

VERSION CONTROL SYSTEM

Setting up CVS

PART 2 Jono Bacon sets up a CVS server and shows you how to make good use of it.

Last issue we looked at the CVS client program to connect to a CVS server, check in some code, checkout code and browse log entries. This is all useful if someone provides a CVS server for you, otherwise you will need to set up your own – but it's not that hard, and this month we will investigate the processes involved in doing this.

Setting up a CVS server is not a difficult procedure to administer, but there is a distinct set of processes in which you can easily make mistakes that can be difficult to solve later on. We will look at these processes and how to avoid the pitfalls.

A quick recap

Before we get started, let's have a quick recap of what CVS is and what we can use it for; this will be of particular use for those people who missed last month's issue.

Imagine you are a maintainer of a project. You are in control of the entire project and how it runs. Let us now assume that after a recent announcement you made of the project, you get an email from two developers who would like to contribute some code to your project. This is great news and can increase the development pace of your application. The problem you have now is how you can manage all three developers working on the same project. One option could be for the other developers to send you patches (small files containing new code for a project), which you can then apply to your source code. This is fine until you realise that after every patch you will need to release the source code so the developers are up to date. Now, let us assume that developer 1 writes a patch and developer 2 writes a

different patch that uses some of the same source files. These two patches are fine by themselves but together there will be a conflict. The conflict would have to be resolved by you, and it may be in a part of the code you are not very familiar with. As you can see, this is a very messy situation for three developers – now imagine this for a huge project with hundreds of developers from around the world. The situation can get unbearable. This is where the wonderful world of CVS comes in.

CVS is a system that will manage the code you and your developers work on and also how you and the developers interact and change the code. CVS will not only manage the code from all the developers, but will also attempt to resolve conflicts. CVS also has the ability to revert to previously written code right back to your first commits. CVS is in general a very flexible system, and numerous projects such as *KDE*, *GNOME* and *XFree86* use it.

Installing the CVS server is a straightforward process as it comes with the CVS distribution. To install CVS on your system, please see the *Installing CVS* box.

Creating the repository

When CVS is installed on the system, we can now create the CVS repository in which all the code will reside that the developers check in and out. The first thing to decide is where you want your repository to live. To set up /cvs I first created the directory with:

```
mkdir /cvs
```

The command needs to be issued as root as it created a new root level directory. To create the repository I then issued:

```
cvs -d /cvs init
```

If the repository was created successfully, the command will take you back to prompt silently. We can now take a look at what has been created in the directory.

A quick **ls -al** shows us that we have a **CVSROOT** directory that has been created. In this directory we have the following:

```
checkoutlist  cvswrappers,v  modules,v  taginfo
checkoutlist,v  editinfo  notify  taginfo,v
commitinfo  editinfo,v  notify,v  val-tags
commitinfo,v  history  verifymsg
config  loginfo  verifymsg,v
config,v  loginfo,v  rcsinfo
cvswrappers  modules  rcsinfo,v
```

Notice how there are a number of files; some with the same name, and **,v** appended. These **,v** files store the version information. The normal files (without the **,v**) are actual config files. We will take a look at some of these configuration files later.

It is a good idea to create a system account dedicated to CVS:

```
adduser cvs
```

Then change the ownership of the repository to this account.

Configuring the password server

It is all very well having a CVS repository, but it isn't much use without being able to connect to it securely; this is the function of the password server.

When **:pserver** is used in the **CVSROOT** of the client connecting to the machine, the client will make itself known through port 2401. The CVS server does not listen to this port actively, but *inetd* will monitor the port and start the CVS server if



CVS Web in action – it's even easier with a web-based interface.

a connection is made. To enable this to work, we first need to edit `/etc/services` and `/etc/inetd.conf`. First add the following to `/etc/services`:

```
cvspserver 2401/tcp
```

Then add the following to `/etc/inetd.conf`:

```
cvspserver stream tcp nowait root /usr/local/bin/cvs
cvs —allow-root=cvs pserver
```

Now restart `inetd` and you should be ready.

