Version control using Perforce

Fast and powerful

Some of you may have used CVS more than you really want to, as I have, have run into its limits, as I have, and be looking for something better, as I was. In this article I'm going to describe why I was looking and what I found.

BY ARNT GULBRANDSEN

When using CVS, I was unhappy with its bad support for branches, its rotten performance over the network, its undocumented and weird behaviour when I needed to do strange things like running it in a *chroot* jail or store JPEG files in it.

A programmer's complaints

It was really hard to incrementally copy all the bugfixes from a bugfix branch into the main development branch, but avoid copying the workarounds. The time for a *cvs update* was roughly proportional to the number of files in total, even if only one file changed, and it would scatter conflict markers all over the source without warning. I like conflict markers, but I even more like being able to update the rest of the code without having to deal with conflicts right away.

And, oh yes, I hated those *CVS* directories cluttering up everything. My girlfriend would be surprised to hear it, but I'm actually a very orderly sort of person and those *CVS* directories always irritated me.

Perforce is what I found. As an overall system, it's bit like CVS except totally different. It has the same sort of feel, except that it's *p4 submit* instead of *cvs commit*. It has similar emacs integration, except that it's called *p4.el* instead of *vc.el*. But at the same time, different.

The server knows much, much more. All the data that CVS keeps in *CVS/** files is instead kept in a database on the perforce server, which, together with a much more efficient client-server protocol, makes a *p4 sync* incomparably faster than *cvs update*.

The branching system is different, too. In CVS, branches are a part of the file, while in perforce they are part of the directory. That's a big simplification, but it's a good one.

Perforce's access control system is also simple and effective. Most of the simple things work correctly as you would expect, for example there are atomic checkins, and *p4 sync* will never corrupt the file you're working on.

Hands on a solution

In this article, I'll describe the overall philosophy and feel of perforce, show a bit of syntax and generally blather on. If you want to play along while reading, there are versions for lots of OSes available at *ftp.perforce.com*. Just download *p4* and *p4d* and make a scratch installation in */tmp*:

```
mkdir /tmp/p4-examples
p4d -r /tmp/p4-examples &
```

This temporary *p4d* installation is limited to two users and two clients (more about this later). Once done, explain to the client that it should use the newly created server:

```
mkdir projectx
echo P4PORT=`/bin/hostname`:➥
1666 > projectx/.p4config
export P4CONFIG=.p4config
```

Whenever you use the *p4* client-program, it will read the environment variable *P4CONFIG* and look for the file specified in it. It will search the current directory for *.p4config*, then its parent, and so on, and finally use the *p4d* server specified by the *P4PORT* variable.

THE AUTHOR

Arnt Gulbrandsen has been using linux since early 1992. For many years, he worked at Trolltech, doing documentation, programming, system administration and kitchen service. He's currently having fun at a new startup.

A big and fairly complex project usually has people working on several versions of the code. When branches are used, there can be a main development branch (or more), a QA branch, a branch per release, often a vendor branch (where pristine copies of all third-party code are kept), etc.

Let's say this is a large GUI application using Qt and Coin, then the buildmaster, the person who's responsible for the overall management of the code, has both the development code, the QA/release code and the vendor code on hand:

```
$ ls ~/work/projectx
devel   rel1.0   vendor
$ ls ~/work/projectx/vendor
coin    qt
$ ls ~/work/projectx/devel
coin    src     qt
```

Inside *vendor*, there are separate directories for Qt and Coin, and in *devel*, there are all the files the developers work on – including Qt and Coin. Two versions of Qt, two versions of Coin. Why?

vendor/coin contains the vendor version of Coin, *devel/coin* the locally patched and used version. Hopefully the two are 100 percent equal (and if they are perforce will optimize away the unnecessary storage) but in principle they can be different: maybe *devel/coin* contains an older version than *vendor/coin*, or it may have some local hacks.

In perforce, *devel/coin* is a branch of *vendor/coin*. Changes are submitted on one branch and (optionally) integrated into the other.

When a new version of Coin is released, the buildmaster will untar it in *vendor/coin*, check it into perforce, and then *p4 integrate* it into *devel/coin*. The other programmers always use the version in *devel/coin*.

p4 integrate means that perforce should integrate new work from *vendor/coin* to *devel/coin*. All the vendor's new changes (the ones on *vendor/coin*) are merged into the site's own version (the ones on *devel/coin*), and the result is put in *devel/coin*.

The tricky bit is that only changes since the previous integration are considered. Luckily Perforce does most of the difficult and boring work automatically using *p4 resolve -am* (automatic merge). The buildmaster may need to deal with some conflicts, for example by editing files filled with CVS-like conflict markers. There are also other ways of doing it, such as using a user-defined merge tool for game-level files.

