SLAC-PUB-5923
September 1992
(E/I)

UNIX Code Management and Distribution*

Terry Hung and Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University, Stanford CA 94309, U.S.A.

We describe a code management and distribution system based on tools freely available for the UNIX systems. At the master site, version control is managed with CVS, which is a layer on top of RCS, and distribution is done via NFS mounted file systems. At remote sites, small modifications to CVS provide for interactive transactions with the CVS system at the master site such that remote developers are true peers in the code development process.

1. Introduction

A source code management system is a set of tools keeping track of various versions and configurations of programs or applications. For the UNIX operating system, the most frequently used systems are SCCS[1] and RCS[2]. Of the two, SCCS is more widely known and is part of the vendor's software release on most workstations. We have chosen to use RCS because it is more modern, is easier to use, has some additional important features, and is unencumbered by an AT&T source license. At SLAC, RCS has been used for projects for over three years.

Using SCCS or RCS alone, or with a thin layer of shell scripts on top, is sufficient for small projects. However, if the project is large, *i.e.* contains many subdirectories, then the *thin* layer becomes rather thick. For this reason, we have chosen to use CVS[3], which is a layer on top of RCS. It handles the multiple subdirectory problem in an elegant way, and adds additional features which will be described below. CVS has been in use at SLAC since July of this year.

The goal of a code distribution system is to provide source code access to developers at sites remote from the master source code repository and a means for code developed at the remote site to be contributed back to the master site. By use of native UNIX commands, *rsh* and *rdist*, in a set of shell scripts, a remote developer can perform code management transactions at the master site in real-time over the Internet. The syntax used at remote sites is almost identical to that used at the master site. Thus, with this system, remote developers are truly peers in the code development process.

Our initial code distribution system, CDS, was used for over one year. It was then re-written to support CVS, and called *remote* CVS. It has been operational since September of this year. Since these native UNIX tools are not widely known to the High Energy Physics (HEP) community, this paper will also introduce some of their concepts where appropriate.

2. Revision Control System (RCS)

The Revision Control system, RCS, maintains different versions of source code in a single file by automatically generating the differences between versions when new versions are checked in. The files that RCS generates are clear ASCII text with header and footer information wrapped around the latest version of the file. The differences between versions are stored as deltas to go back to an older version, thus access to the latest version is always fast.

When a developer checks out a file from RCS for editing, he issues the check out command with the *lock* option and the user is given a writable version. Without using the lock option, the user is given a read-only version of the file. When a file is locked, other developers can not

* Work supported by Department of Energy, contract DE-AC03-76SF00515.
Contributed paper to the Conference on Computing in High Energy Physics, Annecy, France, September 21-25, 1992.

check out the same file with the lock option, thus preventing overlapping changes. When necessary, the lock can be broken and mail is sent to the developer who had the original lock.

Normally RCS revision are numbered sequentially, *e.g.* 1.1, 1.2, *etc*. One can check out a previous version using this revision number, by date, and other means. An important means is by symbolic tag. A symbolic tag can be applied to all files of a directory and is used to mark revisions that belong to a certain release level. At a later date, after development has continued, one can check out the revision of each file that belonged to a release level by using the symbolic tag.

The symbolic tags are used in conjunction with revision branches in the following manner. Suppose that a release was made and development towards the next release has continued, but that a serious bug was found in the release which must be fixed. One can checkout the revisions corresponding to the release, fix the bug, then check in the fixes as a branch to that revision level. Also, at some later time, one can merge any branches into the main development trunk to ensure that any release patches get into the next release.

Overall, RCS provides a quite flexible and easy to use code management system. It is freely available from the Free Software Foundation, runs on most UNIX platforms (even on MS-DOS), and development work continues on it. It has a large following of professionals in well known software companies and could be said to be an industry standard.

RCS is not without deficiencies, however, which should also be mentioned here. Although RCS documentation mentions *configuration management* it does not handle configuration of source code to take into account operating system differences. Under UNIX, such dependencies are usually handled by a preprocessor to the compiler, such as *cpp.* In HEP, similar preprocessors have been developed for FORTRAN[5][6]. Also RCS works on only one directory of files at a time. If a project requires multiple hierarchical directories to build a program, then RCS needs a layer on top of it, typically implemented with shell scripts. One such layer is the CVS system which will be described in a following section.

