Distributed Software Development using Subversion and SubMaster

Clifford Wolf, http://www.clifford.at/

August 11, 2004

1 Infrastructure for the bazaar

Eric Steven Raymond published his *The Cathedral and the Bazaar* paper in the mid 90s and started a paradigm shift in the open source community with that document. Even companies such as Netscape changed their philosophy on software development and tried to change their development model to a more *bazaar like* one.

But such a bazaar is hard to handle and much more complicated than any cathedral style development philosophy. The fact that most development tools and revision control systems are not designed with the bazaar philosophy in mind does not make it any easier.

Subversion (SVN) is a great revision control system without any of the design flaws of e.g. CVS. However, like most other revision control systems, it is not ready for the bazaar.

SubMaster is a proof-of-concept implementation of a system for bazaar-like distributed software development I wrote for to use in the ROCK Linux [2] project (where I am the project leader). The ROCK Linux developers decided to switch to subversion in mid 2003 and are using SubMaster since spring 2004.

SubMaster itself is a small set of scripts to make it as easy as possible for developers to create their own local subversion branches, keep them in sync with the main repositories and send patches upstream. We from ROCK Linux believe that SubMaster can be used as adequate replacement for BitKeeper in many projects and has some important features BitKeeper is missing.

1.1 Revision Control Systems

A Revision Control System manages multiple revisions of files or entire projects. It automates the storing, retrieval, logging, identification, and merging of revisions. As a side-effect it also often allows more than one person to work on the same project at the same time, make their own changes and commit them to some kind of central repository.

In this paper I am not going to take a deeper look into the primary function of a revision control system (to manage revisions and the history of a project) but focus on the features related to collaborative development.

An overview of revision control systems can be found at Wikipedia [11].

1.2 Centralized, Decentralized and Distributed Software Development

Most software projects - from the open source world as well as from the closed source world - follow a centralized software development model. Even those projects which are bazaar-like managed.

Centralized software development in the context of this paper is a development model where the project source is stored in one central repository and only a limited number of people have write access to that repository. In the most extreme case the project is not stored in any special kind of repository at all but just lingers around somewhere in the project maintainers home directory as ordinary files. This is not necessarily a bad development model in general and for many small projects everything else would be overkill.

Tools for centralized software development are e.g. CVS or Subversion.

As already indicated, even a project with a centralized development model can be bazaar-like managed. People can still send patches per email and the project maintainer can apply them. However - with a centralized development model there is always something like a *core team* who has write-access to the source tree and everyone else has to send patches per mail. There are other development models which do scale much better and are more bazaar-friendly.