Setting up CVS users

Although we have configured the server to accept connections from CVS clients, we need to add CVS accounts so a user can connect with a password and not compromise system security. To set this up we will create a file in `/cvs/CVSROOT` called `passwd`. The format is simple, and is in the following schema:

```
[user]:[password]:[system-account]
```

The different parts are:

user – The name of the user connecting to the server. *E.g.* `jono`.
password – An encrypted password for the user. You can copy existing encrypted passwords from `/etc/passwd` or `/etc/shadow`.
system-account – If this optional part is used, this system account name will be used for CVS operations. If `system-account` is not specified, the user account will be used. We could use the 'cvs' system account we created earlier which owns the repository. Here is an example of mine (with the password changed):

```
anonymous:cvs
jono:FedrR$%fDGGHtFGDF$4rDDDFDF343:cvs
```

As you can see, I have two accounts – both use the system user 'cvs', and the `jono` CVS account has an encrypted password, whereas the anonymous user has no password; this would mean that the anonymous user would press enter when prompted for a password.

Anonymous CVS access is a special case as you only want the user to be able to read data from the repository and not be able to commit back to it. This can be achieved by just creating an anonymous account and restricting the permissions for that user, but this is a bit cumbersome, and CVS offers a simple system of giving read/write permissions.

To set up 'anonymous' for read only access, add 'anonymous' to the `/cvs/CVSROOT/readers` file. Each line in the file has a username for those users who can only read data from the repository. There is also a `/cvs/CVSROOT/writers` file in which people with write access can be added. All those users who are not in `/cvs/CVSROOT/writers` are assumed to have read access.

Testing the CVS

To test our setup, load up a terminal and set the **CVSROOT**:

```
export CVSROOT=:pserver:jono@127.0.0.1:/cvs
```

I covered the setting of the **CVSROOT** in the last issue, but here is a quick rundown of each part:

:pserver – This specifies that the password server is used to authenticate access.

jono@127.0.0.1 – This specifies the user and host. The user is 'jono' at the IP 127.0.0.1, which is the local loopback IP address for your machine. You can also set a domain like: `jono@cvs.thismachine.org`.

/cvs – This is the system path of the repository.

Now this is set, log in with:

```
cvs login
```

and specify any password that you may have set for your user. If you are returned to the prompt, it is likely a connection was made. To test we can check out the **CVSROOT** directory from the CVS (each directory in the repository is classed as a module). Go to a directory where you will store your sources and issue:

```
Shell - Konsole
Session Edit View Settings Help
-r--r--r-- 1 cvsuser src 693 Jan 11 22:56 checkoutlist,v
-r--r--r-- 1 cvsuser src 760 Jan 16 22:49 commitinfo
-r--r--r-- 1 cvsuser src 960 Jan 11 22:56 commitinfo,v
-r--r--r-- 1 cvsuser src 527 Jan 16 22:49 config
-r--r--r-- 1 cvsuser src 727 Jan 11 22:56 config,v
-r--r--r-- 1 cvsuser src 753 Jan 16 22:49 cvswrappers
-r--r--r-- 1 cvsuser src 953 Jan 11 22:56 cvswrappers,v
-r--r--r-- 1 cvsuser src 1025 Jan 16 22:49 editinfo
-r--r--r-- 1 cvsuser src 1225 Jan 11 22:56 editinfo,v
-rw-rw-r-- 1 cvsuser src 7509 Feb 6 14:18 history
-r--r--r-- 1 cvsuser src 1141 Jan 16 22:49 loginfo
-r--r--r-- 1 cvsuser src 1341 Jan 11 22:56 loginfo,v
-r--r--r-- 1 cvsuser src 1173 Jan 16 22:49 modules
-r--r--r-- 1 cvsuser src 2052 Jan 16 22:49 modules,v
-r--r--r-- 1 cvsuser src 564 Jan 16 22:49 notify
-r--r--r-- 1 cvsuser src 764 Jan 11 22:56 notify,v
-rw-r--r-- 1 cvsuser src 67 Jan 16 21:41 passwd
-rw-r--r-- 1 cvsuser src 59 Jan 14 14:05 passwd~
-r--r--r-- 1 cvsuser src 649 Jan 16 22:49 rcsinfo
-r--r--r-- 1 cvsuser src 849 Jan 11 22:56 rcsinfo,v
-r--r--r-- 1 cvsuser src 879 Jan 16 22:49 taginfo
-r--r--r-- 1 cvsuser src 1079 Jan 11 22:56 taginfo,v
-rw-rw-r-- 1 cvsuser src 8 Jan 14 14:50 val-tags
-r--r--r-- 1 cvsuser src 1026 Jan 16 22:49 verifymsg
-r--r--r-- 1 cvsuser src 1226 Jan 11 22:56 verifymsg,v
jono@forge: /cvs/CVSROOT$
```

Files from the **CVSROOT** module in the repository.

