LUKE KANIES

using version control in system administration

Luke Kanies runs Reductive Labs (http://reductivelabs.com), a startup producing OSS software for centralized, automated server administration. He has been a UNIX sysadmin for nine years and has published multiple articles on UNIX tools and best practices.

luke@madstop.com

VERSION CONTROL TOOLS SUCH AS CVS and Subversion have long been accepted as necessary for software development, but they serve just as admirably in system administration, especially when doing centralized, automated administration, commonly called configuration management. In this article I will discuss some of the benefits of using version control as a system administrator and then provide some simple examples for doing so.

What Is Version Control?

Version control software provides a convenient way to store and manage changes to files over time. They generally involve a repository for storing all of the file versions, along with client tools for interacting with the repository, and most modern tools support network access to the repository. Although the details vary from tool set to tool set, basically all of them support a similar subset of actions:

- Add files to the repository
- Commit new changes to the repository, recording date and author
- Retrieve changes from the repository
- Compare a file with the repository

There are many different version control systems available, both commercial and open source. They generally have similar client-side features; where they differ most is in how the repositories are maintained or can be used. For instance, CVS and Subversion (both open source) require a single master version repository, while GNU Arch (open source) and BitKeeper (commercial) allow for distributed version repositories, so disconnected users can still perform versioning operations. This article is focused mostly on the client side of version control and thus won't benefit from the additional features of GNU Arch, so I will settle for CVS and Subversion for my examples.

CVS is more common than Subversion because it has been around longer, and it is significantly easier to compile from scratch, so it is a reasonable choice for most purposes. Many operating systems now ship with Subversion installed, though, and Subversion has some key benefits over CVS, most notably that using it over a network is significantly better.

The Benefits of Version Control

For those unfamiliar with version control and why it is so useful, it is worthwhile summarizing some of its benefits. Its true value can change dramatically depending on circumstances—in particular, it becomes far more valuable when many people are modifying the same files or when the same files are used on many machines—but it provides some value to nearly every organization.

The two greatest values it provides are a log of all changes you've made and the ability to easily revert to any version. While backups provide something like this, their granularity is usually relatively low—at most once a day, and often less. With version control, you decide how much change is worth committing a new version (usually, one work session correlates to one new version), and you can always come back and revert to a specific version in one simple command.

An oft-overlooked benefit of version control is that it provides a very easy way to centralize information. I version-control all of the configuration files in my home directory (notably not most content files, just the config files—MP3s don't belong in version repositories), along with all of the articles I write (including this one). This makes it easy to have a consistent login environment on all of my machines, which I find stupendously useful as a sysadmin, and it also makes it easy to sync data between my desktop and laptop when I travel. I also like committing changes from my laptop back to a central server when I'm traveling, as it's an easy way to make off-site backups of new content.

Version Control for Sysadmins

System administrators using version control software will often find themselves making decisions that software developers do not encounter. In particular, the final product of software development is usually wrapped into some kind of package or binary and shipped to customers as a software release, but the final product of system-administrative version control is as individual configuration files on production servers. This difference provides some options that are not open to most software developers.

Software developers generally make all of their modifications in what is called a "sandbox," which is a checked-out copy of the repository in their home directory. Changes in this sandbox do not modify the repository until they have been committed, so mistakes can be made and fixed without anyone else knowing or caring. Developers make their changes, test them, and then commit them to the repository, which is the first time those changes can affect anyone else.

System administrators do not necessarily need a sandbox, though; they can check out the files directly on production servers, which can immediately change the running state of the system. This is an important design point: as a system administrator, you can choose to use a sandbox, which provides a clean separation between file modification and deploying those modifications to the server, or you can choose to modify the files directly on your servers, which provides no such separation but is much simpler.

I always recommend making your changes in a sandbox whenever possible, partially because it makes the security picture much cleaner (normal users only modify the repository, and a special user can be used to retrieve new versions) but also because it forces you to commit any changes you make—you make your changes, commit them to the repository, and then retrieve them on your servers (usually automatically, using a tool like Puppet or cfengine). Otherwise, users can make changes on the production servers without committing them to the repository, which can be problematic.

The downside of using a sandbox to make all of the changes is that it makes it more difficult to test changes, since they often must be deployed before you can test them, and it does add some complexity.

One of the other differences in using version control for system administration is that you will always have to have a privileged system user actually retrieve updates from the repository rather than doing so individually, as developers do. Only a privileged user will have write access to the files you are maintaining, and you also won't want to require that a user be logged in to retrieve file updates.