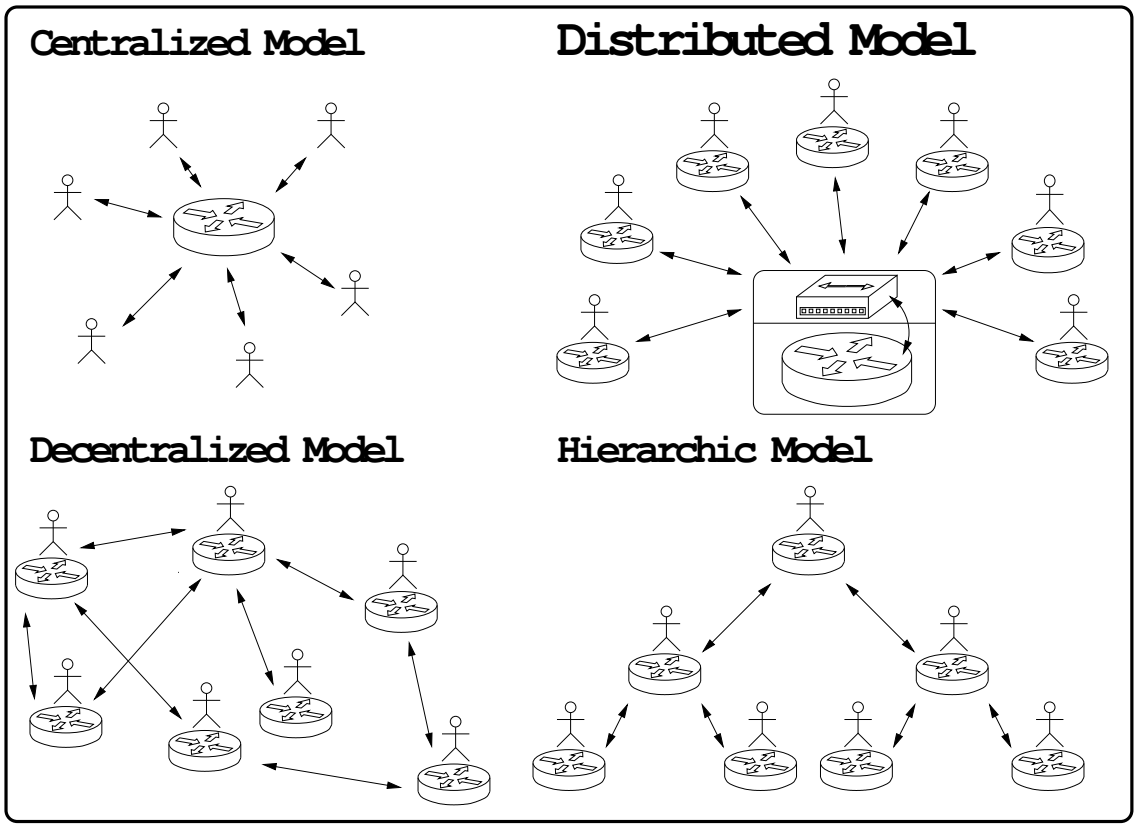

Figure 1: Software Development Models

One of this other development models is the decentralized development model. This development model simply does not have a central repository. Instead, everyone has his own local repository and commits changes to this local repository. There also exists some infrastructure for exchanging local changes with other people so they can import those changes to their trees.

In theory this should result in a true bazaar-like system where everyone is exchanging his modifications with everyone else. But in reality this usually results in some kind of hierarchy with the project maintainer on the top. Patches *hike* this hierarchy upwards until they reach the project maintainers tree from where they are then distributed to everyone in the project (usually everyone is importing all changes from the project maintainers tree). The *private tree* of the project maintainer thus also becomes the *official source tree* and the whole thing is not decentralized anymore.

Tools for decentralized software development are e.g. BitKeeper or SVK.

SubMaster adds a new concept which I am calling *distributed software development* in this paper.

In decentralized software development there is nothing like a *center node* or primary repository by design. But as I stated already, one of the repositories - the repository of the project leader - will become such a center node.

In the real world, the most important feature of decentralized software development tools is not the lack of a center node. It is the way changes go from the author to the center node.

With centralized tools every change happens directly in the 'official' repository. If someone is working on a bigger project it is necessary to create a branch which involves a person who has the permissions to create branches, etc.

With decentralized tools everyone can do such stuff locally in his own repository, and as soon as the work on

