SSH Can Do That?

Productivity Tips for Working With Remote Servers

by Smylers

Smylers@stripey.com • Twitter: @Smylers2

2012 February 29th • latest version: http://blogs.perl.org/users/smylers/2011/08/ssh-productivity-tips.html

SSH has many features which are helpful when working regularly with files on remote servers; together they can give a vast increase in productivity over the bare use of SSH. If you regularly use SSH, it's worth spending a little time learning about these and configuring your environment to make your life easier.

This has been presented at Yapc Europe 2011 in Riga and the Floss UK Spring 2012 Conference in Edinburgh. If you'd like me to come and talk about this at your user group or workplace, please get in touch.

Multiple Connections

Often it's useful to have multiple SSH connections to the same server, for example to edit a file, run some file-system commands, and view a log file all in different terminal windows. Except sometimes that can seem too much hassle, so we compromise and end up repeatedly cycling through quitting and restarting a few different commands in one window.

Fortunately OpenSSH has a feature which makes it much snappier to get another terminal on a server you're already connected to: **connection sharing**. OpenSSH is the implementation of SSH that comes with many Unix-liked operating systems, including all the common Linux distributions and OSX.

To enable connection sharing, edit (or create) your personal SSH config, which is stored in the file *~/.ssh/config*, and add these lines:

```
ControlMaster auto
ControlPath /tmp/ssh_mux_%h_%p_%r
```

Then exit any existing SSH connections, and make a new connection to a server. Now in a second window, SSH to that same server. The second terminal prompt should appear almost instantaneously, and if you were prompted for a password on the first connection (which we need to sort out anyway — keep reading — but it's a convenient way of verifying this shared connection stuff) you won't be on the second.

What About Windows Users?

Some of these productivity features are implementation-specific, so not available if you use a different SSH client, such as Putty. However, OpenSSH is available for Windows. If some of these OpenSSH tips sound useful to you (and you don't feel like switching to a different operating system) it may be worth installing it: http://sshwindows.webheat.co.uk/

Copying Files

Shared connections aren't just a boon with multiple terminal windows; they also make copying files to and from remote servers a breeze. If you SSH to a server and then use the `scp` command to copy a file to it, `scp` will make use of your existing SSH connection — and in Bash you even get `Tab` filename completion on remote files. Connections are also shared with `rsync`, `git`, or any other command which uses SSH for connection.

Repeated Connections

If you find yourself making multiple consecutive connections to the same server (you do something on a server, log out, and then a little later connect to it again) then enable **persistent connections**. This is simply one more line in your config (in addition to the two above for shared connections):

```
ControlPersist 4h
```

That will cause connections to hang around for 4 hours (or whatever time you specify) after you log out, ready to spring back into life if you request another connection to the same server during that time. Again, it really speeds up copying multiple files; a series of `scp` commands doesn't require authenticating with the server each time.

Don't Type Passwords

If currently you type a password when making an SSH connection, you can make connecting much more pleasant by setting up **SSH keys**. With keys you do get prompted for a pass phrase, but this happens only once per booting your computer, rather than on every connection. With OpenSSH generate yourself a private key with:

```
$ ssh-keygen
```

and follow the prompts. Do provide a pass phrase, so your private key is encrypted on disk. Then you need to copy the public part of your key to servers you wish to connect to. If your system has `ssh-copy-id` then it's as simple as:

```
$ ssh-copy-id smylers@compo.example.org
```

Otherwise you need to do it manually:

1. Find the public key. The output of `ssh-keygen` should say where this is, probably *~/.ssh/id_rsa.pub*.
2. On each of your remote servers insert the contents of that file into *~/.ssh/authorized_keys*.
3. Make sure that only your user can write to both the directory and file.

Something like this should work:

```
$ < ~/.ssh/id_rsa.pub ssh clegg.example.org 'mkdir -p .ssh; cat >>
.ssh/authorized_keys; chmod go-w .ssh .ssh/authorized_keys'
```

Then you can SSH to servers, copy files, and commit code all without being hassled for passwords.

SSH Keys with Putty

Putty can do SSH keys too. Download PuttyGen and Pageant from the Putty website, use PuttyGen to generate a key, copy it to remote servers' *.ssh/authorized_keys* as above, and then run Pageant. Introduce Pageant to your private key, and it will keep running in the background. Putty will spot this, and automatically use the key provided by Pageant instead of prompting you for a password. Full details are in chapters 8 and 9 of the Putty manual.

