DVCS or a new way to use Version Control Systems for FreeBSD

Ollivier ROBERT
<roberto@FreeBSD.org>

11th May 2006

Abstract

FreeBSD, like many open source projects, uses CVS as its main version control system (VCS), which is an extended history of all modifications made since the beginning of the project in 1993. CVS is a cornerstone of FreeBSD in two ways: not only does it record the history of the project, but it is a fundamental tool for coordinating the development of the FreeBSD operating system.

CVS is built around the concept of centralised repository, which has a number of limitations.

Recently, a new type of VCS has arisen: Distributed VCS, one of the first being BK from BitMover, Inc. Better known from the controversy it generated when Linus Torvalds started using it, it has nonetheless changed the way some people develop software.

This paper explores the area of distributed VCS. We analyse two of them (Arch in its Bazaar[1] incarnation and Mercurial[2] and try to show how such a tool could help further FreeBSD development, both as a tool and as a new development process.

1 Introduction

FreeBSD[3] has been using CVS as its main version control system (VCS) tool for as long as it exists and has now more than 10 years of history. A few years ago, limitations inherent in CVS design became too much to workaround and the project begun using Perforce[4] for projects that needed to change fast without "polluting" the main tree.

It works well but using two VCS instead of one is making merges harder than it needs to be and CVS limitations have become too much even for the main tree itself. The separation of the repository into 4 different ones helped but it is still complicated.

After a bit of history, we will explore how we could solve this problem.

2 A brief history of VCS

2.1 Ancestors

At the beginning, the main tools used to manage different versions of software were pretty primitive but it was enough for most people during early stages of development and large pieces of software were developed using SCCS or later, RCS.

Both SCCS & RCS uses the same basic principles to handle changes with a special directory in the working area, storing the files and the diferents revisions in a special format with a fundamental difference between the two: SCCS uses a special format called "weave" (see [5]) whereas RCS stores separate deltas: always storing the latest revision and storing differences down to the first one.

RCS assumes you will want a fast access to revisions close to what we will call the *HEAD*, the latest checked-in revision in the main line of development, the main inconvenience being that as you add branches and tags, the backend storage file gets more complicated and the system will gradually slow down.

AT&T and CSRG at Berkeley both used SCCS to manage whole versions of UNIX^{TM} and one can find in many files the marker for SCCS[1]. The `what(1)` command is used to "reveal" SCCS strings in binaries. The author even used to embed the SCCS marker inside RCS `$Id$` strings to be able to use either `what(1)` or `ident(1)` from RCS on such binaries.

Both SCCS and RCS use a "locking" model where checkout means locking a file before being able to modify it. This model essentially works because it assumes that a given file will be modifed by only one person at some point in time. It is pretty easy to see that it doesn't scale especially with teams in different locations, as neither of them support remote operations. That means that sharing a tree can only be done using NFS or an equivalent sharing file system.

What is actually interesting is that among the currently available VCS (distributed or not), most of them use SCCS or RCS as the base for its design, either by copying the User Interface (UI) – SVN for example – or by extending it in different ways:

- Perforce uses the RCS file format with DB files for metadata
- BK is more or less a rewrite of SCCS to allow cloning of repositories/branches (See the announce posted on the `linux-kernel` list [6]).

There is another area where these venerable VCS don't work efficiently: binary file support. They are fundamentaly designed to cope with text files such as source code; binary files would be stored as "text" with all the potential loss of information and any command that tries to display or merge would generate gibberish on the screen.

2.2 The *Golden Age* of CVS

In 1986, Dick Grune created CVS with two of his students as a set of shell scripts over RCS in order to be able to work on pieces of a compiler independently[7]. His work was then rewritten in C by Brian Berliner and a paper published in 1990 at USENIX Winter Technical Conference[8].

At first, CVS didn't have remote operations and thus, to work on a given CVS tree, you would have to be logged on a machine with "physical" access to the tree (of course it could be through NFS[2]).

The main advantages of CVS apart from being free – a very useful feature in itself of course – at that time were:

- A central repository instead of a collection of small trees each with its own unsharable `RCS` directory
- A "no-locking" mode of operation, allowing concurrent access to the repository through extraction (also known as *checkout*) of a subset of the tree in *sandboxes* – a private workarea from which each developer can commit his work regularely and merge his/her modifications with others.
- A central repository enables the use of custom scripts (for sending commit logs by mail), access control lists and triggers.

For all these reasons and the fact that none others existed, CVS became the VCS of choice for many projects, both proprietary[3] and free ones like all the BSDs[4].

Soon, most if not all FOSS[5] began using CVS with the notable exception of Linux, as the Linus Torvalds disliked CVS so much that he refused to use an imperfect product[6].

Then CVS gained remote operation support (through either RSH, SSH or its own pserver mode) and its adoption by SourceForge[9] and similar *projects repositories* really pushed CVS in

[1]The marker is `@(#)` and the RCS equivalent is `$Header$`
[2]NFS: Network File System
[3]The author knows for a fact that the French telco company Alcatel has built its own VCS on top of CVS.
[4]All 4 of them, not counting TrustedBSD: FreeBSD, NetBSD, OpenBSD and more recently DragonflyBSD
[5]FOSS: Free and Open Source Software
[6]How he managed to not use a VCS for so long (1991 - 2000) keeps on baffling the author of the paper. . .

the open. The various bits of documentation available first on FTP sites then on the WWW, the books (see [10], [11] and [12]) and the simple but effective UI of CVS also helped a lot to lower the difficulty of using a VCS and thousands of people began using it.

2.3 CVS flaws

Nowadays, many developers confess to stumbling on one or several misfeatures or design problems with CVS. Its flaws are now very well-known:

- Commits are not atomic (i.e. there is no concept of a changeset[7]), the granularity being the directory: in a single directory, you get consistency between the various impacted files through a lock but if a given commit spans multiple directories, you are on your own: that's why the various convertion tools like Tailor[13] or `cscvs`[8] have some difficulties finding all files in a given changeset.
- You can not rename files and directories. The only way to achieve that is either by *delete+add* – which is very wrong and loses history – or manually edit the repository to copy or move the corresponding backend file.
- CVS has no fine grained access control facilities and needs to rely on filesystem permissions for many things although heavy users such as the FreeBSD project have developed customs wrappers and scripts to add per-branch ACL, review workflow and more.
- Directories are not versioned meaning that permissions and ownership is not preserved or kept but also that you can not have an empty directory. That's not a major inconvenient and people have been living without it for a long time but it is desirable.
- Branches are cumbersome to use, especially when you want to merge bits of this and bits of that from another branch. As the main entity versioned is the file and CVS has no memory of branching, you have to keep somewhere the exact revision for each file to be merged or use tags and/or dates to get this information.
- Third-party code integration and maintainance is very cumbersome[9] as it is done through a special branch called the *vendor branch*.