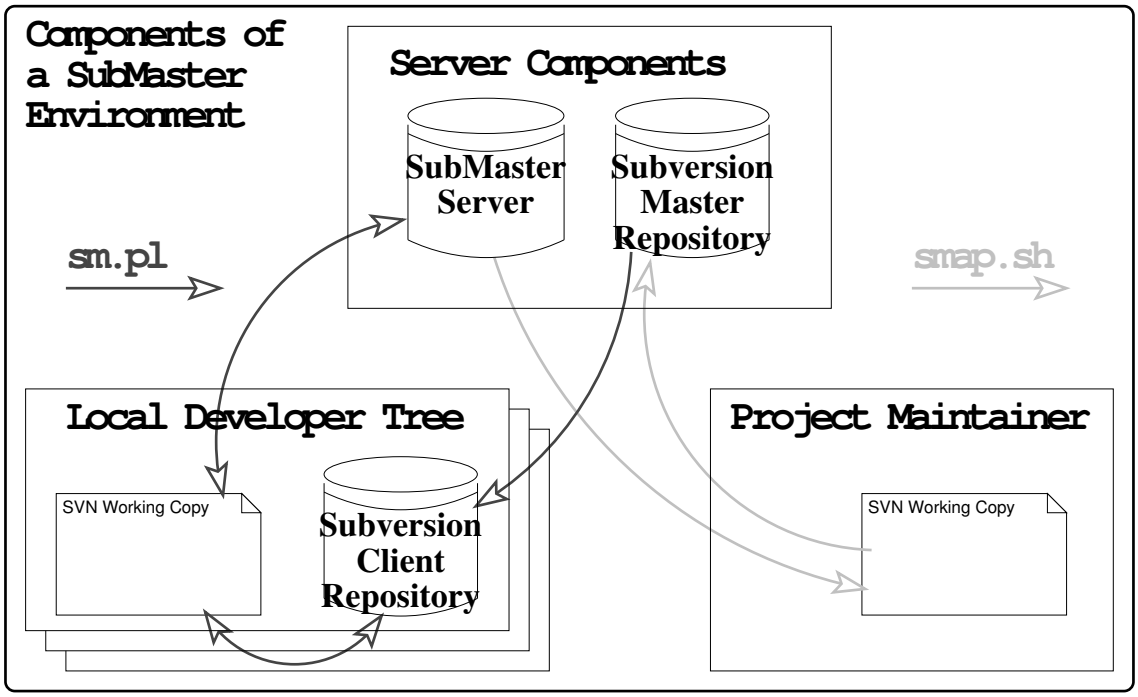

Figure 2: Components of a SubMaster Setup

a feature is finished, the change can be sent upstream, maybe discussed on mailing lists, etc. and eventually get applied by the maintainer of the center node.

Sure - that can also be done by e.g. working in a local CVS checkout and then send the patch created by `cvs diff` to a project mailing list. But in this case there is not much support by the tool chain. E.g. if one wants to create a 2nd patch based on the first one (before the first one got applied), a simple `cvs diff` is not sufficient anymore for creating the patch.

With SubMaster and the distributed model I have tried to create a tool chain for collaborative development which supports exactly that kind of work flow which seems to be the 'natural' way of doing development in mid- to large-scale projects:

1. People develop features locally, independent from the rest of the world and without fiddling around in any kind of 'official' repository.

2. When the change seams to be ready, it is somehow sent upstream. This is usually done by sending it to a mailing list or a single person or by 'pushing' it to somebody else's repository.

3. The change is tested and discussed by others and eventually applied or rejected in the official source tree in some kind of central repository.

4. The change is finally distributed from the central repository to everyone else's local repositories.

Note that this simple process does not provide a way for anybody to commit changes directly to the central repository by bypassing step 3. While I believe that this will always be possible in one way or another and will always have its uses, I also believe that this is only useful in some very rare cases.

Each of this steps has its own issues which should be targeted by the tool chain:

Ad 1. Working on many changes simultaneously or one after the other should be as easy as possible. E.g. a common problem with the `cvs diff` approach mentioned above is that it is hard to have a multiple distinguishable set of local changes which can be sent upstream or otherwise worked independent with it. Also there should be some kind of local revision management for the changes so that the changes can be seen as a little project on its own where it might be useful to e.g. rollback a change or manage different local branches for different approaches to solve a problem.

Ad 2. Sending patches upstream must be as easy as possible. It is a common problem that one-line fixes are not sent upstream because sending them is much more work than writing the patch. In a good tool chain for distributed software development it must be as easy to send patches upstream as to commit a change in a traditional revision control system.

Ad 3. A common pitfall is that changes get lost at this stage - especially if a mailing list is used to manage the pending changes; so there must be some kind of queue for the pending changes where patches must be explicitly applied or rejected. Additionally a system for collecting and summarizing feedback is needed.

Ad 4. It is possible that merge-conflicts with local changes happen at this stage. So a concept for resolving such conflicts needs to be provided.

2 A short Introduction to Subversion

The Subversion Handbook [4] gives a pretty good introduction to Subversion. So I am providing the reader only with the most important essentials in this document:

The tool 'svnadmin' can be used to create and administrate repositories.

The tool 'svn' can be used to check out files from the repository, create diffs and commit changes back to the repository. Using 'svn' is very similar to using 'cvs'.

There are three ways of accessing a Subversion Repository:

1. Using the local file system (file://)
2. Using the WebDAV extension to HTTP (http://)
3. Using Subversion's own protocol (svn://)

When using WebDAV it is even possible to work *directly on the repository* without using a local working copy (if the operating system or desktop environment supports WebDAV, even without a special Subversion client).

All the file systems structure and data live in a set of tables within the db subdirectory of the repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of Berkeley's database tools.

Subversion has no explicit support for tagging or branching. Instead it has support for copying directory trees in constant time without really copying all the data in the tree (instead, just the meta information that tree A is copied to tree B as of revision N is added to the repository). So, instead of *tagging* a version or *branching* a project, the source tree is just *copied*. Changes in one tree can always be migrated to another one using the `svn merge` command.

