



# Speaking UNIX, Part 5: Data, data everywhere

Move and manage files stored across multiple systems

Level: Intermediate

[Martin Streicher \(martin.streicher@gmail.com\)](mailto:martin.streicher@gmail.com), Chief Technology Officer, McClatchy Interactive

28 Nov 2006

Take a look at several techniques that illustrate how to move files among systems and how to keep such far-flung data in sync.

In recent years, computer hardware has become ridiculously inexpensive. A gigabyte of hard disk space costs US\$0.50, a 19-inch flat-panel display for less than US\$200, and a laptop that can run UNIX® costs less than US\$1000. Even specialized server hardware is priced as a commodity.

At such low, low prices, it's quite affordable for a medium- or large-sized organization to dedicate individual pieces of hardware to unique tasks. Moreover, expanding compute capacity can be as easy as connecting the machine to the network, copying a disk image to a new *white box* (a generic computer), and restarting. Of course, there's no free lunch. Every computer requires a healthy diet of electricity, cool air, and love and affection.

One of the most common problems of managing large numbers of computers is how to keep so many systems up-to-date and consistent. In some cases, you want the same version of an application deployed everywhere, lest users be confused by inconsistent idiosyncrasies. Or, as another example, you probably want the same operating system on each server that serves the same purpose. Predictability is good.

Even if you have only a laptop and a desktop computer, keeping just those two machines in sync can be a daunting task. Today, you're on the road working on your portable computer. Yesterday, you tinkered from your desktop. On both days, perhaps you uploaded or downloaded files from a central file server. With data going and coming, you can quickly become confused about what's where.

In Part 5 of this series, let's look at a handful of techniques that can help keep explosions of files under control.

## Tarred and forwarded

Obviously, the simplest way to maintain consistent data across multiple computers is to simply keep your files (a spreadsheet, database, text files, and so on) with you. If you had physical access to every machine you use, you could carry a portable disk, CD-RW, or large flash memory *keychain* and simply plug in the storage device whenever you needed your files.

However, if a machine you use is remote or inaccessible, say, in your company's machine room in Boise, Idaho, connecting the peripheral device isn't an option. Instead, you can make an archive of (some or all) your files, connect to your local area network (LAN) or wide area network (WAN), copy the archive to its destination, and restore the files to continue your work. (Moreover, you can use the archive as a simple backup to protect your files in the event of disaster, such as leaving your computer in a taxi.)

On UNIX systems, the stalwart utility `tar` makes light work of building archives. The `tar` utility bundles one or more files and directories into a single file, maintaining bytes, owners, permissions, file type, and station in the file system hierarchy of the original files. The `tar` utility records your files verbatim on tape -- `tar` is definitely an acronym for *tape archiver*.

For example, assume that you have a directory of miscellaneous files, as shown in [Listing 1](#).

### Listing 1. File directory

```
$ cd stuff
$ ls -lR
.:
drwxr-xr-x  2 mstreicher mstreicher  4096 Oct 12 19:11 css
-rwxr-xr-x  1 mstreicher mstreicher    91 Aug 17  2005 demo.rb
-rw-r--r--  1 mstreicher mstreicher 111563 Oct 12 19:10 tech.pdf

./css:
total 16
-rw-r--r--  1 mstreicher mstreicher   711 Mar 25  2006 style.css
-rw-r--r--  1 mstreicher mstreicher 11353 Apr 10  2006 valid.css
```

To create an archive of the two files and the directory, run **tar**:

```
$ tar --create --verbose --gzip --file archive.tgz *
css/
css/style.css
css/valid.css
demo.rb
tech.pdf
```

The **--create** option tells **tar** to create an archive; the **--verbose** option generates a list of files that **tar** has processed; the **--gzip** option enables *gzip*-style compression, which shrinks the archive; and **--file archive.tgz** specifies the name of the archive.

The shell interprets the asterisk (\*) as "any file" and, therefore, expands it to name the two files and directory. As you can tell from the output immediately above, **tar** archives the **css** directory and recurses to archive that directory's contents.