Per-Server Version Control

Smaller sites may not need centralized version control, especially those with only one or two servers, but could still benefit from better version records. In those cases, it might make sense to create a version repository for each server; this retains the fine granularity of change recording along with the ability to easily revert to older known-to-be-good configurations while adding very little maintenance overhead.

CVS's simplicity makes it perfect for this usage. Create a CVS repository according to the documentation at http://www.nongnu.org/cvs/. I will only manage /etc/apache2 here, but you could just about as easily manage the entire /etc.

This is very simple—just import your /etc/apache2 directory into the server's version repository:

```
$ cd /etc/apache2
$ sudo cvs import -m "Importing" etc/apache2 LAK gibberish
<feedback from CVS>
```

I use sudo here to do the work, because I make it a policy never to do any work while logged in directly as root—sudo logs everything I do as root, which I find extremely valuable. The `-m` flag to `cvs import` provides the log message that you want associated with this change; `cvs log` retrieves these messages, along with the date and author of the change, so you can figure out not only what changed but why (of course, these messages are useless if you don't provide useful information in them). The `etc/apache2` argument just tells CVS where to put the files inside the repository, which we will just map directly to the system.

The next two arguments are basically useless to sysadmins, although I assume that developers find them useful. I usually use my initials for the second argument (which is normally a vendor tag) and some gibberish for the third argument (which is supposed to be a release name but strangely cannot start with a number or contain any non-alpha characters).

A desirable but somewhat surprising aspect of this import is that it does not modify anything in /etc/apache2, it just copies the state of the directory into the repository.

Once the files are imported, check them out into a temporary location and then copy them into place:

```
$ cd /tmp
$ sudo cvs checkout etc/apache2
<feedback from CVS>
$ cd /tmp/etc/apache2
$ sudo cp -R . /etc/apache2
```

The CVS checkout creates the entire path in my current directory, so in this case it creates /tmp/etc/apache2, with the versioned content in it.

I copy the files into place because CVS is not able to manage symlinks, which are heavily used in Debian's Apache2 configuration (which is what I am using). Copying the files allows me to just put the now-versioned files in place without messing with the symlinks.

The only difference you will notice in /etc/apache2 is the presence of a CVS directory in each subdirectory, which is used by CVS to manage file versions. Do not modify or delete this directory or its contents, as doing so will effectively disable CVS.

This system requires only one addition to your normal workflow: after you make a change to a configuration file, commit that change to CVS. For instance, here is what it would look like modifying one of your virtual host configurations:

```
$ cd /etc/apache2/sites-available
$ sudo vi reductivelabs.com
<edit file>
$ sudo cvs ci -m "Modifying rewrite rules" reductivelabs.com
<feedback from CVS>
```

CVS finds the change and commits it to your repository. If you do not specify a file on the command line, CVS will search the entire directory tree looking for changes. Sometimes this is desirable, but not always. File modifications are all stored relative to the repository root, so you do not have to worry about duplicate file names within a repository—in this case, CVS uses its control directory to construct the path to the file I've modified, etc/apache2/sites-available/reducctivelabs.com, and applies the change to the equivalent file in the repository.

This may not seem useful—after all, you do have backups, right?—but it becomes incredibly valuable when you accidentally break your configuration and you need to restore immediately, which you can do by just updating to yesterday's revision (as one way of reverting). I've often made changes that I thought worked just fine only to figure out a week or more later that the change broke some small part of my site; and I usually only find it out when it's suddenly a crisis but it's been long enough that I don't remember exactly what I changed. CVS allows me to undo the most recent change without having to delve into a backup system, and then it allows me to go back and figure out exactly what changed, when, and maybe even why. This is especially useful when the other guy broke it but you have to make the system work while keeping the change.

Site-Wide Scripts Directory

The next example will create a versioned, centralized repository for all those scripts that every site uses to perform different maintenance tasks around the network. Most sites I have been at use ad hoc mechanisms to get these scripts where they need to be, such as using scp to copy them over when necessary, but these ad hoc mechanisms often result in scripts that behave slightly differently on different systems, because scripts are modified when necessary but then not propagated to the entire network.

I will use Subversion for this example, both because its networking is much easier to set up and because it manages file modes in addition to content, which is important since all of these scripts will need to be executable. I will be storing the scripts at /usr/local/scripts, but you should use whatever is appropriate for your site. Creation and configuration of a Subversion repository are beyond the scope of this article, but the documentation on Subversion's Web site (http://svnbook
.red-bean.com/) does a great job of covering the process.