2.1 Creating branches with Subversion

The basic idea in Subversion's model of doing distributed development is to create branches for every developer in one central repository. And merge the changes from the branch back to trunk every now and then.

It's also possible to regularly merge changes in the trunk to branches, e.g. if the developer likes to keep track of ongoing development in trunk.

So creating a branch is pretty easy:

```
~$ svn copy -m "Branching Project 1 for Alice" \
svn://svn.example.com/prj1/trunk \
svn://svn.example.com/prj1/branches/alice
Committed revision 341.
```

Now Alice can check out her branch and work with it as usual:

```
~$ svn co svn://svn.example.com/prj1/branches/alice \
prj1-branch
```

Maybe she wants to synchronize her branch with the official tree some time later:

```
~/prj1-branch$ svn merge -r 341:HEAD \
svn://svn.example.com/prj1/trunk
~/prj1-branch$ svn commit
Committed revision 417.
```

Subversion is smart enough to ignore changes the user is trying to merge to a tree which obviously happened already there, but this does (naturally) only work until a particular level of complexity is reached.

So, Alice always needs to keep track of the revision number she merged or branched so she can always merge the right set of changes back to trunk or in her tree respectively.

Also, subversion - on its own - does not support having the branch in a different repository. So every developer needs a branch in the main repository and needs to have write permissions in his branch.

3 The SubMaster Client (sm.pl)

While Subversion is a pretty big and complex piece of software, SubMaster is a set of comparable trivial scripts which provide developers with tools for distributed development - using subversion as back end.

The installation is easy: Type `make install` in the package source directory to install the scripts in `/usr/local/bin` (with the `.pl` and `.sh` extensions stripped of).

The SubMaster client (sm.pl) is basically a helper script to create a local copy of a (read-only) remote subversion repository and enable the user to make local changes, converting the local changes to unified diffs and keeping the local repository in sync with the remote repository.

In theory, this could also be done without `sm` directly with `svn`, but `sm` makes the task easy enough to be used in the real world.

The `sm` script is also capable of uploading the generated patches to a SubMaster server. But it is also possible to use `sm` stand-alone on the client and e.g. send the patches to the usual mailing lists. This at least reduces the usual pain with creating patches.

3.1 Creating a SubMaster Working Copy

The local subversion repository and its additional meta data files is called a *SubMaster Working Copy* in this document. Creating such an SM Working Copy is rather easy. E.g.

```
$ sm create svn://hostname/.../trunk rock
```

This creates two new directories: `rock` and `rock.sm`. The `rock` directory holds a subversion checkout of the local repository and the `rock.sm` directory all the meta data including the local subversion database and the created patches.

The local subversion repository has two directories in his file system root. A `MASTER/` directory containing a local copy of the remote repository and a `WORK/` directory where the local changes go to. The `rock` UNIX file system directory in the example above is a subversion checkout of the `WORK/` directory tree.

Whenever changes from the master are synced to the local tree, they are first synced to the `MASTER/` directory and then merged to the `WORK/` directory.

3.2 Making local changes to a SubMaster working tree and creating patches

Working with the local SM Working Copy is as easy as working with an ordinary Subversion working copy. The only difference is that the command `sm` has to be used instead of `svn`. Most commands such as `sm add` or `sm stat` are just passed directly to `svn`.

But the command `sm commit` is special. It first commits the changes to the local subversion repository and adds the newly created revision number to the *change queue* afterwards. All revisions in this *change queue* are waiting to be converted to patches and then eventually being sent upstream. `sm queue` prints the current status of this queue:

```
$ sm queue
r44 | root | 2004-01-10 12:10:58 | 3 lines
Added package oprofile.
------------------------------------------------
r51 | root | 2004-01-10 13:34:35 | 3 lines
Oprofile: don't run depmod.
------------------------------------------------
```

The revisions 44 and 51 can now be easily converted to a patch by running `sm patch 44 51`. This will merge both local revisions into one patch file! So there is no need to make only one commit per patch. It is possible to make a few changes, commit and test them, commit a few fixes and then create one patch file for the entire issue.

`sm patch` also removes the revisions from the patch queue.

If a commit should be converted to a patch directly, `sm instant` can be used instead of `sm commit` to create the patch file bypassing the change queue.

`sm patch` and `sm instant` are using the `svn diff` command and the patchutils package to create the patch files.

