International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 04 Issue: 04 | Apr -2017 www.irjet.net p-ISSN: 2395-0072

CONCURRENT VERSION SYSTEM FOR CONFIGURING THE PROJECT

K V SAI KRISHNA, MADHUMATHI R, DEEPA N

[1] *Sai Krishna, Student, School of Information Technology, VIT University, Tamilnadu, India*
[1] *Madhumathi, Student, School of Information Technology, VIT University, Tamilnadu, India*
[1] *Deepa N, Assistant Professor, School of Information Technology, VIT University, Tamilnadu, India*

Abstract - Concurrent Versions System (CVS) is a tool or program that lets a code developer to store and retrieve different versions of source code. It also lets a team of developers share different versions of files in a common repository of files. This kind of program is sometimes known as a version control system. CVS lets groups of people work simultaneously on groups of files.

Key Words: Repository, Version, Concurrent version systems, Configuration, Directories, etc …

1. INTRODUCTION

Software product undergoes many process, steps and strategies to deliver the right outcome to the customer. The main burden of the keeping the software updated starts in the change management process. The customer requirements changes over time, so new versions of the software is developed as the customer wants to modify the features of the existing product. The task of tracking and controlling changes in the software after delivery is called software configuration management. SCM practices include revision control and version control. The main task of version control system is to track or record all the changes to a file or set of files over time so that you can recall specific version later.

CVS works by holding a central `repository' of the most recent version of the files. We can at any time create a personal copy of these files by `checking out' the files from the repository into one of our local directories. If at a later date newer versions of the files are put in the repository, we can `update' our copy. We can edit our copy of the files freely. If new versions of the files have been put in the repository, then making an update merges the changes in the central copy into our copy. When we are satisfied with the changes we have made in our local copy of the files, we can `commit' them into the central repository. When we are finally done with our personal copy of the files, we can `release' them and then remove them.

Concurrent version system was developed in the UNIX operating system environment and is available in both open source and commercial versions. CVS is a popular tool for developers working on Linux and other UNIX-based systems.

1.1 Basic Units of CVS

Following points contribute the basic units of cvs:

1. Repository: The master copy or server where CVS stores a project's full revision history. Each project has exactly one repository
2. Working copy: The copy in which you actually make changes to a project.
3. Check out: To request a working copy from the repository.
4. Commit: To send changes from your working copy into the central repository. Also known as check-in.
5. Log message: A comment that we attach to a revision when we commit it, describing the changes.
6. Release: Signs out with CVS servers.
7. Conflict: Situation when one or more developers may try to commit the changes to be the same region of same file.
8. Update: To bring the other change from repositories into your present working copy and also to show if you're working copy can have any uncommitted changes.

1.2 Activities on CVS:

Following are the activities on the cvs:

- Importing Assets
- Creating a Repository
- Viewing changes
- Checking out a working copy
- Working with previous versions
- Committing changes

2. CVS Commands

General Syntax:

Cvs *cvs-options* subcommand *subcommand-options*

Subcommand option is nothing but the function that you are asking the cvs to do. To see the syntax of a given command.

Cvs -H *subcommand*

Cvs --help says you how to get list of files what the different subcommands are.

Checking of Files Out:

When you are working with CVS there are two copies of files that you wanted to be very concerned:

1. Local copies, which are visible to you.
2. The repository, which is visible to everyone.

Before starting you wanted to check out the local copy of repository files. Here comes an example of it:

```
cvs checkout: Updating mymodule1
$ cvs checkout mymodule1
U mymodule1/file1

$ ls
total 1
1 mymodule1/
```

"mymodule1" is a module in repository. Checking out the modules places a local copy in the present directory. Changes are made to the files here, then put back them to the repository.

Modules are directories underneath $CVSROOT. In other words:

```
$ ls $CVSROOT
total 2
1 CVSROOT/     1 mymodule/
```

CVSROOT contains administrative and configuration files used by CVS.

Editing Files:

Editing files is an easy task - once you found a local copies, just edit it. None of your changes will be seen or shown to other users until you have committed them.

Starting over is easy one even if you messed up the local copy of the file. We can delete the file and get the fresh copy from the repository using the command cvs update.