The last two points are critical for projects such as FreeBSD because we have to maintain parallel branches for STABLE/CURRENT developement and releases (and security branches). The weak support for branches is also something that slows down development in general. It makes working on specific sub-projects or features more complicated. That's why a few years ago, FreeBSD made the choice of using a different VCS for such cases: Perforce (see below).

Offline work is possible through *CVSup* (see below) but it is still very limited.

2.4 Enter Subversion

To remove all these limitations, SVN was born.

SVN is often cited as the natural successor to CVS and looking at it, it is easy to see why:

- It is written by former CVS developers
- It is touted as *CVS done right*
- Resembling CVS as much as possible is one of its primary goals (you can alias the `cvs` command to `svn` and get running in no time – ignoring the differences of course)

It has the same properties we all come to like in a centralised VCS with many CVS flaws corrected: atomic commits, easy and fast branching and tagging[10], triggers, ACLs and all that. It has also its own book [14] and the Pragmatic Programmers have written one too [15].

[7]See `http://en.wikipedia.org/wiki/Atomic_commit`

[8]A cvs-to-arch conversion tool. See `http://wiki.gnuarch.org/cscvs`

[9]The author is also maintainer for the ntp codebase integration in FreeBSD and suffers everytime it is time to upgrade…

[10]It is even the same mechanism here, a tag is just a one time branch

You can even have a DVCS on top of that through `svk`[16].

So what are the problems with SVN?

If we ignore the centralised part (we will look at these aspects in section 2.5), SVN has still some important shortcomings (although it gets better over time):

1. It is a big program, with some large dependencies (like Apache2 to get a web interface), `apr` & `apr-util` (both part of Apache2). Apache2 is *not* mandatory though, there are other ways to access a given repository (`svnserve` and `SSH` are available) and SVN ships with its own version of libraries.

2. It used to require Berkeley DB as storage backend for everything (strings contained in the checked-in files, and so on) and experience has shown us that BDB usage in SVN is not stable enough (and in the early days, the DB schema was changing for almost every release which was painful). Current versions use FSFS as default storage method.

3. The way branches and tags are implemented, replication of a given repository would generate copies of entire files on the client (see below for an explanation)...

4. It is snapshot based as opposed to changeset-based (See [17] for a very nice description of both types of VCS) which scale less than the latter ones.

5. In addition to the previous point, it has no memory of what has been merged overtime (like CVS).

While both the first and second ones make integration with FreeBSD rather difficult and the third one surprises me (but is explained in [18]), the last point is the worst one. It complicates merging between branches and makes managing third party code (found in for example `src/contrib` in FreeBSD) more difficult.

SVK[16] could be a way to work around the centralised design of SVN but as it does not change the first and fourth problems and adds a Perl layer to SVN, making integration even more difficult although there are good things coming fromSVK: it is faster and uses less disk space for the working copy and SVK metadata. SVK apparently is solving many of the main problems with SVN (memory for example) but it should be seen as an extended client, not a replacement.

However, when FreeBSD started to look at another tool, SVN was not ready and Perforce was choosen and we have been pretty happy with it; it helped a lot getting projects such as SMPng and others up and running (and finally integrated into the main CVS tree of course). We also have a regular export from the Perforce tree in a special CVS tree to allow people to see what is going on on these projects[11].

Perforce is a very nice *centralised* VCS (supports fast and easy branches, is fast and well supported) but in that respect, it is even worse than CVS: all operations are done through the network and one has to be connected to the server to do anything[12]. Perforce is also snapshot-based, see SVN point 4 above.

Another problem, as we will see in more details through section 2.6, is that it is closed-source proprietary software and that creates a questionable dependency for an open source project.

One last bit of information about CVS: after the last round of CVS security advisories, some OpenBSD folks have decided to rewrite CVS completely to get a more secure source code base, see the OpenCVS website[19]. Some of the enhancements planned a long time ago for CVS like atomic commits and rename seem to be forthcoming but it has not been released yet (Oct. 2005).

[11]Note that it allows external people to get the code but does not give an easy way for them to hack on it and contribute back.

[12]The author used to be a Perforce user for 5 years but the lack of easy synchronisation between repositories became too much and he switched to Arch.

2.5 Enter the distributed VCS

In 1999-2000, Larry Mc Voy, formerly of Sun, Inc., started his own company called BitMover, Inc. around a new product named BitKeeper. BitKeeper (BK in short) was a distributed version control system based on the venerable SCCS[13], extending it to cope with repository cloning and merging.

Apart from being close to its ancestor SCCS, it brings the distributed aspect to revision control and with it a whole new way of working and sharing source code.

Up to now as we have seen earlier, developers had to share code either through a common tree (CVS, Perforce, and so on) or through the much more cumbersome way of generating patches. With BK it becomes as simple as cloning a given repository and start hacking on it with pull/push mechanisms to share code and patches.

With its vision of *a repository is a branch*, generating a branch is the same as cloning meaning that you can have as many branches as you want and that:

- They are cheap (so you can throw them away if not needed anymore or a dead end),
- It is easy to merge between all these branches as the system knows where the branch was created from and which changesets are present.

The concept is rather new and we should thank McVoy for pushing the limits for all VCS developers because it was the starting point of what we have now. BK really took the lead of DVCS when Mc Voy, for good or worse, convinced Linus Torvalds to finally start using a VCS for the Linux kernel.

2.6 The BK debacle

It was a big change for Linus (not so much for the developers' community as many of them had started using CVS for their own trees) and it also pushed many people towards using a DVCS.

