Setting up Git and Mercurial Servers

GitHub provides an excellent web-based interface to Git with extensive project management tools. Bitbucket provides an equally excellent web-based interface for Mercurial.

What you will learn…

- How to configure permissions on Git and Mercurial servers
- How to manage users and groups for DVCS platforms
- Conceptual differences in managing DVCS from CVS and Subversion

What you should know…

- How to install applications
- How to manage users, groups, and file permissions
- How to use Git and Mercurial

However, project requirements, management concerns, or security needs may prevent the use of public storage tools for distributed version control. Under these circumstances, both Git and Mercurial are easy to set up and use on a BSD-based server. The niceties of the web interfaces are lost, but the full power of both *distributed version control system* (DVCS) platforms are available at the command line.

This article outlines the basic directory and permissions structure necessary to maintain a Git or Mercurial server on a BSD platoform and accessible over SSH. However, this article assumes are you already familiar with how DVCS platforms operate and with server and SSH operations.

In addition, this article assumes you are familiar with installing applications through the ports and package systems, as appropriate, for your operating system. In general, these tips are equally valid on other Unix-like platforms, as well.

Incidentally, there is no reason not to manage both Git and Mercurial servers on a single server. The two DVCS platforms operate independently of each other and do not interfere with each other. This is valuable if local conventions cannot be mandated and cooperation with external entities mandates working with both Git and Mercurial. Because Git and Mercurial repositories ultimately form a mesh or star network of patches and forks, working with an external repository can be aided by maintaining a local server which centralizes synchronization.

Installation

Unlike some systems, neither Git nor Mercurial require separate servers in the usual sense. Both can operate over SSH and HTTP. Git can also transport version control information over a native protocol, but this protocol's server is bundled directly into the Git client. However, both require their respective client to be installed on the server to operate it. Because of this, installation on a BSD-based server is as simple as installing the clients. Both Git and Mercurial can be installed using your BSD's native application packaging system or can be configured and installed directly from the package distributions provided by each development group.

Of note, Git is mostly C language and consists of many different programs each of which provides small parts of program's subcommands. Some are implemented in Perl and as shell scripts. In contrast, Mercurial is pure Python and requires a complete Python installation as a result. Both are relatively easy to install when using the native packaging system.

A Repository Home

One of the key aspects of both Git and Mercurial is how they store their repositories. If you are familiar with CVS

or Subversion, these turn version control on its ear. For CVS and Subversion, the working copy after a checkout is an image of the repository at a certain point in time. The history is stored in a central location. DVCS systems change this by packaging the history with each copy of the repository.

With CVS and Subversion, the server copy is special and cannot be treated as a working copy. A Git or Mercurial *server* is a copy of the repository just like any other, though the local checkout may not be present. Because of this, a Git or Mercurial central repository requires minimal planning and foresight. Indeed, the idea of a central repository in Git and Mercurial is more of a social convention than something technically enforced.

The first question to answer is where will storage of these repositories be kept. It is not unreasonable to store them with user accounts under `/home`, using `/home/git` and `/home/hg` for each. Given the nature of source code repositories, storing them under `/var` or `/var/db` is also reasonable. In this case, I have used `/var` for both repositories leading to the directories `/var/git` and `/var/hg`.

In each case, I created symbolic links from `/git` to `/var/git` and `/hg` to `/var/hg`. This shortening will be useful in creating remote paths. When tunnelling Git over SSH, paths are mapped one-to-one and shorter paths are desirable. With symbolic links in place, the path becomes `user@host:/git/repo`. Repositories on other locations can be accessed in the usual way, with one in howardjp's home directory being addressed as `user@host:/home/howardjp/repo`.

Mercurial offers the same advantage, but with a slightly different nomenclature. When using SSH, Mercurial requires a protocol specification that Git does not, so SSH-tunnelled Mercurial connections resemble `ssh://user@host//hg/repo`.

Managing Repository Permissions

Repositories themselves are managed in the tradition BSD way. In my example, I have created two user accounts to manage these storage areas. From `/etc/passwd`:

```
git:*:902:99:Git Repository Owner:/var/git:/usr/sbin/
                    nologin
hg:*:903:99:Mercurial Repository Owner:/var/hg:/usr/sbin/
                    nologin
```

Like all properly managed role accounts, these accounts are disabled through the use of an asterisk in the password field. Additionally, both have their shells set to nologin, which automatically disconnects a user when launches the shell. The only purpose of these accounts is to own the parent directory for repositories and they could be merged into one account, if that is the local preference.

The group number listed, 99, is a group called `src`, which is otherwise unremarkable. Any group name and number will do. Users can be added to the src group to give them access to both Mercurial and Git repositories. Further restrictions of access are possible with the usual BSD group mechanisms. If ACLs are available due to special filesystem capabilities, they will be honored, as well.

But if a repository is meant to be shared among multiple users, it should have its permissions set appropriate to ensure all necessary users share read and write access correctly. The logic way to manage this is by setting the group on a repository to a project's group and making the repository readable and writable by the group. This must be done recursively on all files in the repository directory.

Users familiar with administering CVS central repositories can lock down individual components within the CVS tree and mark off sections of the tree for editing by some users through BSD's permissions structure. With both Git and Mercurial (and, incidentally, Subversion), this type of restriction is not possible. Git and Mercurial use an internal database format for storing changes leading to an all or nothing permissions situation. Environments which require multiple sets of editing permissions on repositories are best off dividing projects into multiple repositories.

Conclusions

These basic steps will help ensure a smoothly running and easier to maintain Git or Mercurial server. However, these tips cannot address every possible issue or local configuration requirement you may encounter in building a Git or Mercurial server. But these tips will provide the foundation for a sound server installation for DVCS platforms. Fortunately, unlike other popular version control systems, Git and Mercurial will continue functioning when the server is unavailable allowing the opportunity to fix mistakes.

JAMES P. HOWARD, II

The author is a senior analyst in Washington, DC, in the United States where he focuses on statistical and mathematical systems. He can be reached at jh@jameshoward.us or via Twitter @howardjp.