Refreshing Local Copies:

In case you're working with a team, remember that your team members are also making changes. Periodically, you'll want to update your working copies. This is done with the **cvs update** command.

```
$ cvs update -P –d
cvs update: Updating .
U myfile
```

Above, we can see that someone has modified the myfile, and the copy in the current directory was out of date; cvs updates file1 to the current version. The updating is optional. -P "prunes" directories which are empty, and -d tells cvs to include any new directories which aren't in your local workspace. Once you have a local copy, **cvs checkout** and **cvs update -d** are more or less equivalent functions.

Update also takes arguments, if you are updating a or specific or files specific directory with a directory. If arguments are not given, cvs recursively updates the directory tree rooted at the current directory.

Seeing Changes:

To see if someone else has changed a particular file, use **cvs status**.

```
$ cvs status file1
===================================================================
File: file1        Status: Up-to-date

   Repository revision:    1.2
/home/srevilak/c/mymodule/file1,v
   Working revision:    1.2    Thu Apr 10 14:49:15
2017

   Sticky Date:       (none)
   Sticky Tag:        (none)
   Sticky Options:    (none)
```

"Up-to-date" means that the file is on date or current.

To have a glance look we can go for this commands:

```
$ cvs -n update
```

The -n option says about cvs "don't change the disk". Where local files doesn't match the repository copy, you'll have to see that the name of the file and a status..

Committing Changes:

Okay, we have done some work and we are satisfied with the output. To incorporate our changes into the repository, use the command **cvs commit**.

$ cvs commit *filename*

CVS invokes CVSEDITOR so, you can make few comments. Once done you quit from the editor, the changes which you have made will be put back into the repository.

Adding Files and Directories:

Directories and files are added with the command called **cvs add**.

To add the binary file

```
cvs add -kb newfile
cvs commit newfile
```

To add the file

```
# create the file, edit it
cvs add newfile
cvs commit newfile
```

To add a directory:

```
mkdirnewdir
cvs add newdir
```

-kb indicates cvs that file is a binary file.

Merging Revisions:

Hypothetical Situations: you take a copy of Myfile.java home, and you have worked on it. Meanwhile, other developers may commit changes to the file. The dilemma – you would like to mix what you have done, but your copies of files are now out dated. You are always not wanted to undo the work that others have already done. Here's a way to deal with this kind of situations.

1. Run cvs update to refresh your repository copy.
2. Find out what revision your copy of the file is based on. That will be the revision number in the Id or $Revision$ tags. If you can't determine the revision, this approach won't work, and you'll need to do a manual merge.
3. Run cvs log MyFile.java to get the revision number of the copy that you just checked out of the repository.

For the sake of demonstration, let's say the copy of MyFile.java that you were working on at your workstation is revision 1.6, and the current repository version is 1.10.

Copy the MyFile.java that you have worked on, at workstation to your checkout directory. We now have the following arrangements:

- You are missing few differences from 1.7 - 1.10. (Note: this is the reason you do not want to commit a file yet. Doing so can remove anything done between 1.7 and 1.10).
- The original file in your checkout area is revision 1.6 + changes.
- The version of repository is 1.10, which you have just checked out. As far as cvs is concerned, your local copy will be up to date.

To pick up some modifications made from 1.7 - 1.10, you have to merge:

cvs update -j 1.7 -j 1.10 MyFile.java

In cvs-speak, this is nothing but "taking the changes from revision of 1.7 through the revision 1.10, and then you can apply them to the local copy of the file." Assuming that there wasn't merging conflicts, checking the results:

cvs diff -w MyFile.java

Make sure it compiles, then commit.

If things aren't well, you will be needed to examine the results and you have to resolve if any conflicts arise that happens as result of the merging.

Resolving Conflicts:

```
$ cvs commit foo.java
cvs commit: Up-to-date check failed for `foo.java'
cvs [commit aborted]: correct above errors first!
```

Above, you have made changes to foo.java file, but someone has already committed a new version to repository. Before you commit the file, you'll have to update your working copy.

If you and other developers were working on different files, cvs is gives best about merging them together. That might be seen that the last set of modifications lines were in 75-100,

the changes you have made are in lines of 12-36. In this case, the file is patched and your work is still unaffected.