Many people recognise that BK works well, is reasonably fast and it does the job[14]. These people also generally agree on two points:

- The license is one of the worst I've seen. Not only there are many unacceptable restrictions (like being prevented to work on developing any other VCS during the license validity plus one year and being forbidden to reverse engineer the wire protocol and product – something one can not forbid in the EU) but if you wanted a "free" license, you have to send all your commit logs to them,
- Worse: all the generated metadata that makes it interesting (like who branched what and when and all that) is considered as *proprietary* data by BitMover, Inc. even though it concerns a FOSS project![15]

Add to these the fact that Mc Voy has constantly been saying on several mailing-lists (mainly the `linux-kernel` one) how difficult it was to write such a product, how costly it would be and don't bother trying to reproduce it, it is too difficult and so on[16].

To add insult to injury, he also said that if anyone tried to reverse engineer anything related to BK, he would change the wire protocol and prevent people to do it.

In the end, what had to happen did: In April, 2005, Andrew "tridge" Tridgell, of Samba fame, tried to reverse engineer the wire protocol – which proved to be trivially easy thanks to BK itself – and BitMover decided to revoke all "free"

[13]The author's guess is that McVoy used it at Sun and felt it was a good starting point.

[14]The author tends to agree even though there are some questionable things in it like the 65536 limit on the number of changeset – it is now fixed – and the heavy use of `system(3)` throughout the binary.

[15]Note that this is very different from Perforce: both are proprietary software but the format of Perforce's metadata is known and there is even a Perl `p5-VCS` module for it meaning that you are not locked by using Perforce.

[16]It is interesting to note that the development of Mercurial started in March 2005 and is now pretty close to BK featurewise in only six months.

[17]As it is, BitMover is still trying to stiffle the competition by forcing people not to work on free projects; a major contributor to Mercurial has been recently "asked" not to work on it as long as his company has a commercial BK license.

licenses therefore putting Linus and other Linux developers in a difficult position[17].

I will not dive into Linux politics and what happened but we must see that the whole debacle was the driving force behind the current trend of DVCS and spurred development of many systems now available.

People are now aware of the problems and caveats of distributed developement and the solutions behind them. We now have several very interesting VCS, some close to Linus' own `git`[20] (`cogito`[21] and in some ways Mercurial) and many others, each with its own set of interesting features (Darcs and the theory of patches[22], Monotone[23], and so on).

The second consequence is that people are hopefully convinced that using a proprietary VCS as the main one is a *very* bad idea.

3 The FreeBSD context: figures and processes

The FreeBSD project started in 1993 just after NetBSD using 386BSD as its base tree. Originally planning to be 386BSD 0.1.5, it finally became FreeBSD as both Bill and Lynne Jolitz, the original authors, refused contributions and maintaining the various patches became too cumbersome.[18]

3.1 Figures

The current CVS tree is the second one we have been maintaining. Due to legal restrictions coming from the AT&T/BSDi/CSRG lawsuit, we were forbidden to keep on using and distributing the FreeBSD 1.x repository so a whole new tree was created in 1994 with the import of 4.4BSD-lite.

The whole repository was broken up into four a few years ago to be close to the organisation of committers: we have *src*, *ports*, *doc* committers and those who can commit in several or all categories. The *doc* committers includes those working on the `www` subtree. The following table lists the sizes as of mid-Sept. 2005:

Repository	Size (MB)	Directories	Files
doc	183	1653	6171
ports	903	43490	124338
src	1402	9030	60708
www	112	595	3479

I do not have figures about the number of changesets as the notion doesn't exist in CVS but when P. Wemm did some conversion tests back in 2000 during our evaluation of Perforce, we were already at more than 75000 changes in the Perforce converted tree. I estimate the current tree to have more than 200,000 - 220,000 changesets by now, all repositories considered (more on these figures below in 6.2). .

When using *CVSup*[19], all the repositories can be combined in a single one through symlinks as it is easier to work with. Note that of course, having the entire repository does not allow to commit (or it would completely mess up with the next *CVSup* run). One feature was added to *CVSup* to ignore a special branch and allow for local modifications while syncing the CVS tree but that is only a *hack*.

3.2 Development process

Today, the developement process in FreeBSD is pretty straightforward: committers have access to all repositories, the main difference between types of committers will reflect in the commit log. If a *doc* committer checks in a change in a manpage in `src/share/man`, the commit message will say at the top that it was done by a *doc* committer.

Committers are strongly advised to *CVSup* the repository on their local machines, edit, compile and test and then push to the real one by overriding the repository path. That way, the network and the CVS machine are not overloaded and we can keep disk space at a reasonnable level. Of course when a commit must be tested on the FreeBSD cluster with different machines and architectures and the committer doesn't have the local resources, local checkouts are allowed.

[18]The complete history of FreeBSD and its relation to the other BSD can be found on the web, I will not reproduce it here.

[19]CVSup: CVS-aware replication tool – `http://www.cvsup.org/`

The central repositories are also responsible for sending the commit logs to the various mailing-lists (`cvs-all` has everything but there are also broken down for specific subtrees such as `cvs-ports` and `doc`); this is an important part of the process so any system aiming to replace CVS must be able to offer and support such features.

In day-to-day operations, we see CVS's flaws in action when we need to move things around (it can be because a port was not imported in the right place or in case of code reorganisation); we have some people called *CVSmeisters* that are specifically allowed to manipulate the repository and execute the unfortunately common *repocopy* operation[20]. That way, history is not lost.

It is unfortunate that we have to manually edit the repository fairly often[21] but there is no other way due to CVS's limitations.

The two products we are going to evaluate have ways of replicating a given repository to remote sites but we will keep on exporting all changesets in a CVS repository for easy duplication through *CVSup*, anonymous CVS usage and more generally because it is so well-known even by some non-technical people. The nice thing is that converters to CVS are not difficult to implement or find and it is easier to go from a changeset-based system to CVS than the reverse.

3.3 Release Engineering

The FreeBSD project maintains several branches in parallel to support our notions of STABLE and CURRENT trees. We also have security branches on which only security fixes are applied (this happen to all STABLE versions after they have been released) and they are supported for a limited amount of time (that varies from branch to branch and can be more than 18 months). We have also recently allowed non security fixes in the release branches (`RELENG_*`). To help release builds, we have some period of time during which the trees are either completely frozen or strictly controlled by the Release Engineering team (also known as the `re@` alias) and `portmgr` (for the `ports` tree). Such freezes happen independently in the *src*, *doc* and *ports* trees but the goal stays the same: to be able to have a *stable* tree to cut a release from.

