Transfer Tips, Part I

By Jerry Peek

You're in Cape Town, but your data is in California. You're using a Sun system in your office, but your bleeding-edge Mozilla browser with the very cool theme and all of your favorite bookmarks is on your Linux box at home. How can you get bits from there to here without being *there*?

Linux has more remote access solutions than you might realize. You've probably heard of *scp* and *ssh*. But did you know that you can run an X application securely through an *ssh tunnel?* Or that *ssh* can run almost any command line on the remote system — not just a simple command? Indeed, both (and more) are possible, so let's dig in and learn how.

If you aren't familiar with *ssh*, see the introduction to OpenSSH in the December 2000 issue of Linux Magazine, available online at http://www.linuxmagazine.com/2000-12/openssh_01.html.

scp Helpers: Two Shells

The secure file-copy command, *scp*, has a syntax like the Linux *cp* command: the first arguments are the files to copy and the final argument is the destination. Unlike *cp*, *scp* lets you copy files to and from a remote host. If you want to refer to a file on a remote host, simply prepend a hostname (and a username if it's different than your local username) and a colon (`:`) to the file path and name.

For instance, to copy both the file *.cshrc* from your home directory on host *foo.bar* and the file *prog* from *zoe*'s *bin* subdirectory on host *baz* into the current directory (`.`) on your current host, use the command:

```
% scp foo.bar:.cshrc zoe@baz:bin/prog .
```

Enter your *ssh* password or passphrase if you're prompted for it. (If you don't want to repeatedly type your password or passphrase, check out *ssh-agent* and *ssh-add*. The sidebar "SSH Agent" has more details and setup tips.)

How can you copy all of the files from your local *bin* directory to *zoe*'s *bin* directory on *baz*? A wildcard will do it:

```
% scp bin/* zoe@baz:bin
```

(If you need a refresher on relative pathnames like `bin/*`, see "What's in a Pathname?" in the October 2002 issue, available online at http://www.linuxmagazine.com/2002-10/power_01.html.)

Now, let's try to turn the previous example around and copy all files from *zoe*'s remote *bin* to your local *bin*:

```
% scp zoe@baz:bin/* bin
zoe@baz:bin/*: No match.
```

Unfortunately, your local shell won't run this *scp* command because the wildcard doesn't match.

What's wrong? When you type a wildcard like `*` or `?` on a command line, the *shell* (*not* the application program, *scp*) tries to expand the wildcard into matching pathnames *on the local host where the shell is running*. (The shell doesn't know anything about *scp* or remote hosts. It just knows how to expand wildcards into pathnames.) Because there's (probably!) no local subdirectory named `zoe@baz:bin`, the shell prints `No match` and aborts without running the application (*scp*).

(By the way, if you use *bash*, the *scp* command will probably succeed — but only by luck. A Bourne-type shell starts *scp* and passes the *unexpanded* argument to it. But there's always a chance that your argument will expand into a pathname on the local host, so it's always safer to tell the local shell, even *bash*, "Don't try to interpret this wildcard!")

You can prevent processing of wildcards by the local shell by, escaping (or quoting) the wildcard character(s) that you don't want interpreted like this:

```
% scp zoe@baz:bin/\* bin
```

This *scp* command works. Why? Here, the local shell sees the backslash, removes it and passes the result (`zoe@baz:bin/*`) to *scp*. *scp* parses the argument, finds the username and hostname, and passes the rest of the command line to a remote shell running on *baz*. The remote shell tries to expand its command line argument, which is `bin/*`. Since the remote *scp* shell's current directory is the home directory on that host, the wildcard pathname expands into all relative pathnames starting with `bin/`, which is *bin/afile*, *bin/bfile*, and so on. Finally, the remote shell gives those arguments to the remote *scp* process, which arranges to copy those files back to your local *scp* process, which writes them to the destination.

Whew!

Why is this Shell Stuff Important?

Once you understand that there are two shells — one local shell and one remote shell — involved with *scp* (and, as we'll soon see, with *ssh*), you can use the shells' interpretation and execution powers to do much more than just copying a few files or starting a remote program.