3. Code Distribution With RCS (CDS)

The first iteration of our Code Distribution System (CDS) was designed to support the very thin layer on top of RCS that had been used for two years. With this layer, a developer's working directory consisted of symbolic links to the master source code repository. The repository itself consisted of a read-only reference copy of the latest revisions of the source files and an RCS directory containing the RCS ",v" files used to store all revision levels. From his working directory, the developer could then issue the check out command, *e.g. co -l foo.c*, and he would receive a writable copy for editing. When he compiled and linked using *make,* the symbolic links supplied all source that had not been checked out.

The check in procedure required the only shell script that was needed in this scheme. The check in script had three steps. First an RCS check in command was issued which updated the RCS revision control files. Then, a checkout was performed in the repository to update the reference copy. Finally, a new symbolic link was established in the developer's working directory.

This scheme of using RCS was carried across the network to remote sites by the CDS system. On a daily basis, it would make an update of the master repository's reference files on remote clients. The UNIX *rdist* command was used for these updates. It made a clone copy of a directory across the network by comparing the dates and times of files, taking into account time zone differences, and transferring files that were newer at the source. The remote developers could then link against this reference copy in the same way as developers at the master site. CDS also provided an *update* command to force an immediate update in case a developer didn't want to wait for the next scheduled update.

Remote check in and check out were handled with the *rsh* command. On check out, the remote developer issues a CDS command on his machine, which performed a check out on the master site machine and returned the source code to his working directory. On check in, the CDS command did the check in at the master site, updated the reference copy at both the master and remote sites and re-established the symbolic link in the caller's working directory.

Thus, with CDS complete network transparency was achieved. The remote developers were true peers to those at the master site. They received both the benefits and deficiencies of the code management system. Even at remote sites with relatively poor Internet connections, response time was adequate. For example, McGill University used to have a *ping* echo time of over 1 second (for reference the echo time between SLAC and CERN is 0.3 seconds), but could still do an update of the reference copy in 30 seconds. A check in or check out of any single file was essentially immediate. Naturally, the "closeness" remote developers felt to the master site made them much more productive members of the global development team. This CDS scheme was used for one full year without problems.

4. Concurrent Versions System (CVS)

The Concurrent Versions System, CVS, is a front end to the RCS system which extends it in a number of ways. The two most important are the notion of concurrent development and the extension of a project from a collection of files in a single directory to a hierarchical collection of directories each containing RCS managed files. There is also support for merging updated *third party* releases with local modifications, a modules facility for combining components in the repository in different ways according to a project's needs, producing a *patch* format file[4], preparing a release for distribution and some other tools. Space permits only a discussion of the first two items.

In large software development projects, it is not uncommon for two or more people to have a need to modify the same file at the same time. With RCS or SCCS, this is prohibited because files are checked out locked, thus giving only one person a writable copy of a working file. Although desirable in theory, locking has undesirable consequences such as the serialization of the development process, with some developers waiting on others to finish their modifications and check in the needed files. It may also lead to *cheating* whereby developers make their own modifications on an unlocked version with the intent of merging their changes with others at a later time.

With the CVS system, each developer checks out a writable version of all the source code of a project into his working directory. For larger projects, the hierarchy of the working directory reflects the hierarchy of directories containing the RCS files in the repository. The *modules* facility allows including additional directories in the working directory hierarchy. No lock information is generated in the repository. The developer can then proceed to modify any file he requires. One immediate benefit is stability, no changes in the repository will affect his or her current development. Another benefit is that the developer has all the source code he may need to browse available in his working directory. Optionally, a developer can check out only those files that he plans to edit and make use of libraries to link a program, but this is not as safe as checking out everything that is needed.

At any time, the developer can update his working directory with any changes made to the repository since he last checked out or updated his files. CVS will update the working directory in a number of ways. First, any files added or removed from the repository are also added or removed from the working directory. Secondly, any files updated in the repository and not modified in the working directory will simply be updated. Thirdly, any files not updated in the repository but modified in the working directory are left alone. Finally, any files that are updated in the repository and modified by the developer undergo a three-way merge while the original working file is moved to the side for reference.

