Technical Explorations

by Keith R. Bennett

Copying (RVM) Data Between Hosts Using ssh, scp, and netcat

Dec 29, 2012 • keithrbennett

Occasionally I need to copy nontrivial amounts of data from one machine to another. I describe in this article three approaches to doing this on the command line, and explain why the third, using ssh and tar, is the best one.

As test data, I decided to use RVM's hidden control directory, `~/.rvm`. I deleted my non-MRI 1.9 rubies to reduce the transfer size.

I haven't tested this, but I imagine that for installing rvm on multiple similar systems (e.g. those with compatible native compilers, libraries, etc.), it may be possible to save a lot of time by a full install of rubies and gems on only one machine, then on the others doing a minimal install of rvm and then copying the fully populated .rvm directory.

Not So Good – Using scp

Note: This approach requires that the ssh port (22) be open on the destination host, and sshd is running. On the Mac, this is done by enabling "Remote Login" in the Sharing Preferences.

A very simple way to do this is to use `scp` (secure copy, over ssh) with the `-r` (recursive) option. For example:

```
>scp -r source_spec destination_spec
```

…where source_spec and destination_spec can be local or remote file or directory specifications. (I'll use the term *filespec* to refer to both.) Remote filespecs should be in the format *user@host:filespec*. Don't forget the colon, or the copy will be saved to the local host with a strange name! Here is an example that works correctly:

```
>#  To create ~/.rvm on the destination:
>time scp -rq ~/.rvm kbennett@destination_host:~/temp/rvm-copy/using-scp/
Password:
scp -rq ~/.rvm kbennett@destination_host:~/temp/rvm-copy/using-scp/  25.38s user 40.99s system 3% cpu 31:12.66 total
```

When I tried this, I was astonished to see that the destination directory consumed more than twice as much space as the original! To easily get the amount of space consumed by a directory tree, with the size in human readable format, run `du -sh directory_name`. For example:

```
>#  At the source:
>du -sh .
427M    .

#  At the destination:
>du -sh .
1.1G    .
```

After some investigation I found that it was due to the fact that scp "follows" symbolic links. That is, instead of creating a symbolic link on the destination, it makes a copy of the linked-to directory with the link name as the name of the containing directory.

For example, here are the ~/.rvm/rubies listings on the two machines:

```
>#  Source
>ls -l
lrwxr-xr-x  1 keithb  keithb   41 Dec 28 21:45 1.9 -> /Users/keithb/.rvm/rubies/ruby-1.9.3-p362
lrwxr-xr-x  1 keithb  keithb   41 Dec 28 21:45 default -> /Users/keithb/.rvm/rubies/ruby-1.9.3-p362
drwxr-xr-x  8 keithb  keithb  272 Dec 26 16:51 ruby-1.9.3-p362

#  Destination
>ls -l
total 0
drwxr-xr-x  8 kbennett  staff  272 Dec 28 22:12 1.9
drwxr-xr-x  8 kbennett  staff  272 Dec 28 22:19 default
drwxr-xr-x  8 kbennett  staff  272 Dec 28 22:24 ruby-1.9.3-p362
```

Entries with the letter "l" in the leftmost position of the line are symbolic links, and their listings show the "real" directories to which they refer. In this case, those links were created by the first two rvm commands below:

```
>>rvm alias create 1.9 1.9.3-p362
Creating alias 1.9 for ruby-1.9.3-p362.
Recording alias 1.9 for ruby-1.9.3-p362.

>rvm --default 1.9

>rvm alias list
1.9 => ruby-1.9.3-p362
default => ruby-1.9.3-p362
```

Because scp ignores the alias' symbolic link status, all the links above have a "*d*" signifying a real directory in the destination listing rather than the "*l*" signifying a link. After doing a little research I could not find any way to change this scp behavior. That being the case, I decided this approach was unacceptable.

Better – Using tar with nc

nc, also known as *netcat*, is a utility that, at its simplest, merely simulates *cat* over a network. That is, nc reads from sockets and writes to sockets where cat reads from stdin and writes to stdout. Execute `man nc` to find out more about it. Combining the netcat and tar commands on both the source and destination, you can:

On the source machine: create a tar file and send it to the destination

On the destination machine: receive the tar file and extract it, writing the files to the file system

Otherwise put, the flow of the data looks like this:

tar (source) -> nc (source) -> nc (destination) -> tar (destination)

One great thing that may not be readily apparent is that, thanks to Unix, all the component subtasks can be working simultaneously. The tar file is never really a static file, but rather a stream. One side benefit of this is that there is no need for disk space to accommodate the tar file on either end. This makes it possible to copy full partitions.

Here are the actual commands I used:

On the destination machine, with the current directory set to where I want the files to be written:

```
>nc -l 12345 | tar xzf -
```

Some notes on this command:

- the "-l" tells netcat to listen on the specified port (12345 in this case)
- tar's "x" option tells tar to extract
- tar's "z" option tells tar to (un)compress
- tar's "f -" option says "I'm specifying the path, and it's stdin" (the hyphen is used by convention to mean stdin or stdout)

On the sending side, we execute this command:

```
>>time tar czf - . | nc destination_host 12345
tar czf - .  21.05s user 1.47s system 9% cpu 4:07.18 total
nc destination_host 12345  0.09s user 1.04s system 0% cpu 4:07.20 total
```

- tar's "c" option tells tar to create an archive
- nc's options specify a destination host whose name is `destination_host`, on port 12345

This approach results in the correct handling of symbolic links. In addition, it's about 7.5 times faster, partly due to the data compression, but probably mostly because only one file (the tar file) needs to be handled by the transfer protocol, rather than tens of thousands.

Best – Using tar with ssh

Note: This approach assumes that the source host's ssh port is open and sshd is running.

Although the previous approach was a great improvement, there are two potential issues with it:

- it requires access to shells on both hosts
- although the data is compressed, it is not encrypted

After some web searching, I found this excellent solution posted at http://www.linuxquestions.org/questions/linux-general-1/recursive-scp-w-o-following-links-658857/.

It takes advantage of ssh's ability to execute a remote command and send its output to the local host's stdout. This makes it possible to pipe the output to a local command, tar in this case:

```
>>time ssh keithb@source_host "cd ~/.rvm; tar czf - ." | tar xzf -
Password:
ssh keithb@source_host "cd ~/.rvm; tar czf - ."  2.31s user 0.75s system 7% cpu 38.676 total
tar xzf -  2.01s user 6.74s system 22% cpu 38.675 total
```

This is over six times faster then the nc approach and 48 times faster than scp! In addition, it is a single command executed from one host, and travels over an encrypted connection. Sweet!

Published with GitHub Pages