Let's consider an important example: copying a directory tree. The *scp* `-r` option copies all files and links from a directory and all of its subdirectories. But, like `cp -r`, the `scp -r` command creates plain files — not links — at the destination. It also copies files one-by-one, which can be tedious and a bit slow if the directory tree is big.

The shell (and other programs) can solve this problem — and a lot of others! But first we need some more background.

The *ssh* Data "Pipes"

scp and *ssh* open an encrypted connection between your local host and each remote host, and use the connection (or connections) to transfer data in both directions. The arguments to *scp* are pathnames, while the arguments to *ssh* are complete command lines.

Here's a trivial example: assuming you have a home directory on the remote host named *bar*, the following command lists the contents of that directory:

```
% ssh bar ls -F
bin/ xfr/
```

How does it work? The local *ssh* got three command-line arguments: `bar`, `ls`, and `-F`. The first argument is the remote hostname and the rest of the arguments represent the command line for the remote shell. The remote *ssh* process runs `ls -F`, encrypts the standard output of that *ls* process, and routes the encrypted output to the originating (local) *ssh* process. The local *ssh* process decrypts the informations and prints it to its own standard output in your terminal window.

How can you save that listing into a file named *bar-ls*? Just use the shell's redirection operator > like this:

```
% ssh bar ls -F > bar-ls
% cat bar-ls
bin/ xfr/
```

Redirection is another instance where it's important to understand how the shell works. *Before* the shell runs the local application (*ssh*) it first arranges for the standard output of that process to go to the file *bar-ls*. Then the local shell runs the program. So, the file will be created on the local host, by the local *ssh* and the local shell!

What if you wanted to run a remote process and redirect *its* output to a file *on the remote host*? Tell the *remote* shell to do the redirection. How do you do that? By quoting the > operator. That prevents the local shell from interpreting the redirection operator. It's probably simplest if you quote the entire remote command line, like this:

```
% ssh bar 'ls -F bar-ls'
```

SAVING KEYSTROKES WITH *SSH-AGENT*

The job of the *ssh-agent* and *ssh-add* programs is to collect your passphrase once — typically, as you start your login session — and then let you use *ssh* and *scp* commands without entering your passphrase again. This makes working with remote systems almost as easy as working on your local host. Learning how it works will help you set it up, so let's have a look.

ssh-agent makes a socket in a subdirectory of */tmp* and starts a detached process that negotiates with later *ssh* and *scp* commands. It also outputs shell commands that set two environment variables. Because of that, you need to run *ssh-agent* using command substitution (backquotes) and the shell's *eval* command to read that output. Finally, *ssh-add* adds identities to the agent.

```
% eval `ssh-agent`
Agent pid 272
% ssh-add
Enter passphrase for identity 'jp@laptop':
```

You can see the environment variables set by *ssh-agent* — `SSH_AUTH_SOCK` and `SSH_AGENT_PID` — by searching the output of *env*, which prints all current settings:

```
% env | grep SSH
SSH_AUTH_SOCK=/tmp/ssh-natVG242/agent.242
SSH_AGENT_PID=272
% ls -l /tmp/ssh-natVG242/
total 0
srwx—x—x 1 jpeek ... agent.242
```

The two environment variables, `SSH_AUTH_SOCK` and `SSH_AGENT_PID`, are propagated to all subprocesses of this shell. So you'll want to start the agent process as high as possible in your process tree, ideally before you start your window system. (You can run *ssh-add* after your window system is running.)

Later *ssh* or *scp* jobs will check their environment to locate the agent. If the two environment variables aren't set (weren't passed from the environment of the shell that started them), *ssh* and *scp* will prompt for a passphrase.

If you don't have an easy way to start the agent before your window system, start it from a shell prompt in one window and save its output (the environment-setting commands) to a temporary setup file. Next, from all windows where you want to use *ssh*, read the commands from the setup file into the shell using the command *source* for C-like shells or `.` (a dot) for Bourne shells. (Using the option `-c` for C shells or `-s` for Bourne shells makes sure *ssh-agent* uses the right syntax.) So, on the first shell, you'd do:

```
% ssh-agent -c > ~/ssh-agent-setup
% source ~/ssh-agent-setup
Agent pid 272
% ssh-add
```

Then, in other shells, simply repeat the *source* command to set the variables in those shells.