These procedures are somewhat of a necessary pain because CVS is not as we've said before very helpful with its branch handling (sometimes the trees stay semi-frozen for weeks). This is one of the main reasons not to branch the entire ports tree for each release: It would be taking too much time to tag the tree as every single file needs to be written into and we need to block everyone through various scripts we have developed over CVS.

Switching to another VCS requires these issues to be cleanly handled.

3.4 FreeBSD requirements

From the previous sections, we can extract a set of FreeBSD requirements that we want a future tool to handle.

- Atomic commits to get *real* changesets
- Easy & cheap branches (and merging) and tags to enable parallel lines of development (that is essential for projects like *SMPng* which have a very big impact on many source files)
- Fast system for common operations
- Ability to keep and distribute a "reference" tree, knowing that it should also be exported to CVS
- Ability to rename files within directories without losing history
- Ability to help simplify the way we handle releases (and freezes, slushes, ...) in order to avoid locking the trees.
- Ability to digitally sign revisions or repositories to avoid file corruption and to detect unwanted modifications
- Automated or mechanically assisted merging

[20]It is achieved by `cp` the `foo,v` file from the old place to the new one.

[21]On the other hand, manually editing is faisible, which can save your day if you have a repository corruption.

- Ability to work *offline* – like on a plane – without requiring too much work: not only being able to list differences but also to commit

Most of these requirements can be met by centralised VCS but the second and last points are those pointing to a non-centralised or distributed VCS.

4 Is Arch/bazaar suited to FreeBSD?

In this section, *Arch* is the "protocol" (for lack of a better word) designed by Tom Lord and both tla[24] and Bazaar are implementions of this protocol. Both implementions are compatible with each others (unless you specifically ask at creation-time for a `baz` archive which `tla` can not read) but Bazaar has the backing of a company (Canonical, Ltd.[25]) and is the only one currently maintained. Tom Lord has announced he was stopping all developments on both tla and revc[22].

Arch has some unique features among the DVCS:

- It is both a VCS and a cataloging system:

  Everything is divided into archives, whose name generally contains the email address of the developer like in `lord@emf.net--gnu-arch-2004`.

  Archives contain the equivalent to CVS modules named here *categories*. Whereas most DVCS use as many repositories as you have branches, *Arch* still uses a separate sandbox as workarea.

  Categories are the main work unit in *Arch*; they can be checked out for editing, branched (here it means both as a local branch and as a remote one) and versioned. Branches and versions are specified in the full name of the category, separated by "`--`" like in `calife--pam--3.0`

  The main inconvenience is that you must type a lot more to refer to something managed by *Arch*[23]

- Whereas many modern VCS try to duplicate the well known UI of CVS, tla has a lot (and I really mean *a lot*) of different commands for dealing with archives, categories, revision libraries, branching and merging and so on. Bazaar has tried to redesign the UI to be easier for beginners but still, the output of `baz help | wc -l` shows 187 lines...

- Both tla and Bazaar use weird-looking filenames for temporary subtrees and filenames (with ++ or „ as prefix). The metadata directory in the sandbox is named `{arch}`. Most VCS use `.something` (and `_darcs` for Darcs[26]) to store that information. That is not a big point but it does confuse newcomers.

- *Arch* eats a lot of diskspace. In addition to the archive which contains directories of changesets, you will need space for the checked out categories and either a "pristine" copy of the sandbox (a gzipped-tar file) inside `{arch}` or a revision library (a complete tree of hardlinked files for most or all checked out/merged revisions).[24]

- *Arch* needs to uniquely identify all files managed by it so there are several ways to generate a unique id and to tell *Arch* what it is:

  names The filename is the *id* itself, it is obviously not the recommended way for normal operations

  explicit It is analogous to how CVS works, you use the `add` command to `baz` to attach an *id* to the file[25]

[22]`revc` was supposed to be Arch 2.0 with a whole new storage backend (close to `git`), no more categories/branches/versions and a different archive format along with an heavy use of SHA-1 checksums everywhere.

[23]Fortunately, there are completion modules available for the common shells – `zsh`, `bash` and `tcsh`. Trust me, you can not live without such a completion module.

[24]To be fair with *Arch*, SVN has also a pristine copy of your files inside `.svn`.

[25]It will be stored in a special sub-directory called `.arch-ids`.

tagline This one is special: *Arch* will look into the first and last 1 KB of each file for a special string[26] and use that as unique *id*.

This unique *id* enables *Arch* to track file/directories renames more or less automatically (which is nice) but also, in the *tagline* case, complicates third parties code as you are not really allowed to modify it.

All of these items makes *Arch* rather complicated to use, especially for beginners (in the VCS world) but really, I have been very happy with it for two years. I would even say that the namespace issue for categories forces users to think a bit more on how to organise things in an archive which is not without value.

If we want to use it for FreeBSD, there are several things that we need to look at, mainly because of the design of *Arch* and the whole category feature. Do we want a single category named `freebsd`, separated into branches (like `freebsd-current` and `freebsd-stable`) eventually with a version number or do we want to use a category per subtree?

This is a big point and one that will have an important impact on *Arch* speed because it tends to walk the whole tree several times during commit and other operations (that is called running an "inventory" in Arch-speak). A given category is pretty much independent, if you want to group categories to form a complete source tree, you have to use a special mechanism called *configs*: you have a category with a special file with all the other categories you want to include (pretty much like the `CVSROOT/modules` does for CVS.) Then you use the `build-config` command to extract all categories and create the tree.

The big problem that comes from configs usage is that as I said before, the work unit is the category. What it implies is that commit also works on categories, not on a source tree built with configs… *Arch* has a way to iterate on all categories coming from a config but:

- It is a bit cumbersome although you will end up with writing a lot of aliases or shell scripts to automate this,
- The changeset is not global either: you will have one commit per category.

The second point is pretty much a killer in my mind. If you want to do a sweeping change in `/usr/ports` for example, you want a changeset of the whole thing, not more than 12000 changesets… You also don't want 12000 mails to be sent to the `cvs-all` mailing-list.