3.3 Merging upstream changes to the local SubMaster tree

Whenever a change happens in the master repository, every developer is likely to want this change in his local repository too. If both trees would be in the same repository, this would simply be done with `svn merge`. But usually this is not possible between different repositories.

File	Description
demo/	Local SVN Working Copy
demo.sm/*.patch	Local generated patches, waiting to be sent upstream
demo.sm/SVN/	Local Subversion Database
demo.sm/SENT/	Patches that have been sent upstream
demo.sm/WIP/	The Work-In-Progress Archive
demo.sm/MASTER/	A local copy of the master repository
demo.sm/SM/master.txt	Subversion URL of master repository
demo.sm/mrev.txt	Revision mirrored in demo.sm/MASTER/
demo.sm/queue.txt	The queue managed by sm queue
demo.sm/server.txt	URL, username and password for submaster server
demo.sm/sync.txt	Last master revision we synced with
demo.sm/version.txt	SM On-disk format version (2)

Figure 3: SubMaster Client On-Disk Files

I wrote a simple patch for subversion which makes this possible. Because of its side-effects the switch is called --duplicate .

sm sync does nothing else than run svn merge --duplicate -r<LAST-SYNCED-REV>:<CURRENT-HEAD> <MASTER-URL> . in the local working copy and updates a local meta data file which stores the <LAST-SYNCED-REV> part.

Finally the user is asked to call svn commit after resolving the conflicts (if any). Note that it does not result into a conflict if a local generated change got applied upstream. Subversion is smart enough to handle that on its own.

A fitting svn commit command line is printed by sm sync and can be copy&pasted to the shell command prompt.

3.4 Testing and extending other peoples changes

Somebody else's patch can simply be applied locally by running smap <PATH-ID> (the <PATH-ID> is a unique ID which is assigned by the SubMaster server). This downloads the patch, applies and commits it locally.

This way it is easily possible to create new patches based on the patch written by somebody else, even before the other patch gets applied in the master repository. The new patches should always be marked on the SubMaster server to depend on the other patch to make the maintainers life easier.

(A more detailed description of smap and the related tool fixdiff follow in a separate section in this paper.)

3.5 Full-syncs and diffs against master

In addition to sm sync which merges local changes with changes made upstream, sm fsync can be used to get rid of all local changes and make sure that the local repository has exactly the same data as the master repository.

The command sm xdiff lists the differences between the local tree and the master repository (sm diff just calls svn diff which displays the local changes that have not been committed to the local repository so far).

3.6 Accessing the SubMaster server

The command sm admin is a shortcut for calling w3m with the URL of the configured SubMaster Server and passing user name and password to it.

But it is also possible to open the SubMaster web interface in your favorite web browser..

3.7 Required Patches to Subversion

As noted already above, Subversion on its own is not able to merge changes from one repository to another. Since this is absolutely required for SubMaster's job, we added that functionality to Subversion.

The patch is pretty simple and adds a command line switch which basically does two things:

File	Description
smadm.cgi	The CGI script
action.sh	The main action script
password/	The encrypted passwords (*Username*.txt)
	This directory should not be accessable thru HTTP!
user/	Short user descriptsions (*Username*.desc) and the email adresses (*Username*.email)
data/*YYYY/MM/ID*.patch	The patch file itself with a description in the header
data/*YYYY/MM/ID*.owner	The user who uploaded the patch
data/*YYYY/MM/ID*.votes	Pro and contra votes from various users
data/*YYYY/MM/ID*.msg	Additional comments written by various users
data/*YYYY/MM/ID*.info	Additional comments generated from action scripts
data/*YYYY/MM/ID*.done	The patch-status for non-open patches
open/*YYYY_MM_ID*.open	An empty file for every open patch

Figure 4: SubMaster Server On-Disk Files

1. For merges, it disables the check if source and target are in the same repository.

2. It deactivates the mechanism which usually creates links to files created in the source tree when merging this file creations to the target tree. That is why the switch is named --duplicate.

4 SubMaster Server

The SubMaster Server is a rather simple CGI script written in Perl. It features a simple user-database and allows users to upload their own and comment other peoples patches.

Installation of the SubMaster Server is easy - just copy the scripts, create a few directories, add an admin user and give the apache user write access:

```
# cp .../smadm.cgi .
# cp .../votecheck.pl .
# mkdir -p data open password user
# perl -le "print crypt('secret', 'XY')" \
> password/admin.txt
# touch password/admin.super
# chown -R apache. data open password user
```

5 On-Disk Files

5.1 SubMaster Client On-Disk Files

Figure 3 has a list of files created by sm create ... demo.