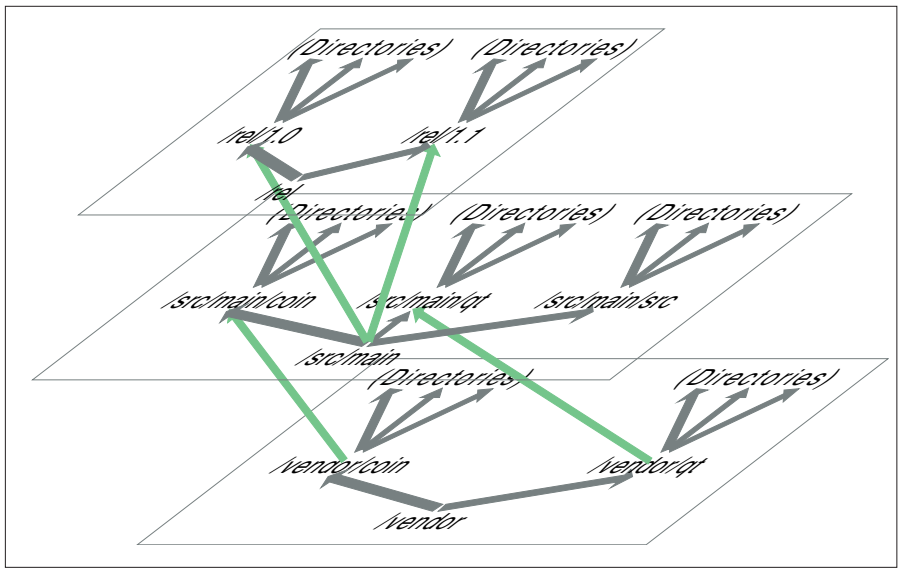

Figure 1: A 3D visualization of the example depot as it evolves at the end of this article: The planes show the release-, development- and vendor-branches with their respective subdirectories, the green arrows symbolize the integrations.

After the integrate, the buildmaster compiles and tests, and finally runs *p4 submit devel/coin/...* to submit the integration. The next time the other programmers run *p4 sync*, they're upgraded to the new Coin version.

Since the buildmaster needs so many files on hand, a client view that looks at most of the depot is necessary. The important bits are shown in Listing 1.

You can edit the current client specification using the editor defined in *EDITOR* and *p4 client*, or see it with *p4 client -o*. The variables *P4CLIENT* and/or *P4USER* specified in *.p4config* or in the environment define the perforce client and username. On Unix systems, the default values resemble the hostname and the Unix username, respectively, which is fine as long as all files to be controlled by perforce reside below one single directory tree.

This root-directory is defined in the *Root:* line of the client specification. All perforce-controlled files reside under this root. The *View:* decides which part of the depot the client can see: In Listing 1, three different parts of the depot are

Perforce and its makers

Perforce is a product of Perforce Software, a 60-employee company which makes nothing else. It's commercial, but cheap as these things go and free for open source projects, and it's properly warranted. If the software ever breaks, Perforce Software accepts responsibility for that. And there's truly excellent support. Some well-known systems using Perforce are Qt, Perl and Unreal Tournament 2003.

Homepage:	*http://www.perforce.com/*
Download:	*http://www.perforce.com/perforce/loadprog.html*
Pricing:	750 USD pr. user for the first year, 150 USD pr. user subsequently, discounts for large orders, free of charge for free-software projects
Supported plattforms (client and server):	
Linux:	i386, Alpha, Sparc, PowerPC, MIPS, IA64, S390
Other Unices:	FreeBSD, Solaris, MacOS X and almost 20 others
Other OSes:	MacOS 8.5, Windows NT/XP/2000, AmigaOS, OS/2, BeOS, VMS and roughly 10 more

Listing 1: A buildmaster's client view

```
...
# Use 'p4 help client' to see more about client views and options.
...
Root:   /home/billg/work/projectx
Options:     noallwrite noclobber compress crlf unlocked nomodtime rmdir
View:
        //depot/projectx/src/main/... devel/...
        //depot/projectx/vendor/... vendor/...
        //depot/projectx/rel/1.0/... rel1.0/...
```

made visible in three subdirectories of *~billg/work/projectx*.

The *~billg/work/projectx/vendor* directory contains everything perforce stores in *//depot/projectx/vendor*. The three dots ... mean all subdirectories and files. *~billg/work/projectx/devel* contains everything perforce stores in *//depot/projectx/src/main*, and finally *~billg/work/projectx/rel1.0* contains everything in the perforce *//depot/projectx/rel/1.0* tree. An ordinary programmers' client will usually be much simpler like in Listing 2.

Everything under control

Before going on, I'll make a bit of a detour to satisfy the control freaks among you. Yes, you can control access. If you want only the buildmaster to have write access to *//depot/projectx/vendor/* and the programmers to have write access only to *//depot/projectx/src/main*, you can put all the programmers in a group (*p4 group*) and use *p4 protect* to assign rights (see Listing 3).

The second line says that in general, everyone has *list* rights for everything. It establishes the default. The third line gives buildmaster user *billg* root access, meaning he can change people's rights and do other administrative tasks.