Because these are essentially independent scripts that can be tested as easily from a sandbox as from within your scripts directory, I will use a sandbox for modifications. This provides a one-way flow of changes: I commit changes from my sandbox, which then flow to each server.

One of the benefits of Subversion over CVS is that access control is much more flexible and powerful. Subversion over HTTP uses a relatively sophisticated configuration file to determine access, and standard HTTP authentication is used, which means that your Subversion server does not need a normal user account for Subversion users. To guarantee that changes are one-way (that is, that users cannot make changes on the local server and then commit them back), I create an HTTP user, configure Subversion to allow only read-only access to the repository, and then use that user to retrieve file updates.

This does introduce a dichotomy that can be somewhat confusing—most Subversion operations will involve a real user on the local machine and a Subversion user. In the case of the system administrators, those users are generally equivalent, but you are likely to be doing read-only operations as the local root user and authenticating to the repository as a different user (I often use an svn user for all read-only access). Depending on the data you are versioning, you may not even require a password for this user (but if you do use a password, make sure you send it over SSL). Also, Subversion can do credential caching, so that you only need to provide a password the first time, which is especially useful for automation. This does leave a password in your root user's home directory, but that's at least as secure as storing the password where a script looks for it, and this necessary caching is just another reason to use a read-only user.

Once you have your repository and user created, import one of your current scripts directories into the new repository as a user with write access to the repository (usually, your own account):

```
$ cd /usr/local/scripts
$ sudo svn import https://reductivelabs.com/svn/scripts
Adding  ioperf ... Committed revision 1.
$
```

As before, the import did not modify our local files. To get the version-controlled files in place on the server, you need to do a switcheroo between the existing scripts directory and the new repository:

```
$ cd /usr/local
$ sudo mv scripts scripts.old
$ sudo svn co https://reductivelabs.com/svn/scripts
<authenticate as read-only user>
A scripts/ioperf ... Checked out revision 1.
$
```

It is worth saving the old scripts directory until you are sure that you have everything working as desired.

You will find a .svn directory in your newly checked-out directory, which is analogous to CVS's CVS control directory.

You need to perform this switcheroo on all of the machines on which you want this directory available. It is straightforward to write a short script (ironically) to perform this task, and you can also automatically create the credentials for the user doing the updates by copying down a ".subversion" configuration directory for the user doing the checkouts. Again, a configuration management tool makes this significantly easier.

Making Changes

To make changes to the repository, check out the files in your sandbox (which I usually name something like "svn"):

```
$ mkdir ~/svn
$ cd ~/svn
$ svn co https://reductivelabs.com/svn/scripts
$ cd scripts
<make changes>
$ svn ci -m 'I made a change'
<feedback from Subversion>
```

Then you need to update the production copy:

```
$ cd /usr/local/scripts
$ sudo svn update
<list of updates>
```

This updating after each change can get tedious, which is why configuration management tools are usually used to automate it (although it could also be done with a simple cron job). Automation of these updates is especially desirable in this case, since you will want all of your machines to perform this update.

What Have We Gained?

Where it was previously difficult to keep our script repositories in sync across all of our systems, or even to know if they were in sync, using our central version repository it is now very simple. Normal users make all necessary changes in their own sandboxes, which is where they also test those changes. They then commit the changes, which are deployed automatically to all of the servers.

Unfortunately, I have presented a bit of a best-case situation, where all of your scripts are already in sync and you just want to keep them that way. It is much more likely that as you deploy the controlled scripts to each server in turn, you will find some local modifications that you will need to handle. In doing so, you will want to look at how to handle merging and conflict resolution, which is also fortunately well covered in the documentation.

Conclusion

With my first example, that of version-controlling /etc/apache2, I provided a simple way for small sites to track and log all of the configuration changes they make, which is quite valuable. I know of sites that have hard-copy books for this purpose, but those books cannot approach the functionality of a version control system.

The second example delved into using version control to centralize common files, and can be used as an example for any set of files that is duplicated on many machines. One of the additional benefits of this example is that users can be given the right to modify version-controlled files without even being given an account on the system to which the files are deployed. This works excellently with groups like Web developers—they commit their changes to the version repository, and the changes are automatically deployed to the servers, without the sysadmins needing to interfere but also without giving the Web developers unnecessary rights on the Web servers, which can be especially important in Internet-facing servers.

I hope this article has convinced you that version control is just as valuable to system administrators (even home administrators) as it is to developers. It can save individuals plenty of headache, but for large groups I consider it indispensable.