In this case, the local shell strips off the quoting characters, ignoring the special meaning of whatever's between them, and passes that string (`ls -F > bar-ls`) to the local *ssh*. The local *ssh* passes that same string to the remote *ssh*, which passes it to the remote shell. The remote shell parses the command string and executes it. Because the > isn't quoted by the time it reaches the remote host, it's interpreted as the redirection operator. So the file *bar-ls* is created in the remote shell's current directory, which, for *ssh*, is the home directory of the remote account.

Got it? (If not, please re-read the previous section. The concepts are important.) Otherwise, great! Now the fun starts.

Example: Copying Files via *ssh*

Let's get back to our earlier problem: how to copy a directory tree, working around *scp*'s weaknesses.

The *tar* utility is made for reading a directory tree and packing it into an archive file — or into a stream of bits. The *tar* flag `f` tells it to write or read the archive from a file. If that filename is `–`, *tar* writes to its standard output or reads from its standard input.

ssh can run almost any command line on a remote system

Let's start by running *tar* on the *remote* host and redirecting its output to a file on the *local* host. This is similar to the `ls -F` command we ran earlier. (Here we'll quote the remote command line, but for no reason except to make it stand out more clearly. There are no special characters that the local shell shouldn't interpret.) Once we have our local *bar-bin.tar* file, we'll run a local *tar* to unpack the archive.

```
% ssh bar 'tar cf – bin' > bar-bin.tar
% tar xvf bar-bin.tar
 ...verbose output of tar...
```

Ultimately, we don't need the intermediate file `bar-bin.tar` because *tar* can also read an archive stream from standard input. How do you feed the standard output of a process to the standard input of another process? Use a pipe, the shell's `|` operator, like this:

```
% ssh bar 'tar cf – bin' | tar xvf -
 ...verbose output of tar...
```

Again, you might ask yourself, "Which process is running where?" The answer: the remote *ssh* is running `tar cf – bin` and feeding its output (the archive stream) to the local *ssh*. The standard output of the local *ssh* is piped to `tar xvf –`, which is running on the local host, reading the archive stream from its standard input, and (in this case) writing it to the local filesystem. After this command runs, a copy of the entire remote *bin* directory tree appears on the local host.

LISTING ONE: Compressing an *ssh* data stream

```
% ssh bar 'tar cf – bin | bzip2' | bzcat | tar xvf –
 ...verbose output of tar...
```

By default, *tar* doesn't compress its archive. This means our file copy takes more time and bandwidth than it otherwise needs to. So we'll use *bzip2* (a cousin of *gzip*) to compress the archive stream before it leaves the remote host and then use *bzcat* to expand it on the local host and pass it to our local *tar*. The command line to use is shown in *Listing One*.

The remote *ssh* starts a shell which runs the command `tar cf – bin | bzip2`. The last two `|` characters aren't quoted, so the local shell makes two pipes: one feeds the output of the local *ssh* to a local *bzcat* process, which expands the compressed archive stream. Using the next pipe, the output of the local *bzcat* process is piped to the local *tar* process, which reads and stores the expanded archive bits.

Let's wrap up with a few more examples:

```
1% scp "bar:lib/[a-c]*" .
2% ssh bar cat bigf | less
3% ssh bar gzip -c bigf | zcat | less
4% scp bar:bigf bigf.bar
5% ssh bar gzip -c bigf | zcat > bigf.bar
6% gzip -c hugef | ssh bar 'zcat > hugef.rem'
7% ssh bar lpr -Pinkjet_jpeg < Kristy.jpg
```

➤ Command 1 copies all files whose names start with `a`, `b`, and `c` from the `lib` directory on `bar` to the local current directory.

➤ Command 2 runs `cat bigf` on `bar` and pipes it to a local *less* pager, which displays the file in your terminal. Command 3 is like command 2, but it compresses the data on the remote host and uncompresses it locally before displaying it.