Note that some information which might also be stored in Subversion properties is stored in small files in demo/SM/ here. The reason for that is that this way the information can be accessed much easier (and faster!) by the scripts.

5.2 SubMaster Server On-Disk Files

Figure 4 has a list of files in a SubMaster Server setup.

We have decided not to use any SQL Database as back end for the script so the action scripts (see next section in this paper) can access all data as easy as possible from a wide range of programming languages (including UNIX shell scripts).

Such a SubMaster Server setup easily grows to an accumulation of a few thousand small files. So a file system with support for tail-merging and indexed directories might be a good choice to save disk space.

6 SubMaster Action Scripts

Sometimes it is desired to automatically react to certain actions performed on the SubMaster Server, for example by sending notification emails or running some checks over newly submitted patches. This can be easily implemented using so-called action scripts on the SubMaster server.

Whenever a patches entry is changed on the SubMaster server, a user-supplied script (action.sh) is started and the type of modification is passed as command line arguments such as:

```
    <...>
01  list=rock-sm@rocklinux.org
02  me=submaster@rocklinux.org
03  unique="$(date "+%s.$$")"
04  {
05    echo "From: $me"
06    echo "To: $list"
07    if [ "$cmd" = new ]; then
08      echo "Message-ID: <${id//\//}.$me>"
09    else
10      echo "Message-ID: <${id//\//}.$unique.$me>"
11      echo "References: <${id//\//}.$me>"
12    fi
13    echo "Subject: [${id//\//}] $user $cmd ${opt:0:40}"; echo
14    ./smadm.cgi | grep -A99 ^Content-type | tail +2 | \
15      w3m -T text/html -cols 77 -dump 2> /dev/null | \
16      perl -pe 's/\s+$/\n/; s/ {$x}/ / if ($x=length($_)-75) > 0;'
17    echo "$url?i=${id//\//}"
18  } | sendmail -f "$me" "$list"
    <...>
```

Figure 5: Example action script for status mails

./action.sh id user msg 'Message Text'

A full list of possible arguments is contained in the example action.sh which comes with the SubMaster package. This example script also contains code to set the shell variables $id, $user, $cmd and $opt based on its arguments.

In parameters such as the message text in the example above, all characters matching [<>'" \n\r\t] are URL encoded.

6.1 Example: Sending informal status messages to a mailing list

The example code-snippet in figure 5 is a strip-down of the code for sending status mails in the example action.sh contained in the SubMaster package. The code used in the example action.sh also does stuff like sending e-mails to the patch owners, using the users e-mail as from address, etc.

Lines 1 and 2 simply set $list and $me to the To- and From- addresses for the e-mails. Lines 5-17 generate the mail header and body which are piped to sendmail in line 18.

If the action is generating a new patch (line 7), open a new task by only setting the Message-ID header (line 8), otherwise there must already have been a message for the patch we can refer to (lines 10 and 11).

The subject line contains only the most important information (line 13).

The message body contains the current status of the patch with all votes, messages, etc. attached. The example script sets variables such as $QUERY_STRING so that calling ./smadm.cgi does generate the HTML output for the patch details (line 14).

W3m and a Perl one-liner (lines 15 and 16) is used to convert the HTML text to nice ASCII. Finally the URL for going directly to this patch in the SubMaster Web interface is appended (line 17).

6.2 Example: Checking for votes

In ROCK Linux there are pretty clear responsibilities as to who maintains which part of the system. If anyone is sending a patch for a module, it is important for the tree maintainer to have the feedback from the maintainer of that module before applying the patch.

So we added the Perl script votecheck.pl which is called from action.sh whenever a new patch is created or the votes for the patch have been modified. The script is also included in the SubMaster package for reference.

votecheck.pl generates a list of files touched by the

Option	Description
-pN -l -R	pass this option to the patch program
-d	just make a dry-run
-S	do not commit to svn (e.g. if this is not a svn tree)
-E	edit commit message before running svn commit
-D	do not remove temp files, even after success
-a filename	add this smap command to journal file after success
-O filename	merge diff with this file (create combined diff)
-P dirname	copy all applied patches to this directory
-A	do not apply anything, just do -a, -O and/or -P
-M	mark patch as applied on SM Server
-C	do nothing if already marked as applied
-g	just get the patch, do nothing else
+?	revert the effect of a prior specified -? option.