Another subtle characteristic of *Arch*: when merging multiple changesets between archives, on the receiving end, there will be only *one* changeset incorporating all the changesets (named *patch logs* in *Arch*) and users will be able to see only the summary lines for each embedded changeset. If the sending archive is available, full commit messages can be retrieved of course. This is clearly different from BK and Mercurial where every remote changeset is included as-is.

Bazaar satisfies one of our requirements: every commit can be digitally signed with PGP/GPG, this is an important security feature.

Last but not least: Bazaar is rather complicated to build; it does not use the autoconf system but its own home-built system (called *package-framework*) and has dependencies on several external packages such as `gpgme`, `libgpg-error`, `neon` (for http/webdav access) and very recent versions of various GNU utilities (`patch`, `diffutils`). It does complicate its possible inclusion in the main FreeBSD tree. tla is not as complicated – although it does use *package-framework* as well – but tla should now be considered as dead (and probably not worth maintaining due to the above limitations).

4.1 Common operations

We will take `/usr/src`[27] to make most of our tests, knowing that it is a moderately large tree (checkout is around 448 MB) with more than 33000 files inside 3766 directories…

[26] `<comment characters>arch-tag: <unique-tag>` (people often use UUID[27] for that purpose: `/* arch-tag: b11c0274-29ee-11da-9b43-000d93c89990 */`)

[27] The main problem is that Bazaar 1.5 keeps on dumping core on my FreeBSD 4.11 system when using `/usr/ports`

In order to avoid wasting too much disk space among developers, each of them having possibly several checkout copies lying around, we can define a *revision library*. This area will hold hard-linked copies of the checkout files and so only the modified files will take more space between all users. Of course it does eat space (but we hope to reduce the overall diskspace requirement) and all developers must configure their text editor to break hard-links to avoid corrupting this revision library.

At first, we will try to import that as a single category because we want changesets to span the whole tree. To have Bazaar work as transparently as possible, we will use the *names* tagging method.

Operation	Time	CVS equiv.	Time
`baz import`	11:21	`cvs import`	4:18
`baz get src`	3:28	`cvs co`	14:43
`baz commit -s`	4:29	`cvs commit`	11:52
`baz status`	6:05	`cvs update`	5:22
`baz status`	3:33	`cvs update`	idem

NOTES:

- The first `baz status` command generated a revision library entry while the second one is just using it.
- The `baz get` command used the revision library to hardlink all files in it.
- For some operations, system limits (see `get/setrlimit(2)`) had to be raised (datasize in particular) or `baz` would dump core.

Bazaar is clearly faster than CVS but not by a large margin and some operations require multiple traversal of the whole tree (the inventory system) which slows it down. `commit` can take an optional list of file names to be considered by the commit itself but on a very large tree such as `/usr/ports` it is really painful to list all modified files. The correct method is generally to have a *wrapper* command around the actual command-line interface (CLI) that builds this list and hands it out to the tool when committing.

Disk space requirements must also be considered: If a given tree is N MB, it will generate N MB as a revision library entry and gzip(N) MB in the archive itself. Commits are stored as compressed changesets so it takes much less space. For each commit, a plain text version of all modified files will be added in the next revision library entry and the rest is hardlinked. Revision libraries must be pruned regularly of course as you'll accumulate revisions you'll probably never extract gain.

The alternative is to avoid using a revision library but then, Bazaar will generate a complete copy of the checkout files – called a *pristine tree* – below `{arch}` which does take as much disk space as the checkout tree…

4.2 NOTE

It must be noted that most of Canonical's effort has been recently concentrated on the next generation of bazaar: `bzr` *aka* bazaar-ng *aka* bazaar 2. Version 0.1 of `bzr` has just been released (Oct., 11th, 2005), incorporating a very important change in repository format: it is now using the *weave*[28] format instead of the full-text one previously used (the same as `git`). It is too early to really test `bzr` as it is pretty young and performance is still lacking but it is very promising.

This is an important change and one that will make Bazaar 2 much more interesting. There will be an upgrade path from Bazaar 1 to Bazaar 2 but they are completely different in design

5 Mercurial to the rescue

While working with *Arch* and trying to see how to overcome the limitations and design problems described in 4, I found *Mercurial*. Following what we have seen in section 2.6 and the appearance of Linus' `git`, exploring what have been started with Darcs and *Monotone*, Matt Mackall announced he had started to write a DVCS[28][29].

Another reason to look at Mercurial is that Bazaar 2 was far from being feature-complete (without even thinking about performance)

[28]See `http://bazaar.canonical.com/BzrWeaveFormat` for a detailed explanation about weaves.

[29]Mercurial was started because of the BK debacle according to the author.

[30]See the roadmap: `http://www.selenic.com/mercurial/wiki/index.cgi/RoadMap`

What is really nice about Mercurial is not so much its speed – although it is important and impressive – but the fact that in a few months, it has grown into a nearly-mature product, with almost all features you could ask for a DVCS[30].

Add to that:

- A very friendly and open-minded author
- A growing community both on the mailing list[31] and on IRC (#*mercurial* on the Freenode[32] network).
- A relatively small and portable system compared to others like Monotone or Arx[29]
- Written in Python[33] without too many external dependencies

You end up with something small, fast and easy to use and setup. As we will see in the timing section 5.2, its handling of large trees is adequate for most usage and it is evolving without breaking too many things from one version to the next (no repository format change is foreseen in the near future for example, something that other VCS have done on a regular basis).

Of course there are a few things that need to be implemented to have a complete system like:

- Better handling of binary files. You can put binaries in a Mercurial repo but you will not be able to use `hg export` to submit; the only way to do it is either to use the `bundle` command that create a binary version of a set of changesets or to use the `push/pull` mechanism.
- Better rename/move support. At the moment, history is preserved by the `copy/rename` operations but it is not available to the user so it appears to be lost[34]
- Better support for managing changesets within a repo: currently, there are different way to revert a changeset or a set of changesets (`undo` only reverts the last one). It means that if you make a mistake, it may become a bit difficult to undo it.
- Support for digital signature of commits (most of the infrastructure is there but needs to be completed and on by default). UPDATE: the `gpg` extention has been integrated in the main Mercurial tree and just need to be enabled in `hgrc(5)`.
- Full permissions are not versioned except for the 'x' bit. Permissions are kept but if you change a file from `600` to `664`, it will not be not taken into account.
- Lack of Internationalisation (i18n) support. It is necessary to lower the entry bar for many people.
- More documentation

All these should be corrected for the *1.0* release during 2006.