➤ Command 4 copies `bigf` from `bar` to a local file named `bigf.bar`; Command 5 does the same things but compresses the file to save bandwidth.

➤ Command 6 copies the local file `hugef` to a file named `hugef.rem` on `bar`. The quotes mean that the > redirection is done on *bar*.

➤ Command 7 uses the shell's < redirection operator to feed a JPEG photo to the standard input of your local *ssh*. The remote *ssh* starts `lpr -Pinkjet_jpeg` which, because it doesn't have any other arguments, reads standard input (which has the photo to print).

X Tunnels Through *ssh*

We've seen that *ssh* can do more than file transfers. It can run any program on the remote host, with one class of exceptions: you can't run interactive, full-screen programs like *less* or *vi* with a command like `ssh baz vi somefile` because the remote *ssh* process isn't running in a *tty* or *pty*.

However, you can do an interactive login to a remote host, with a simple command like `ssh baz`. This starts an interactive shell, in a *pty*, on the remote host.

ssh can also run an X application on a remote host, with the display appearing on your local X host. Better yet, *ssh* can tunnel the otherwise insecure X protocol (meaning prying eyes can see the raw X traffic) through a secure, encrypted connection and handle all the messy details of authentication, too. This is called *X forwarding*.

Using X forwarding is simple: just run the X application like any other *ssh* command line. Type `ssh`, the hostname, and the name and arguments of the X application. You may need to give an absolute pathname (like */usr/X11R6/bin/xterm*) if the X application is in a directory that the remote *shell* doesn't know about. (Hint: because the remote *ssh* starts a non-login shell, you can try setting the remote shell's search path in a non-login setup file like *.bashrc* or *.tcshrc*.)

Try this first simple X command to display the remote host's load average graph on your local display. (Choose a remote host that has X installed.)

```
% ssh hostname xload
```

If you get the error `xload: command not found`, try `/usr/X11R6/bin/xload` instead. For other *ssh* problems, add the *ssh* option `-v` to get verbose output. If that still doesn't help, see a good *ssh* reference — like O'Reilly's book *SSH, The Secure Shell*. If all's well, though, a little *xload* window should pop up within a few seconds. Use CTRL-C to kill *ssh*. The *xload* window should close a moment later.

Here are a few more commands to try. These use the shell's `&` (ampersand) operator to run the *ssh* process in the background. This lets you open multiple remote processes from the same local terminal. When you exit the remote process, the local *ssh* process exits too. (Job control commands, like `kill %3` to kill the third job, should work too, though they may not give the remote application time to quit cleanly.)

```
8% ssh baz /usr/local/bin/mozilla &
9% ssh baz xterm &
10% ssh baz xterm -e vi afile &
11% xterm -e ssh baz &
```

➤ Command 8 starts the Mozilla web browser from host `baz`, with all the local customizations you've made there. (Mozilla's release notes mention that Shockwave plug-ins before Version 6 may not work on remote X displays.)

ssh can tunnel insecure X protocal through a secure connection

➤ Command 9 starts *xterm* running on the remote host.

➤ Command 10 starts a remote *vi* running inside an *xterm*.

Both commands 9 and 10 send all of the graphical traffic across the network. It's probably more efficient to start the *xterm* locally and have it open an interactive shell, as in command number 11, where you can then type the *vi* `command`.

Part II

Next month's column will cover ways to automate file transfers with *ftp*, *rsync* and friends. See you then!

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: Quick Returns

Do you need to go to some directory to run a command or two, then come back to your original directory? In *bash* and some other shells, the command `cd -` changes to your previous directory. Another `cd -` brings you back.

A similar trick that works in all shells is a subshell. Because a subprocess can't change its parent's environment, the subshell can *cd* but, when it exits, its parent shell has the same current directory. Here's an example:

```
$ pwd
/somedir
$ (cd /longpath; ./configure; make)
...configure and make run...
$ pwd
/somedir
```

Here's another example: this command creates an alias that *cd*s to another directory and runs commands there without changing the current directory in your shell:

```
alias setistat '(cd /tmp/seti; ps `cat
pid.sah`; grep "prog=" state.sah)'
```