Figure 6: SMAP Parameter List

patch and compares that with its current votes and a hardcoded table of file patterns and maintainer-names. If a maintainers vote is missing, the script adds a small red remark to the patch description saying that the maintainers vote is still missing. If a maintainer voted against a patch, the script adds a fat red warning to the patch description saying that the maintainer voted against it.

7 Applying patches

7.1 The smap script

The tool smap can be used to apply patches in SubMaster working copies, in ordinary subversion working copies and in unmanaged file system trees. The default behavior of smap can be changed using the command line options shown in figure 6.

If smap is running in a SubMaster working copy, it reads its configuration from the SubMaster configuration files. In all other cases it is configured by a file smap.cfg in the current directory:

```
set -- -M -C "$@"
URL=https://www.rocklinux.net/submaster
USER=username
PASS=password
```

The first line (set -- -M -C "$@") overwrites the default settings for smap. This example is useful for managing the master subversion repository: Patches are ignored if they are already marked as applied and new patches which are smaped are automatically marked as applied on the SubMaster Server.

When called with -a (this option is usually passed in the configuration file), smap writes a journal with the smap commands and (as comments) the patch descriptions. This journals can be executed as shell scripts somewhere else to apply the same set of patches. This way it is possible to e.g. apply a set of patches and test them intensively in one location, and then send them to the project maintainer to be applied in the master repository. Those journals can also be pasted in the SubMaster server Web frontend to add them to a patch selection (e.g. if the maintainer wants to check for new comments before applying the journal).

7.2 The fixdiff script

Sometime a patch gets "out of sync" with the master repository and does not apply anymore. For those cases we added the fixdiff script to the SubMaster package.

E.g. if `smap 20040405094233279l3` fails because the part of the patch for `scripts/functions` failed, `fixdiff ed 20040405094233279l3.patch scripts/functions` opens `scripts/functions` and the `*.rej` file in `vim` in a split-screen and lets the user apply that part manually. `fixdiff` then creates a file `fixdiff _20040405094233279l3.patch` which is a "patch for the patch".

All changes to the patched files (`scripts/functions` in our example) is undone by fixdiff. The script only creates the `fixdiff _*` file and updates the `*.patch` file, but does not actually apply the patch.

`smap` is always looking for `fixdiff _*` files in the current working directory before applying patches.

8 Other Subversion Projects

8.1 Transvn - a 'patch-scripts' package based on Subversion

Transvn [8] is like Andrew Morton's patch-scripts package, but implemented using Subversion as back end.

Transvn enables the user to manage a "stack of patches" for a third party project. It is meant for maintaining a huge number of complex third party patches for source code projects and helps with keeping in sync with changes in the project the patches are for and handling of events such as patches being applied upstream.

8.2 Svk - a decentralized system

Svk [9] is a decentralized version control system written in Perl using the Subversion file system as back end. It features distributed branches, advanced merge features and changeset signing.

In Svk there is no central master repository as in centralized systems such as CVS and SVN or distributed systems such as SubMaster. Instead changes are exchanged directly between the local repositories.

Some people believe that this is a good idea in common and in fact it depends a lot on the project size and nature if plain Subversion, Svk or SubMaster is a better choice.

8.3 Svm - mirroring subversion repositories

Svm [10] is part of the SVN::Mirror CPAN package which can be used to mirror changes from one subversion repository to another. Svk is using Svm for keeping the repositories in sync and there are plans to integrate it in SubMaster too.

9 References

[1] Subversion Homepage
http://subversion.tigris.org/

[2] ROCK Linux Homepage
http://www.rocklinux.org/

[3] SubMaster Homepage
http://www.rocklinux.org/submaster.html

[4] The Subversion Handbook
http://svnbook.red-bean.com/

[5] Perl
http://www.perl.org/

[6] Clifford Wolf's Homepage
http://www.clifford.at/

[7] LINBIT Information Technologies
http://www.linbit.com/

[8] TranSVN
http://alexm.here.ru/transvn/

[9] Svk
http://svk.elixus.org/

[10] Svm, SVN::Mirror
http://search.cpan.org/ clkao/SVN-Mirror/

[11] Wikipedia, Revision Control
http://en.wikipedia.org/wiki/Revision_control