Different uses for CVS

It's not just for programming applications

CVS is often used in any place where source code needs to be managed in some way - this has in turn made CVS appear in places you would not expect to see it. This box gives some information on different uses of CVS that you may find useful:

Application Development

This is probably the most used area in which CVS is used. CVS is used to manage source code which developers commit and CVS will ignore object code, backup files and other unneeded files. CVS is at home in this situation with many projects using it for this kind of development.

Web Development

Many people don't realise how CVS can be used in web development; CVS is a handy tool which can make web development easier and more secure. A typical example is if you are writing a site in PHP and using MySQL.

All the pages can be kept in the repository and then the developer can set the *Apache* webroot to point to the working directory of the checked out code. This gives the developer the ability to work and test from the checked out directory and commit code straight back to the CVS server. This code on the CVS server could then be checked out to a working directory which is the webroot of the live site. This checkout can be done using *cron*, so the whole procedure is automated. I myself have developed a number of sites using this method and it works well.

Document Development

Many people who are using tools such as *DocBook* have been using CVS to manage updates to the documentation, and to create branches in documentation for different versions of the application that the documentation covers.

```
cvs co CVSROOT
```

If the files are checked out, you know the connection is working.

Administering the repository

Now that the server is set up, we can begin to configure the server and modules. To administer the server, we use CVS itself to handle the configuration files in the **CVSROOT** module. So any changes we wish to make should be made in the local copy of the **CVSROOT** and then checked back into the repository.

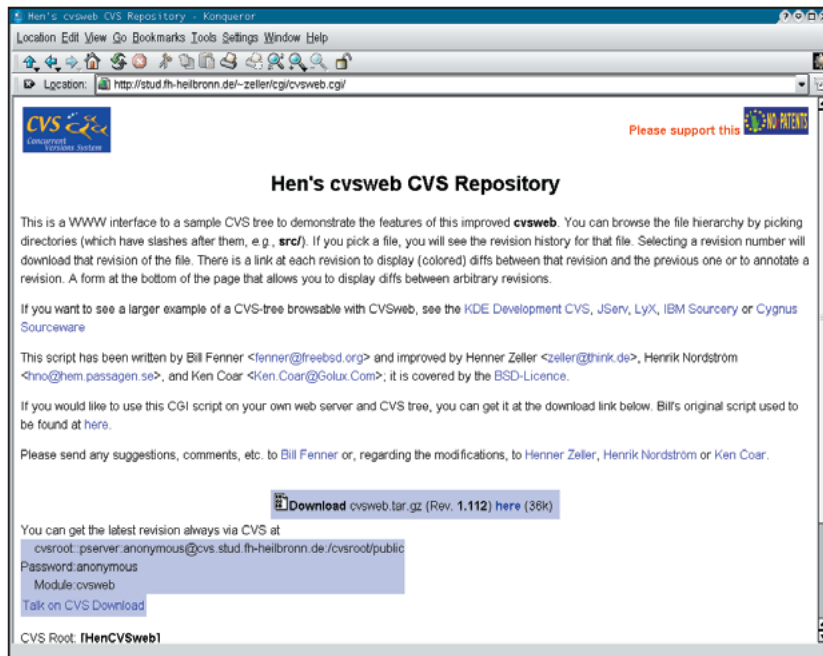
The first thing we should look at is setting up the 'modules' file. This file contains a list of modules in the CVS, and is often used by clients to see what is the repository. The format of the file is this:

```
[module-name] [path-to-module]
```

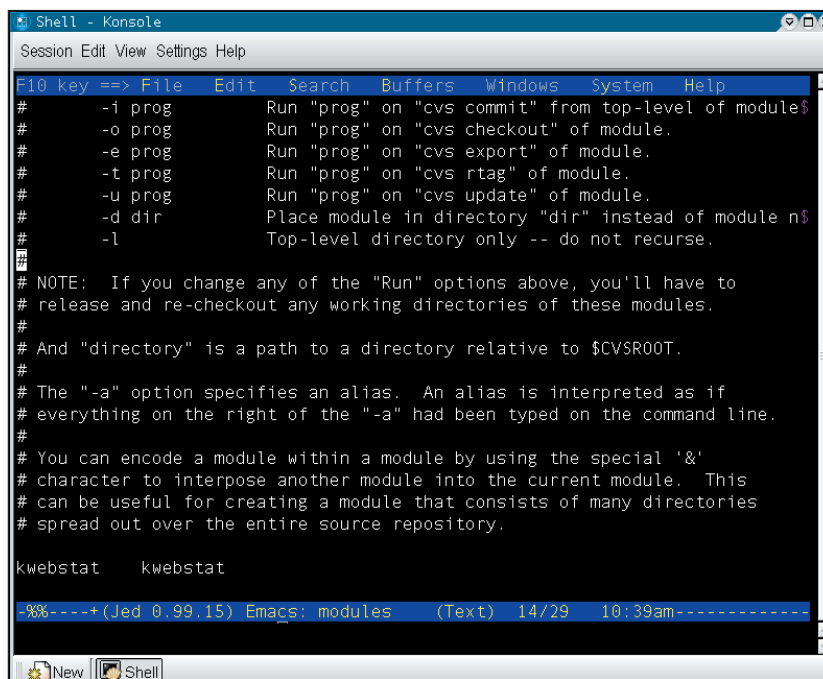
The **module-name** is the name of the particular module in the repository, and the **path-to-module** is the path within the