If two of you have changed the same area of the files it is possible to have conflicts:

```
$ cvs update foo.java
RCS file: /home/srevilak/c/mymodule/foo.java,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into foo.java
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in foo.java
C foo.java
```

To fix the merge, two things have to be done.

1. A pre-merge copy of the file has made.
   2. $ ls -a .#*
   3. 1 .#foo.java.1.1

2. Cvs inserted a conflict marker in your working copy.
   3. <<<<<<< foo.java
   4. static final int MYCONST = 3;
   5. =======
   6. static final int MYCONST = 2;
   >>>>>>> 1.2

The conflicts lie between rows of less than and greater than signs. Now you should do is to decide what version will be right and then remove the conflict markers, and commit file.

Table -1: Finding difference between versions

cvs diff filename	Shows differences between your local copy and the repository version that *filename* was based on.
cvs diff -r 1.3 filename	Shows differences between your local copy of the current version of *filename* and the version 1.3 of *filename*.
cvs diff -r 1.3 -r 1.4 filename	Shows differences between versions 1.3 and 1.4.
cvs log filename	Show the commit log for *filename*.
cvs annotate filename	Shows each line of *filename*, the name of the person who added it and prefixed with the version number where the line was added. Useful for checking who made a particular set of changes.

Backing out a Bad Commit:

Let us suppose think that you committed the file, but this made a result of breaking something badly. This is how to undo commit:

1. Take the version number before the commit. Now this will be lower than your current version. Let the old version be 1.4.
2. Take version number after the commit. You can also use an Id tag in the file. And the new version is 1.5.

Now:

```
cvs update -j 1.5 -j 1.4 filename
cvs commit filename
```

The above mentioned is an example of **merge**. You have requested cvs to take the difference between 1.5 and 1.4version and then apply to your current working copy.

Deleting Files:

To delete files, use the command **cvs delete**:

```
rm filename      # remove working copy first
cvs delete filename
cvs commit
```

"lazy" system is used for deletion; delete only changes the way that are stored in the repository. It is still possible to check out revisions that existed before the file was deleted or to undelete few files. The file will no longer be when you do updates or checkouts.

Cause of this, we cannot delete a directory entirely. One can use the -P flag with cvs checkout and cvs update to prevent the empty directories from the being retrieved files.

Manual states: “the usual way for getting reserved checkouts with CVS is called cvs admin -l command."

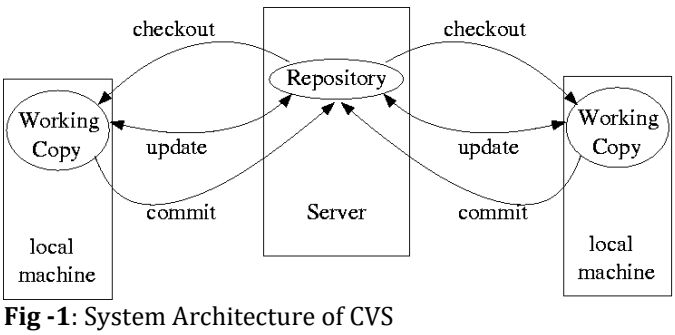

Fig -1: System Architecture of CVS

3. CONCLUSIONS

Conflicts arises when one or more developers trying commit the changes to the same region which belongs to the same file. CVS is not having situation of "locking" file for the occurred changes. If you wanted to start editing of a file, you just start editing it simply. There is another option which is optional edit-notification which says you that when someone on other side trying to editing the same file. This mechanism does not prevents you from the editing of files, but it gives you an opportunity for finding out who else are editing the file and talk with them so that you can figure out whether you are interacting on each other's file.

CVS do have concept of locking, even though it's not default.

REFERENCES

[1] *Barrett, Arthur* . "Anonymous or Developer checkout with TortoiseCVS".

[2] Charles D. Cranor, Theo de Raadt "Opening The Source Repository With Anonymous CVS, USENIX 1999" (1999).

[3] Karl Franz Fogel, Moshe Bar. Open Source Development with CVS.

[4] Version Management with CVS – manual for CVS 1.12.13, by Per Cederqvist et al.

[5] Bjoern-Elmar Macek, "Profile mining in CVS logs and face-to-face contacts for recommending software developers", 2011 IEEE third international conference on privacy.