All these reasons made the author choose Mercurial first for his own usage and second to include it in the scope of this paper. The rather fundamental technical differences between *Arch* and Mercurial designs do not have a big impact on section 6 about processes and policies changes that are needed when moving from a centralised to a distributed VCS. These differences will have an impact on the technical side of the migration and setup of course.

Apparently the folks managing OpenSolaris didn't find these flaws blocking enough and choose Mercurial over Bazaar-NG and Git after careful evaluation in March 2006 [35])
These main differences between Mercurial and *Arch* are:

[31]Mailman interface: `http://www.selenic.com/mailman/listinfo/mercurial/`

[32]See `http://www.freenode.net/`

[33]While Python is not the preferred language of the author of this paper, it is easy to understand and thus to contribute [to the project]

[34]This is true as of version 0.8.1 released on April, 7th, 2006

[35]`http://www.opensolaris.org/jive/thread.jspa?threadID=7611&tstart=0`

- *Arch* follows the traditional design with one side the archive/repository and on the other side the working trees/sandboxes
- Mercurial does not force a specific namespace on repository and module naming (like *Arch* does in `archive/category--branch--version`)
- In Mercurial, there is no inventory like the one done in *Arch*, no tagline/explicit/implicit/names method of include/exclude files from being versioned.
- The work unit is the tree/branch, not a subset of it

5.1 Technical specifications

Mercurial shares some common characteristics with the other available DVCS:

- A repository is a branch (this is a simplification as you can have several branches within a given repository)
- The working directory is the repository, there is no *sandbox* like in CVS or SVN
- Branches are cheap and the main way to replicate (called *cloning*) repositories
- You can lay down tags on a given revision but with a twist: tags can be either local or global, the latter means that if you clone a repository, you will get the tags along the way.
- You must have a *merging* tool like `kdiff3` or `tkdiff` to handle any conflict during merging. It must be noted that merging is done on a separate branch within the repository first then you merge the result with your own local changes. This approach generally lowers the number of conflicts when dealing with external sources.
- It has an integrated *CVSweb*-like interface, either through a CGI script or through its own `hg serve` command.

It has also an interesting technical feature, shared in principle by *Arch*, the various files in the `.hg` tree are append-only. That means that it is a bit more robust (compared to the RCS file format where everything including tags are stored in a single `,v` file) and that going back to a previous revision is done through simply truncating the file.

The storage method used seems to be pretty efficient, specially when compared to the default `git` backend where full files are stored for a given revision and various tests done by the author ([30] for example) shows the differences. I do not believe that the fact that hard disks are now cheap is a good reason to waste that space.

Something interesting has been available for Mercurial for quite some time: an extention to manage "stacks" of patches has been written. This extention, called `mq` does something similar to `quilt`[36]; it allow to manage a series of patches by keeping track of the changes each patch makes. Patches can be applied, un-applied, refreshed, etc.

UPDATE: `mq` has been integrated post-0.8 and is now bundled with Mercurial. You just need to enable in `hgrc(5)`.

5.2 Tests timing

We take the same `/usr/src` tree to make comparisons with *Arch* and CVS.

Let's assume we want to put `/usr/src` under Hg, discarding the previous CVS history for the moment[37].

Operation	Time	CVS equiv.	Time
`hg clone src`	3:09	`cvs co`	14:43
`hg commit -A`	5:12	`cvs import`	4:18+14:43
`hg commit -m`	0:09	`cvs commit`	5:32
`hg status`	0:06	`cvs update`	3:30

NOTES:

- `clone` and `co` don't do the same exact thing as there is no history in `co` case.
- `cvs import` creates the repository but we need a checkout to work; Mercurial doesn't need that phase as the working directory is the repository.

[36] `http://savannah.nongnu.org/projects/quilt`

[37] Due to CVS design and misconception, converting a whole tree is rather complicated and *very* slow.

- `cvs update` is not strictly equivalent to `hg status` but `status` is much more verbose.

It is *very* fast. It is fast enough that we don't really care about trying to use sub-trees (see 6.1) as it gets more complicated to submit patches.

As Mercurial can handle the `/usr/ports` tree, here are some timings:

Operation	Time	CVS equiv.	Time
`hg clone ports`	9:18	`cvs co`	16:35
`hg commit -A`	10:34	`cvs import`	4:41+16:35
`hg commit -m`	0:39	`cvs commit`	11:52
`hg status`	0:52	`cvs update`	5:22

Even on a much larger tree – `/usr/ports` is more than 124,000 files in more than 32300 directories – Mercurial manages to stay fast.

Repository overhead is small too, although on pathologic cases such as `/usr/ports` with a lot of very small files, the fact that you have *two* files for each versioned files is showing:

Tree	Size	`.hg` size
`/usr/src`	417 MB	227 MB
`/usr/ports`	430 MB	358 MB

The nice thing is that as the trees will accumulate revisions, the way Mercurial does store changesets is very efficient: if the delta between the next version and the original is bigger than some amount, the new version is stored compressed in its entirety. It ensures than we don't need a huge amount of data to reconstruct *any* version of a given file. Add to that the fact that all files below `.hg` are append-only, you have a repository that can resist corruption better than others.

The author of this paper would have liked to do more speed comparisons, especially when working with older branches (an area where CVS is rather weak as it needs to go back and forth with in the repository to reconstruct a branch) but the difficulties with repository conversion prevented that. The author would like to point out that due to its design, Mercurial would probably shine in that respect because generating an older branch is consist of merely cloning (something which is fast) the given reference tree…

Since my talk at EuroBSDCon 2005 in November, Chris Mason has been working on some very interesting bits for Mercurial: using revlogng [38] as a starting point, he has written a new repository format that does pack all files within a directory into a single revlog file. Although it currently has some bad side-effects[39], it has some benefits in terms of disk space and clone times.

After Matt Mackall mentioned this issues, he suggested another modification whereas a file will be packed if the size of the data + index is below a reasonable threshold (128 KB for now) and unpacked otherwise. This approach will greatly reduce the repo size because in `/usr/ports` most files don't have a big history.