After running **tar**, the current directory contains a new file, **archive.tgz**:

```
$ ls -l archive.tgz
-rw-r--r--  1 mstreicher mstreicher 105470 Oct 13 17:16 archive.tgz
```

You can now copy **archive.tgz** to another computer and use **tar** on the remote computer to extract what was previously archived. In fact, the command line to restore the files is almost identical to the previous command line. To extract the archive, use:

```
$ tar --extract --verbose --gunzip --preserve-permissions --file archive.tgz
```

This **tar** command extracts the contents of the **archive.tgz** tarball. The **--extract** option is the opposite of the **--create** option; **--gunzip** is the inverse of **--gzip**, and **--preserve-permissions** recreates the permissions of the original files.

After running this command, the files you saved are restored intact, preserving the time stamp, permissions, and file name. Also, the directory named **css** is recreated with its contents extracted in situ.

The **tar** utility has a multitude of options: **--create**, **--extract**, and **--list** (which catalogs the **.tar** file without expanding it). Other options (such as **--gzip** and **--preserve-permissions**) control how **tar** creates the archive. See the **tar** man page for your version of UNIX for more details and the proper syntax for each option.

Creating an archive, copying it to its destination, and extracting it is useful, but it can become laborious. Additionally, if the archive is extremely large, you might not be able to store both the archive and its expanded files. To save time and, if the source and destination computers are connected by a LAN or WAN, you can combine the Secure Shell, SSH, and **tar** to archive, copy, and extract the files in one fell swoop. Here's how:

```
$ (cd ~/stuff; tar --create --gzip --file - *) | \
ssh destination tar --extract --gunzip --file --verbose -C stuff
```

There's a lot going on in that command, so let's decompose it:

1. The parenthesized series of commands is called a *subshell*. Changes made in the subshell -- for example, changing directory -- do not affect your command line, but it does affect the environment of the subshell. Hence, the first phrase, `(cd ~/stuff; tar --create --gzip --file - *)`, changes to the directory `~/stuff`, and then runs `tar`. Because the subshell is followed by a pipe, all the output of the subshell is sent to the next command in the pipeline.
2. Like many other UNIX utilities, `tar` can write to and read from standard output (`stdout`) and standard input (`stdin`), respectively. Both `stdout` and `stdin` are typically denoted as a hyphen (-). So, the phrase `--create --file -` creates the archive on `stdout`.
3. The pipe (`|`) pipes all output of the subshell to `ssh`. This effectively transfers all the output from the source computer to the destination computer.
4. Finally, the destination machine runs its own instance of `tar` to extract the archive. Here, however, `--extract --file -` reads the archive from *standard input*. The `-C` option forces the *receiving* `tar` to change directory to `stuff` (in your remote home directory) before it begins any processing. The end result is that the archive transmitted through `ssh` is unpacked in `~/stuff`.

In one (somewhat lengthy) command, you archived, transferred, and extracted your archive. By the way, a near transpose of the command line allows you to fetch and extract an archive created on the remote computer to your local computer. Here is that solution, run from the local machine:

```
$ ssh destination cat archive.tgz | \
  (cd ~/stuff; tar --extract --gunzip --file -)
```

The remote archive is opened on the remote machine, and the byte stream from `cat` is sent to a subshell that first changes directory, and then it extracts the archive. (Adding `-C ~/stuff` to the `tar` command achieves the same effect; the example just shows that subshells can consume input, too.)

You can use the same technique to mirror files on the same machine. Try something like:

```
tar --create --file - * | tar -C /path/to/directory --extract --file -
```

## Copying is the sincerest form of flattery

The `tar` utility with `ssh` is a convenient method for transferring files from one machine to another. The `tar` utility creates the archive, and `ssh` facilitates secure transfer of the archive.

Another technique is to use the innate abilities of the SSH to transfer files from one machine to another. `sftp`, another "personality" of SSH, provides all the features of the File Transfer Protocol (FTP), yet protects file data while in transit. (In general, the use of FTP is frowned upon, because it is insecure; however, public FTP sites are one significant exception to the rule.)

If you've ever used FTP, `sftp` is virtually identical. Simply type `sftp destination` to connect to the remote machine named *destination*, and run FTP commands such as `cd`, `lcd`, `mput`, and `mget` to move files back and forth.