The three-way merge (using the *rcsmerge* command) involves the original version the user checked out, the latest version in the repository, and the working file. In practice, merging is not needed very often and, when it does occur, the merge most frequently involves different areas of the file. Occasionally, the attempted merge involves the same lines in the source file, and in this case, the developer is left with a file containing both sets of modifications with a clear indication which lines came from which source and a renamed copy of his working file. It is his responsibility to resolve the conflicts.

It is actually beneficial for true conflicts to be plainly visible to the developer. If two developers modified the same lines of code, someone has to decide which set of modifications are the correct ones. Without merging, one developer might fix code and the other break it later, or vice versa. If one developer doesn't understand the other's modifications, he can go ask. In practice, CVS users have found that conflicts are usually easy to understand and resolve.

Once a developer's working directories are up to date with respect to the repository, that is his files were derived from the same version, he can check in his changes using the *cvs commit* command. CVS first scans the working directory recursively to check that the files are up to date, and make a list of files that need to be checked in. CVS then prompts the user, once for each directory with modified files to enter an RCS log message using his favorite editor. Thus, a consistent set of files are checked in together and the developer doesn't have to worry about forgetting some file. CVS also has *hooks* just before and after the commit process that can be used to run custom scripts that do additional checking or logging on the CVS transaction.

5. Code Distribution with CVS

The goal of the second iteration of a code distribution system was to provide the same level of network transparency for a CVS based system as was done for our simple RCS based system. At the same time, some of the administrative details of the system were simplified. For example, the nightly updating of remote sites is no longer done. Since remote developers eventually receive their own copy of source code in their working directory, there is little need for it. Instead, updates are made on demand as described below.

The remote CVS check out and update procedures follow similar patterns. The first step is to run *rdist* on the master directory where the CVS modules information is stored to update the remote site. From the files received, the directories that are involved in the check out or update are obtained and the *rdist* command is run on those. Once completed, CVS can be run locally to fulfill the developer's request. It is estimated that these steps will take 30-60 seconds depending on how out of date the remote site is with respect to the master. The more active remote developers have been, the less files in the local repository will be out of date.

CVS commands that update the master site, such as *commit*, are the trickiest. The first step is again to obtain which directories are involved, but that information has already been stored in the remote developer's working directory by CVS. Next, the remote site is updated with respect to the master and a *cvs commit* is run locally against the updated directories. Finally, the remote site will update the master. Naturally, CVS locks will need to be applied at both sites in the correct order to avoid two updates of the master from occurring simultaneously (CVS already uses locks for single site updates).

Actually, remote CVS is simpler than CDS. It should be more robust and easier to maintain. It consists of a set of modifications to five of the 40 CVS source files. It works well for sites with reasonable connections to the Internet such as McGill University which currently has a *ping* echo time of about 0.4 seconds. It has even been used with V.32bis modem connections running SLIP/PPP.

6. Conclusions

We have achieved a code management and distribution system that allows remote sites to have transparent access to source code stored at a master site. Transparent access is important for remote developers so that they can be peers in the group development process as is very frequently necessary in HEP. The key elements of our success have been the quality of connectivity that the Internet provides, and the underlying UNIX tools that make use of it.

7. Acknowledgments

The original version of CVS was written as a set of shell scripts by Dick Grune at Vrije Universiteit in Amsterdam. The current version (v1.3) was written in C by Brian Berliner and Jeff Polk while at Prisma, Inc. Many thanks to Brian Berliner (now at Sun Microsystems) for consulting with us during the development of remote CVS, to Dag Brück of the Lund Institute of Technology and Art Isbell of Cubic Solutions for pointing us in the direction of CVS, finally to David Britton of McGill University for stoically testing the remote aspects of both the original and CVS based schemes.

References

[1] Bell Telephone Laboratories, "Source Code Control System Users's Guide.", *UNIX System III Programmer's Manual,* October 1981.

[2] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System.", *Proceedings of the 6th International Conference on Software Engineering,* IEEE, Tokyo, September 1982.

[3] Brian Berliner, "CVS II: parallelizing software development." *Proceedings of the Winter 1990 USENIX Conference*. Washington, DC, January 1990. USENIX, 1990.

[4] Larry Wall, The *patch* program is a tool for applying a diff file to an original and can be found on uunet.uu.net in ~ftp/pub/patch.tar

[5] Fermilab Computing Division, "Expand", *Report PU0093,* Fermilab, 1990.

[6] Adam Boyarski, "FPP", *SLAC report (to be published).*