LinuxFormatTutorialCVS



The CVS Web homepage. Go here, read – two steps to more CVS knowledge.



The 'modules' file – often used by clients to check the contents of the repository.

CVS Web

Viewing your CVS with CVS Web

CVS Web is a CGI script which will create a web based interface to your CVS. It will allow people to view files and log entries; generate diffs and more. It is a useful tool that can help with keeping a track on what is being committed to your server.

To install CVS Web, go to <http://www.cvsweb.org/> and download the tarball. You can then install it:

1 Copy the `cvsweb.conf` to the Apache \$ServerRoot/conf. Edit the file where needed,

but the most important part is the:

'Local-CVS' => '/cvs'

(if /cvs is your repository)

The other parts of the file adjust the look and feel of CVS Web as well as some other settings.

2 Copy the `cvsweb.cgi` to the cgi-bin of your server. Edit the \$config variable to point to your configuration file.

3 Restart Apache with:
apachectl restart

repository to the module. As most people call their modules the same name as the directory, you frequently see the 'module' file entries as:

thismodule	thismodule
thatmodule	thatmodule

Once you have set the modules file up, the server is pretty much ready for use.

Adding a module

One of the first things you will want to do with your brand new CVS server is add some modules to the repository that people can start working on. This is a simple procedure and involves a single command – but before we actually import the project, we need to prepare it, and remove any unwanted files and junk that we don't want in the repository.

The only things that go in the repository should be files which are to be distributed with your project. This means that the following types of files should *all* be removed.

~file – This is often a backup left by a text editor. Check the new version of the file and make sure you are fine to delete the backup and then remove it.

Makefile.in – These are generated files from `Makefile.am` and `configure.in.in`. These generated `configure.in` files (as well as other generated files) should be removed.

file.o – This is an object code file with compiled object code in it. Remove these.

Look out for generated code from tools such as *Qt Designer* – keep the file which is used to generate the code, but not the generated code. CVS will remove some junk for you automatically.

With most projects (particularly those using *AutoConf/AutoMake*) you can remove all of this unwanted rubbish with:

make distclean

Note that you must do a **distclean** and not just a **clean** – **clean** is not thorough enough.

When you have removed all the unwanted tat from your project directory, you are ready to import it into your repository (assuming you have write permissions). Go to the directory where the project is:

cd myproject

Then issue the following command:

cvs import myproject 1_0 start

Let's just take a look at what the different parts of this command are:

import – This is the command for putting projects in the repository so CVS looks after the versioning for you.

myproject – This is where in the CVS you want it putting. By specifying **myproject** we are asking CVS to put it in the rootlevel part of the repository (`/cvs/myproject`). We could ask CVS to put it inside another module name (e.g for it to go in 'thatmodule' we would use `thatmodule/thismodule`).

1_0 – This is the vendor tag for that branch in the CVS. A branch is like a specifically developed version of the application. Eg. **1_0** and **1_1** are two separate branches, and hence separate changes to the same source code.

start – This is the release tag which is used to identify files for each leaf that is created by the **import** command. We can use **start** to indicate the start of development.

When you issue the command, CVS will tell you if there were any conflicts when the project was added to the repository.

Security Considerations

Security is an important consideration when working with CVS modules. You must be careful not to commit into a public

CVS passwords and other sensitive data that may compromise your system.

A typical example with this is when working on a web site. You may be using *MySQL* and *PHP*, and to connect to the *MySQL* database you will need to specify a username, password and host. Although this may seem a security risk to commit this information to the *CVS*, there is a simple workaround.