On a dual P4/2.8 GHz machine (with HTT so it sees 4 CPUs) with fast SCSI U320 disks in mirror mode, I get the following figures[40]:

Repo format	`.hg` size	Files	Directories
default	793648	269929	27565
revlogng/mixed	527352	134986	27566
revlogng/packed	398002	55111	27565

`clone` times are also down from 25 to 19 minutes (which is 20% less). For mixed mode, clone time is 21 mn, close to the fully packed version. The mixed packed case seems to be a good compromise there.

6 How to get this to work: processes and policies

A tool, however powerful, is not enough to support a whole project running and do it that way in a reasonable form, especially when dealing with volunteers. A project this size (more than 300 people, working around the world with different timezones) has to have some kind of processes and policies.

Since the beginning of the FreeBSD project, everything has been built upon CVS and upon its features and flaws. In particular, almost all the constraints we have now for Release Engineering and the whole set of policies of freezes, slushes

[38] See `http://www.selenic.com/mercurial/wiki/index.cgi/RevlogNG` for a possible modification to the present repository format

[39] See `http://www.selenic.com/pipermail/mercurial/2006-March/007155.html`

[40] The difference in size for `.hg` are due to difference in file-system characterisrics between the two machines used for testing

and al. have their roots in CVS in one way or another.

It is the opinion of the author of this paper that it is time to review them, classify them as CVS-specific (or not) and see how they would have to evolve if the FreeBSD project was to switch its VCS over to Mercurial[41].

A distributed or decentralised VCS enables a more parallel way of working, facilitates working in different trees and branches without the fear of a complicated merge and without playing with patches. The fact that it enables offline work is a very much needed feature; likewise, merging from offline trees is no different from merging different branches and is as easy.

We should also note that some software would have to be written or changed to adapt to the new VCS as some assumptions coming from a CVS-oriented world are not true anymore: a central server with all the related aspects like pre-commit checks, post-commit triggers and so on.

6.1 FreeBSD environment

If we look at the FreeBSD requirements in 3.4 and try to answer them, we will see that the first three are easily met by Mercurial as they are part of its design. The last one is the key point what we will concentrate on. The point here is not to disturb the developers too much.

Let's see what would be needed to reproduce a CVS-like environment:

- A "reference" tree that people can clone from just like people use *CVSup* now to get the *official* source tree.
- A way to handle either merge requests from the various developers or a way to queue patches sent through various sources (email for example) for integration in the "reference" trees *aka* a patch queue manager[42].
- A way to generate commit messages to be sent to various mailing-lists; if we have the above request satisfied, then the patch queue manager (PQM) is the obvious candidate for this.

When one wants to make some modifications to a given tree, he/she will clone the repository, hack on it and then submit the changesets to the PQM. One easy way to do this is to have a cloned tree that is only updated through `hg pull`, serving as a *local* reference tree and the developer will clone this one at will for specific purposes.

To maintain coherency with these cloned trees, he/she will regularely merge from this reference tree into the other ones. This is where having a fast VCS is interesting because having to wait half an hour just to be able to edit something is not really productive.

Even though Mercurial is fast for cloning big trees, it still takes some time. A possible solution to this problem could be to create sub-trees on demand: when you want to do a small modification, you just go to the sub-tree, `hg init` followed by `hg commit -A` to *import* the sub-tree in a little repository. It is then very fast and easy to generate a diff and submit it to the main tree (then forget the sub-tree with `rm -rf .hg`). Of course, it would be for small modifications which don't require you to keep the repository.

As for the patch queue manager, PQM has already been modified to work with Bazaar 2 and ArX so I think that adding Mercurial support should not be too difficult.

One area where things become easier is *Release Management*: there is no real *technical* need for ports/src freezes/slushes as it is just a matter of cloning the "reference" repository into a branch/release one, making it available through the PQM and use different rules for merging. Freezes will always happen in order to get the tree stabilised up to the point we can cut the release out of it of course.

Likewise, there is no need to manually edit the repository, no more repocopies or tag sliding[43],

[41]Note that most of what we will say there is applicable to any distributed VCS, the key here is *distributed*.

[42]Like the one used by Canonical – `http://mirrors.sourcecontrol.net/robert.collins@canonical.com-general/`

[43]Tags are static in CVS and references a specific file revision. When preparing a release, a critical bug can be fixed and the release/branch tag be modified to reference the fixed revision. This is manual intervention in the repository.

thus simplifying the whole repository administration.

6.2 Repository conversion

The problem of converting the history of the project is a complicated one as the tools we are coming from and the one we would use are completely different in several ways, the major one being that having a repository for each branch complicates conversion as the tool used for that should be aware of branches and should generate a different repository when it comes across a branch tag. At the moment, none of the available tools support that. *Tailor* can convert whole branches into a given repository but we would have to manually do it for each branch starting at the branch point. It can only follow a given path, not descend in other branches.

Other complicating factors includes encoding of commit messages (do you want to convert everything from ASCII or ISO into UTF-8 or UTF-16?), tags (the notion of tags varies betweeen VCS, …

To give everybody an idea of what repository conversion is about, the `/usr/ports` tree already mentioned has 138696 changesets (mid-October 2005) which is a *lot*. `/usr/src` is around 117233 changesets. Last time the author tried to convert `/usr/src`, it took 3 hours for less than 700 changesets (estimating the total time is left as an exercise for the reader) and consumed close to 1.2 GB of memory.

A few weeks after EuroBSDCon 2005, a new tool for converting CVS repositories into Mercurial ones has been written by a DragonflyBSD committer: Simon "corecode" Schubert called `cvs2Ohg`. It is also in python but has been optimised for fast operations. It has an incremental running mode meaning that it can be used for a continuous conversion, updating regularely a given CVS repository. Such a gateway has been implemented on `http://hg.fr.freebsd.org` by Mathieu Arnold, a FreeBSD committer. Updated every two hours approximatively, it gives a changeset-based view of all the repositories.

The only remaining problem is branches of course. `cvs2Ohg` does not support branches either so only `HEAD` is available for *src*. Tags are not supported either but the way they are used in CVS makes the conversion a bit difficult: tags in Mercurial are global whereas they can cover only partial trees in CVS. Most of the tags in the tree were made for branch points or releases.

As one last update to this paper, one of our very own FreeBSD committers, Pierre Beyssac, has just added branch support to `cvs2Ohg` and we are now able to get `RELENG_6` sources in their own Mercurial repository.

