repository layout

Follow these steps to enforce the recommended repository layout in new repositories:

```
    mkdir tmpdir
    cd tmpdir
    mkdir project
    mkdir project/trunk
    mkdir project/branches
    mkdir project/tags
    svn import . file:///path/to/repos \
        --message 'Initial repository layout'
Adding project
Adding project/trunk
Adding project/branches
Adding project/tags
Committed revision 1.
    cd ..
    rm -rf tmpdir
```

E.3 Adding a post-commit hook to a repository

As an example of how to add an post-commit hook simply place the `trac-post-commit-hook`[11] script somewhere in the file hierarchy (*e.g.* `/path/to/svn/contrib`). Create a shell script `post-commit` in the `/path/to/svn/reposotpry/hooks` directory that contains the below block of code (there should be a template script to copy in the hooks directory)

```
REPOS="$1"
REV="$2"

LOG=`/usr/bin/svnlook log -r $REV $REPOS`
AUTHOR=`/usr/bin/svnlook author -r $REV $REPOS`
TRAC_ENV='/path/to/tracs/trash/'
TRAC_URL='http://lev.thep.lu.se/trac/trash/'
/usr/bin/python /home/max/svn/contrib/trac-post-commit-hook \
    -p "$TRAC_ENV"  \
    -r "$REV"       \
    -u "$AUTHOR"    \
    -m "$LOG"       \
    -s "$TRAC_URL"
```

Make sure the `post-commit` script is executable.

[11] http://projects.edgewall.com/trac/browser/trunk/contrib/trac-post-commit-hook?format=raw

F Configuration options

When you run `svn` for the first time a new directory is created for you, `$(HOME)/.subversion`. In this directory you find the configuration file, `config`, that is used by the subversion programs. We provide a configuration for you with mandatory options and recommended options. Copy these into your configuration file.

```
[miscellany]
# recommended options
global-ignores = *.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store
# mandatory option
enable-auto-props = yes
# all auto-props setting below are mandatory
[auto-props]
*.bat = svn:eol-style=native;svn:keywords=Id
*.c = svn:eol-style=native;svn:keywords=Id
*.cc = svn:eol-style=native;svn:keywords=Id
*.cpp = svn:eol-style=native;svn:keywords=Id
*.css = svn:eol-style=native;svn:keywords=Id
*.doxygen = svn:eol-style=native;svn:keywords=Id
*.dtd = svn:eol-style=native;svn:keywords=Id
*.h = svn:eol-style=native;svn:keywords=Id
*.hh = svn:eol-style=native;svn:keywords=Id
*.html = svn:eol-style=native;svn:keywords=Id Date
*.java = svn:eol-style=native;svn:keywords=Id
*.jar = svn:mime-type=application/x-java-archive
*.js = svn:eol-style=native;svn:keywords=Id
*.jsp = svn:eol-style=native;svn:keywords=Id
*.jpg = svn:mime-type=image/jpeg
*.m4 = svn:eol-style=native;svn:keywords=Id
*.pl = svn:eol-style=native;svn:executable;svn:keywords=Id
*.pm = svn:eol-style=native;svn:keywords=Id
*.png = svn:mime-type=image/png
*.R = svn:eol-style=native;svn:keywords=Id
*.sh = svn:eol-style=native;svn:executable;svn:keywords=Id
*.tex = svn:eol-style=native;svn:keywords=Id
*.txt = svn:eol-style=native;svn:keywords=Id
*.xml = svn:eol-style=native;svn:keywords=Id
*.xsd = svn:eol-style=native;svn:keywords=Id
*.xsl = svn:eol-style=native;svn:keywords=Id
AUTHORS = svn:eol-style=native;svn:keywords=Id
bootstrap = svn:eol-style=native;svn:executable;svn:keywords=Id
ChangeLog = svn:eol-style=native;svn:keywords=Id
configure.ac = svn:eol-style=native;svn:keywords=Id
INSTALL = svn:eol-style=native;svn:keywords=Id
Makefile = svn:eol-style=native;svn:keywords=Id
```

```
Makefile.am = svn:eol-style=native;svn:keywords=Id
NEWS = svn:eol-style=native;svn:keywords=Id
README = svn:eol-style=native;svn:keywords=Id
references.bib = svn:eol-style=native;svn:keywords=Id
TODO = svn:eol-style=native;svn:keywords=Id
```

The `global-ignores` define file patterns that is to be ignored when a `svn status` is issued. The `auto-props` patterns are only used when you add files and directories, *i.e.*, using `svn add` or `svn import`.

We also supply four bash shell aliases that make status checks of your working copy easier, add them to your `.bashrc`

```
alias svnl="svn log -r 1:HEAD -v"    # Show log messages
alias svnm="svn status -u -q"        # Show locally/repository changed items
alias svnn='svn status -q'           # Show locally changed items
alias svns="svn status -u -q -v"     # Show status of all items
```