All you need to do is to create a file (which will not be in the *CVS*) which has some variables which contain the right data. For example:

```
<?php
    $DB_User = "myuser";
    $DB_Pass = "thepass1654";
    $DB_Host = "mysecrethost.org";
?>
```

You can then use `include()` (or whatever is similar for your language) to include this non-repository local file, and when you need to refer to the information, use the variable instead. This also gives the ability to change the information once, and it is reflected across the site (similar to a stylesheet).

Branches

Branches are a useful feature in *CVS* that allows you to fork development into two separate directions where each version does not affect the other. How is this useful you may ask? Let me explain with an example.

Let us assume you are working on an application called *EasyCVS*. *EasyCVS* was created initially, had the junk removed and was then imported into the repository. *EasyCVS* then undergoes some intense development as it builds up towards the *1.0* release. When the *1.0* point is reached, the code is packaged and released.

Now, as a developer you are waiting to add some snazzy new features in the *2.0* release of *EasyCVS*. You begin eagerly working on the new version and you add some major new features which require changing a lot of the code, and some elements of redesigning you classes. This is all going well until you receive an email from someone regarding a fairly serious bug in your application that really needs fixing. You have a problem here now – you have already started writing major new code which involves structure redesigns, but you know that the bugfix code will not work with the new additions. Essentially you are stuck now – do you discard the new additions in favour of the original release that is bugfixed, or do you ditch bugfixes and just go with the new features, taking time to fix the reported bug as you code.

With *CVS*, you don't need to make such a decision. *CVS* gives you the ability to branch your code into a *1_0* and *2_0* branch. When you make the branch, the code is essentially the same, but you can then make the bugfix commits to the *1_0* branch, and then add the new features to the *2_0* branch. This gives you the ability to release updates to the *1_0* release if you need to. The clever thing about these branches is that *CVS* records all of this information in a single module – there are no additional modules for each branch – this makes no difference to client, but to the administrator it saves having lots of extra modules.

To get started with branches, you can create a branch in the *EasyCVS* module by issuing the following in the working directory:

```
cvs tag -b 1_0
```

This creates a branch of the working directory called *1_0*. You could also issue:

```
cvs tag -b 2_0
```

This would create the second branch based on the working directory also. Remember that these branches are *not* made in the working directory, they are made on the *CVS* server, so you

Installing CVS

First step to a CVS server

Installing *CVS* is a fairly straightforward process, and is similar to the installation of other packages and source for Linux.

The first thing to do is to download the relevant packages. Many Linux distributions have prebuilt packages available, which can be downloaded and installed using the package management tools in the distribution. Many distributions also provide the packages for download on their FTP sites. Check your distribution for specific details.

Another site where you can gather *CVS* resources and downloads is <http://www.cvshome.org/>.

When you have got the relevant packages for your system, you can install them using the following instructions:

Compiling the Source

If you got a source package, you need to first of all unzip and untar it into a directory:

```
tar xzf cvspackage.tgz
```

You can then read the *INSTALL* file on specific details of compiling the package, essentially:

```
./configure
make
make install
```

Installing an RPM

If you downloaded a prebuilt RPM package you can install it with:

```
rpm -i cvspackage.rpm
```

Installing from Debian

Using the Debian distribution you install with:

```
apt-get install cvs
```

```
Shell - Konsole
Session Edit View Settings Help
F10 key ==> File Edit Search Buffers Windows System Help
head 1.13;
access;
symbols
    start:1.1.1.1 jono:1.1.1;
locks; strict;
comment @// @;
-
1.13
date 2002.02.06.14.18.40; author jono; state Exp;
branches;
next 1.12;
1.12
date 2002.02.06.00.39.14; author jono; state Exp;
branches;
next 1.11;
1.11
date 2002.02.05.16.45.53; author jono; state Exp;
branches;
next 1.10;
%%----+(Jed 0.99.15) Emacs: kwebstat.cpp,v (Text) 1/572 10:41am-----
New Shell
```

The internals of a ,v file - where the version information is stored.

therefore need to check out the right branch:

```
cvs co -r 1_0 EasyCVS
```

This would check out the *1_0* branch, and all commits would be made to that branch. You are now in a position to maintain the two separate development trees.

Conclusion

CVS is a big topic; there is a lot that can be done with it, and I have tried to cover as much as I can here. There are, of course, many other facilities in *CVS* of which you can make use, but the topics I have covered are the major ones you should learn. I suggest visiting the *CVS* website at <http://www.cvshome.org/> and reading the manual and documentation if you need any further help.

CVS is something that cannot replace proper management of your team and code, but can certainly assist in it – and, with the increasing number of projects using *CVS*, knowledge of it is a prerequisite in many areas. Have fun! [LXF](#)