It seems that `git` now has a working converter that does understand branches and an entry has been put in Mercurial Bug Tracking System[44] about that.

7 Conclusions

It is clear for the author that such a migration should be carefully planned over a few months and that the different issues mentioned before should be fixed. Mercurial itself is still lacking features but is evolving quite fast. Other tools were outside the scope of this paper and maybe should be evaluated but at the moment, only Mercurial has enough features and is stable and fast enough for our purposes.

The infrastructure still needs to be written or adapted to Mercurial and the big question on how to import the previous CVS history is something that should worked upon.

The problem with the repository conversion tools may mean that we would have to maintain both CVS and Mercurial as long as we support older branches or finding a way to partially convert them.

A not-so-minor point to consider: Mercurial is not written in a language that we have in core FreeBSD so the question becomes do we want Python in the core OS or can we accept that our main VCS will force people to install Python from ports. From a pure maintainability point of view, ports is easier.[45]

[44]`http://selenic.com/mercurial/bugs/`

[45]Tcl and Perl were both at some point in time part of core FreeBSD and they were both removed.

The learning curve of the new tool and the new ways of working are also important. The UI is a big part of that and Mercurial tries to mimic the old but well-known CVS one (where applicable of course).
Meanwhile, there are some advantages coming from the distributed part of the new tool:

- There is no need to have such complicated pre-commit and post-commit tools such as the ones we have now in `CVSROOT`, the PQM will manage all that.
- You don't need a central server with SSH keys, Kerberos or any form of access control; people just clone the "reference" repository and work from that. Access will still be needed at the PQM level of course to distinguish between committers, developers and users.
- The new capabilities of Mercurial could open the way to new working-styles like task-oriented branching and merging (like we do in Perforce now). With a possible link to the bug report database, we could think having a PR automatically closed when a task is done.

It is the wish of the author of this paper to help the FreeBSD Project to start thinking about a possible switch. It will be up to the FreeBSD Project to decide whether this is a worthwhile project to engage ourselves into of course.

Discussions about VCS and their use with the FreeBSD Project happen regularely in various mailing-lists. It seems that SVN, especially with SVK has quite a significant number of supporters. Knowing the pain of converting our significant history, the author can understand why although he still thinks a DVCS would be better in the long term. These discussions show that the above wish of pushing people into thinking of a switch has been achieved at least.

8 Thanks

The author would like to thank Phil Regnauld, Mark Murray, Anton Berezin and Robert Watson for reviewing this paper over a rather short period of time. Much appreciated folks!.

Thanks also to Mathieu Arnold & Absolight for setting up the Mercurial repository and conversion crontabs on `hg.fr.freebsd.org`.

And special thanks to Élodie for pushing me into writing this paper.

References

[1] Various authors at Canonical, Inc., *Bazaar, an Arch implementation*. http://bazaar.canonical.com/.

[2] Matt Mackall, *Mercurial, a distributed SCM*. http://selenic.com/mercurial/.

[3] The FreeBSD Project, *FreeBSD, The Power to Serve*. http://www.FreeBSD.org/.

[4] Inc. Perforce, *Perforce, The Fast Software Configuration Management System*. http://perforce.com/.

[5] Mark J. Rochkind. The source code control system. In *IEEE Transactions on Software Engineering (Vol. SE-1, no. 4)*, December 1975.

[6] Larry McVoy, *SCCS & Source mgmt*. 1997. http://lkml.org/lkml/1997/5/23/105.

[7] Dick Grune, *Concurrent Versions System CVS*. 1986. http://www.cs.vu.nl/˜dick/CVS.html#History.

[8] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[9] Sourceforge team, *Sourceforge software development hosting system*. http://www.sourceforge.net/.

[10] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Number 1-932111-81-6 in ISBN. Paraglyph Press.

[11] Andy Hunt Dave Thomas. *Pragmatic Version Control Using CVS*. Number 0974514004 in ISBN. Pragmatic Programmers, 2003.

[12] Per Cederqvist, *Version Management with CVS ('official' manual)*. http://ximbiot.com/cvs/manual/.

[13] Lele Gaifax, *Tailor.py, A tool to migrate changesets between VCS*. http://www.darcs.net/DarcsWiki/Tailor.

[14] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Version Control with Subversion*. Number 0-596-00448-6 in ISBN. O'Reilly, 2004.

[15] Mike Mason. *Pragmatic Version Control using Subversion*. Number 0-9745140-6-3 in ISBN. Pragmatic Programmers, 2005.

[16] ChiaLiangKao, *svk, a decentralized version control system, using Subversion*. http://svk.elixus.org/.

[17] Martin Pool, *Integrals and derivatives*. July 2004. http://sourcefrog.net/weblog/software/vc/derivatives.html.

[18] C. Watson B. Robinson, J. Hess and ISHIKAWA Mutsumi, *Debian X Strike Force Hackers' Guide*. http://necrotic.deadbeast.net/xsf/XFree86/HACKING.txt.

[19] OpenBSD, *OpenCVS, a FREE implementation of the Concurrent Versions System*. http://www.opencvs.org/.

[20] Linus Torvalds, *Linus' own version control software'*. http://www.kernel.org/pub/software/scm/git/.

[21] *Cogito, a version control system layered on top of git*. http://www.kernel.org/pub/software/scm/cogito/.

[22] David Roundy, *Theory of patches*. http://www.abridgegame.org/darcs/manual/node8.html#Patch.

[23] *Monotone, a free distributed version control system*. http://venge.net/monotone/.

[24] Tom Lord, *tla, a revision control system*. http://www.gnu.org/software/gnu-arch/.

[25] Ltd. Canonical, *Canonical main web site*. http://canonical.com/.

[26] David Roundy, *Darcs, a revision control system*. http://www.darcs.net/.

[27] ISO (International Organization for Standardization). *Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)*. ISO organisation, 1996.

[28] Matt Mackall, *Mercurial v0.1 - a minimal scalable distributed SCM*. http://www.ussg.iu.edu/hypermail/linux/kernel/0504.2/0670.html.

[29] *ArX, an easy to use distributed revision control system*. http://www.nongnu.org/arx/.

[30] Matt Mackall, *Patch: Mercurial 0.3 vs git benchmarks*. http://lwn.net/Articles/133594/.