Onward Connections

Sometimes it's useful to connect from one remote server to another, particularly to transfer files between them without having to make a local copy and do the transfer in two stages, such as:

```
www1 $ scp -pr templates www2:$PWD
```

(Aside: note how useful `$PWD` is when copying between servers with common directory layouts.) Even if you have your public key installed on both servers, this will still prompt for a password by default: the connection is starting from the first remote server, which doesn't have your private key to authenticate against the public key on the second server. Do not 'fix' this by copying your private key to remote servers; you don't want to have copies of that stored on servers' disks, and anyway it doesn't help much cos you still need to provide a pass phrase to decrypt it. Instead use **agent forwarding**, with this line in your *.ssh/config*:

```
ForwardAgent yes
```

Or in Putty check the box 'Allow agent forwarding'. Then your local SSH agent (which has prompted for your pass phrase and decoded the private key) is forwarded to the first server and can be used when making onward connections to other servers. Note you should only use agent forwarding if you trust the sys-admins of the intermediate server.

Don't Type Full Hostnames

It's tedious to have to type out full hostnames for servers. Typically a group of servers have hostnames which are subdomains of a particular domain name. For example you might have these servers:

- `www1.example.com`
- `www2.example.com`
- `mail.example.com`
- `intranet.internal.example.com`
- `backup.internal.example.com`
- `dev.internal.example.com`

Your network may be set up so that short names, such as `intranet` can be used to refer to them. If not, you may be able to do this yourself even without the co-operation of your local network admins. Exactly how to do this depends on your OS. Here's what worked for me on

a recent Ubuntu installation: editing */etc/dhcp/dhclient.conf*, adding a line like this:

```
prepend domain-search "internal.example.com", "example.com";
```

and restarting networking:

```
$ sudo restart network-manager
```

The exact file to be tweaked and command for restarting networking seems to change with alarming frequency on OS upgrades, so you may need to do something slightly different.

Hostname Aliases

You can also define hostname aliases in your SSH config, though this can involve listing each hostname. For example:

```
Host dev
  HostName dev.internal.example.com
```

You can use wildcards to group similar hostnames, using `%h` in the fully qualified domain name:

```
Host dev intranet backup
  HostName %h.internal.example.com

Host www* mail
  HostName %h.example.com
```

In Putty you can save a separate session for each hostname, then double-click on it to start a connection to that server. (I don't think there's a way of using wildcards to specify hostname transformations though.)

Don't Type Usernames

If your username on a remote server is different from your local username, specify this in your SSH config as well:

```
Host www* mail
  HostName %h.example.com
  User simon
```

Now even though my local username is `smylers`, I can just do:

```
$ ssh www2
```

and SSH will connect to the `simon` account on the server. Again, Putty users can save usernames in their session config to avoid being prompted on each connection.

Jumping Through Servers

Sometimes you can't make a network connection directly to the server you wish to access; you have to first SSH to an intermediate server and then on to the server you want. This can also be automated. First make sure that you have keys and agent forwarding set up so that you can SSH to the intermediate server in one command and from there to the target server in a second command, each without any prompting:

```
$ ssh gateway
gateway $ ssh db
```

Then in your local SSH config, specify that a connection to the target server should be proxied through the intermediate server using `netcat`:

```
Host db
  HostName db.internal.example.com
  ProxyCommand ssh gateway netcat -q 600 %h %p
```

Then you can just do:

```
$ ssh db
```

And, after a brief pause while SSH chugs through authenticating twice, you'll have a shell on the second server. Note that `netcat` is sometimes spelt `nc` or `ncat` or has a `g` in front of it; check what the command is called on your intermediate server.

Escaping from Networks

Sometimes the problem is that you can't SSH out of a network; you've been provided with web access but SSH is blocked. Or rather port 22, the default SSH port, is blocked. You can allow for this by configuring your SSH server to listen on port 80 or 443, which being web ports will be accessible from any network that provides web access. Edit */etc/ssh/sshd_config* on a server, add this line:

```
Port 443
```

and restart the SSH server:

```
$ sudo reload ssh
```

Of course this only works if your server isn't already serving HTTPS web pages. But you only need to set up one such server on the internet; once you have SSHed there, you can make onward connections to port 22 on other servers as normal. And remember you need to set this up in advance; once you've found yourself on a network with only web access, you're stuck unless you can contact somebody else with access to your SSH server to reconfigure it for you. So how about setting it up now?