The last line grants *billg* the right to write (and read) to the vendor tree. (This very last line isn't really necessary: Since *billg* has superuser rights everywhere, he also has write rights to the vendor tree. It's good to have, though, in case *billg*'s superuser tasks are moved to someone else.) In lines 4–5 all members of the perforce group *developers* (hopefully including *billg*) gain the right to write to the main development tree and to read the vendor tree.

Untouchable

Back to our work scenario: While the buildmaster is upgrading Coin, one of the developers is busy debugging, and has scattered a pile of debugging *printf*s all over her *devel/coin* directory. When she runs *p4 sync* routinely to pick up other developers' changes, most of Coin is updated. But not all: If you're working on a file, *p4 sync* doesn't touch that file. Hence, the files containing debugging instrumentation aren't updated in the developer's working directory.

The developer can either throw away the debugging code with *p4 revert* or try to merge her debugging code into the new version with *p4 resolve -am*. Personally I'd try the automatic merge: perforce usually gets it right and even if I get a message talking about conflicts I can either switch to *p4 revert* or resolve the conflicts by hand if there's just a few. After revert it may also be necessary to run *p4 edit* to start editing the file again.

A little later the bug is fixed and it's time to check in the fix. A careful developer starts by running *p4 diff -du* and examining the result, a unified diff. Any files that contain only debug output can be reverted (*p4 revert*), and then it's time to sync to the latest version, check that everything works, rerun the diff, make sure that nothing looks odd … and at last run *p4 submit*.

Users of perforce integration (such as *p4.el* for *emacs*) will be typing different commands, but the principle is generally the same.

Released!

Another task one simply can't avoid during software development is to release something. I suppose we all hate it, judging by how much we put it off, but it has to be done.

With perforce, the easiest way is to make a new branch for the release, do testing and bugfixing there while most developers go on working on the main development tree, and finally integrate the fixes back to the main tree so they're also included in the next release.

The command to make a branch is called *p4 branch* and works like *p4 integrate*: You tell it where to branch from and where to. Normally both locations are directories. To make a new release, 1.1, *billg* makes a new *rel1.1* directory, runs *p4 client* and changes his view to what is shown in Listing 4.

Next, our friend *billg* needs to integrate the development source tree into *rel/1.1*. The same command as above can be used, *cd ~billg/work/projectx; p4 integrate devel/... rel1.1/...; p4 submit*

Listing 2: An ordinary member of the project

```
...
# Use 'p4 help client' to see more about client views and options.
...
Root:   /home/twee/work/projectx
...
View:
        //depot/projectx/src/main/... devel/...
```

Listing 3: Access control

```
Protections:
        list user * * //...
        super user billg * //...
        write group developers * //depot/projectx/src/main/...
        read group developers * //depot/projectx/vendor/...
        write user billg * //depot/projectx/vendor/...
```

rel1.1/... for example. The *p4* commands could also use the depot paths (*//depot/projectx/src/main/...* and so on) but I find it easier and more natural to use the paths that are visible in my file system.

During the release phase, part of the team works on the *rel/1.1* tree and submit some fixes there. Meanwhile, *billg* makes sure to integrate all the work back to the main tree so that the main team gets the bugfixes too. The command to do it is *p4 rel1.1/... integrate devel/...* – the same as above, except that the direction of integration has been reversed. In case of conflicts, a *p4 resolve -am* or similar may be necessary to resolve them.

As before, *integrate* will only consider those changes that are new since the previous integration. So, if there's a temporary workaround on the release bugfix only meant for this release, it's enough to tell *p4 integrate* "no, I don't want to integrate that one" once, and perforce will remember that.

Finally, when the release is built, one can set a label on the directory tree using *p4 label*, so that later it's easy to e.g. make a diff between release 1.1 and 1.1.1 (*p4 diff or diff2*), or make another branch in case there's a security problem and an 1.1.0.1 must be released.

Listing 4: The view of a new release

```
View:
     //depot/projectx/src/main/... devel/...
     //depot/projectx/vendor/... vendor/...
     //depot/projectx/rel/1.1/... rel1.1/...
```

More features

There's a lot more features. I'll mention three biggies:

- Dispersed teams can use *p4p*, the perforce proxy, to get more speed across the network. With the proxy, data is cached extensively. The proxy runs at each site, and there's one main perforce server. For example, during Unreal Tournament 2003 development five different companies around the world all accessed the same server.
- Anyone working with graphics or other binary files can use the file type support, e.g. to turn off *Id* expansion on XPM files. In a 200k XPM file, there's about 0.04 percent chance of finding the four bytes *0x24 0x49 0x64 0x24* that code for *Id*. High enough that sometimes XPM files are corrupted, unless the version control system gets it right.
- There's a rudimentary job tracking system built into perforce, and an integration system called P4DTI. P4DTI is much bigger, it's really worth an article all by itself. It integrates perforce with bugzilla and some other similar systems.

Now, at the end, if you like what you've read, you may want to install perforce for real and try it out. If you want to use it for a free software project with more than two people, ask *support@perforce.com* for a free license. And see if you can make your *p4d* live longer than mine: 455 days before a power cut spoilt the fun. ■