Defeating Web Proxies

Sometimes not only does a network provide just web access, but to get even that you have to use a web proxy. Fortunately a program called **Corkscrew** can send SSH traffic through a web proxy. Corkscrew is really simple to use. Whenever I've needed it, I've searched the web for it, downloaded it, followed the instructions on its website, and it's just worked. You (temporarily) use configuration like this:

```
ProxyCommand corkscrew proxy.example.org 8080 %h %p
```

Gui Applications and Remote Windows

It can be useful to run gui programs on files on remote servers. For example, to edit an image, or view a PDF file, or simply for editing code if your text editor of choice isn't terminal based; I find GVim more usable than terminal Vim, and also like that running `gvim` opens a new window and frees up the shell prompt for typing further commands. This can be made to work over SSH too, using a feature call **X forwarding**. Enable it in your config:

```
ForwardX11 yes
```

This also requires the server to co-operate. It needs to allow X forwarding, enabled with this line in /etc/ssh/sshd_config (and a restart):

```
X11Forwarding yes
```

It also requires the `xauth` command installing, as well as the editor, image viewer, gui debugger, or any other graphical applications you wish to run. This works on Linux and any other operating systems with a local X server. X servers for Mac and Windows are available, but you may be more hassle than they are worth; switching to using Linux may be easier.

Operating on Remote Files Locally

An alternative to having remote gui applications display locally is to have local gui applications operate on remote files. This can be done with **SSHFS**. Simply create an empty directory then use `sshfs` to mount a remote directory there; specify a server and directory on that server, and your empty directory:

```
$ mkdir gallery_src
$ sshfs dev:projects/gallery/src gallery_src
$ cd gallery_src
$ ls
```

Then you can run any local applications on files in that directory. Well, files that *appear* to be in that directory; magic happens which makes them appear there whenever requested, but actually stored on the server. To unmount you need to use the `fusermount` command. Don't worry if you find the name of that command hard to remember; the `sshfs` manual page includes it in the synopsis at the top.

```
$ cd ..
$ fusermount -u gallery_src
```

SSHFS works on Linux and OSX. I haven't found anything equivalent for Windows users.

Using Vim on Remote Files

Vim can edit remote files, using `scp` URLs:

```
$ gvim scp://dev/projects/gallery/src/templates/search.html.tt
```

This uses a Vim plug-in, but it's one shipped with Vim and enabled by default.

Obviously this is more restricted that SSHFS, but it may be handier in circumstances where editing a remote file or two is all you want to do. And this can be made to work on Windows. In Vim see:

```
:help netrw-problems
```

Connecting to Remote Services with Local Apps

Sometimes there is a service, such as a database or a web server, which is available on a remote server but it would be handy to connect to it from a local application. This can be achieved with **port forwarding**. For example, if the `db` server is running Postgres (and only

allows access locally) you can put this in your SSH config:

```
Host db
  LocalForward 5433 localhost:5432
```

Then when you SSH to that server it sets up port 5433 (a number I just made up) on your computer to forward all traffic to port 5432 (the standard Postgres port) on the server, and that, so far as the server is concerned, the connection comes from `localhost`.

```
$ ssh db
```

So in another window (or after immediately exiting the shell created by the above command if you're using persistent connections) you can connect to the DB with a local Postgres client:

```
$ psql -h localhost -p 5443 orders
```

This is especially handy if you wish to use a graphical DB client which isn't available on the server:

```
$ pgadmin3 &
```

Or if you have a 'back-end' web server which isn't directly accessible on the internet, you can forward it traffic from a local port:

```
Host api
  LocalForward 8080 localhost:80
```

Then SSH to the server:

```
$ ssh api
```

And point a local web browser at the port number you picked to view the server's output:

```
$ firefox http://localhost:8080/
```

Avoiding Delays

If connecting to a server seems to sit there for a few seconds not doing anything, try adding this line to your config:

```
GSSAPIAuthentication no
```

And if that works, ask the server's sys-admin to disable it in the server config, for the benefit of all users — exactly the same line as above, but in */etc/ssh/sshd_config*.

Faster Connections

If you are connecting to a server across a network which is already secure (such as your internal office network) then you can make data transfer faster by choosing the `arcfour` encryption algorithm:

```
Host dev
  Ciphers arcfour
```

Note this speed-up is achieved by using less secure 'encryption', so do *not* do this for connections over the internet. Make sure you only specify it for particular hosts, not as the default. And only use it on a laptop (which of course can be used on multiple networks) for connections that go over a VPN.

Go and Do It!

The above are a collection of productivity tips for using SSH. If you have any more to share, please do get in touch on Smylers@cpan.org or to @Smylers2 on Twitter.

Now go and use these features. Take a little bit of time now to get SSH set up nicely, and make working with remote servers easier for yourself.

Acknowledgements

Thank you to Aaron Crane, who as well as being my keyboard monkey during the talk also taught me much of what I know about SSH, some of which came from his Speeding up SSH Logins article, to Paul Knight for the tips about Vim's `scp` URLs and persistent connections, and to Aristotle (@apag on Twitter) for the tip about `arcfour`.