Yet another way to transfer files between two machines is to use `scp`, or *secure copy*. As its name implies, `scp` works much like plain old `cp`: It copies files from one place to another, either on the same machine or between two machines.

For example, if you wanted to copy some files and directories to another directory on your local machine, you'd run something like the code shown in [Listing 2](#).

**Listing 2. Copy files between two machines**

```
$ ls -lF
drwxr-xr-x  2 mstreicher mstreicher   4096 Oct 12 19:11 css/
-rwxr-xr-x  1 mstreicher mstreicher     91 Aug 17  2005 demo.rb*
-rw-r--r--  1 mstreicher mstreicher 111563 Oct 12 19:10 tech.pdf
$ cp -pr * /home/joe/stuff
$ ls -lF /home/joe/stuff
drwxr-xr-x  2 mstreicher mstreicher   4096 Oct 12 19:11 css/
-rwxr-xr-x  1 mstreicher mstreicher     91 Aug 17  2005 demo.rb*
-rw-r--r--  1 mstreicher mstreicher 111563 Oct 12 19:10 tech.pdf
```

In this example, **cp -pr** recursively copies all files and directories to `/home/joe/stuff`. The **-r** causes recursion; the **-p** preserves the time stamps of the files.

You can do the exact same thing (that is, copy locally) with **scp**:

```
$ scp -pr * /home/joe/stuff
```

But if you specify a remote system, **SCP** copies the files over the network:

```
$ scp -pr * destination:/home/joe/stuff
```

Assuming that `/home/joe/stuff` exists on the destination machine and is writable by you, the two files and the directory are copied verbatim to the remote machine. Like **cp**, **SCP** recognizes **-p** for preserve and **-r** for recurse.

**scp** is easy to use, especially if you establish a private-public key pair to avoid typing your password for each **ssh/scp/sftp** operation.

However, **scp** does have one peculiarity to be aware of. Assume that you have a directory named `doc` in your home directory and you want to copy it to a remote system. Furthermore, you want the contents of `~/doc` to replace the contents of the remote `doc` directory whenever a file or directory have the same name. The command to use would be something like this:

```
$ scp -pr ~/doc destination:/path/to/doc
```

Notice that the path on the destination machine lacks a trailing slash (`/`). **SCP** interprets the path as "copy the contents of the `~/doc` directory to the directory `/path/to/doc` on the destination machine." As with **cp**, remote files and directories that have the same name as the local files and directories are overwritten; unique files on the remote system are left untouched.

If you add a trailing slash, however, as in:

```
$ scp -pr ~/doc destination:/path/to/doc/
```

**scp** interprets the latter path as "copy the directory `~/doc` to the directory `/path/to/doc/`." So, instead of overwriting the contents of the remote directory, the local `doc` directory is copied into the remote directory.

The trailing slash is not an error. At times, you might want to use it; at other times, you might not, depending on your intent.

**Keeping in sync**

`scp` is exceedingly useful, because it mimics `cp` so closely. `tar` and `ssh` are slightly more complex, but they preserve file metadata, such as owner and permissions.

But both `tar` and `scp` fail to synchronize the contents of the local and remote directories. For example, if you changed one file on the local system and another and a different file on the remote system, you'd have to run two `scp` commands to make a working mirror. Now imagine that you have handfuls of changed files, many of them named identically. Very quickly, you can see just how complicated synchronization can become.

Luckily, there's an amazing utility called `rsync` that synchronizes sets of files. Better yet, `rsync` transfers only the data that's changed, minimizing the amount of data transferred.

Like `tar`, you can combine `rsync` with `ssh` to connect to remote systems and synchronize a local and remote collection of files. Like `scp`, you can use `rsync` to copy files locally. You can also use `rsync` to list files.

And best of all, `rsync` has options to make one directory a true mirror of another directory, using options to delete files that don't exist in the original directory. Let's look at some examples:

```
$ rsync -e ssh --times *.txt destination:
```

This command copies all text files in the current working directory to your home directory on the machine named *destination*. The `-times` option preserves the access, creation, and last modified time of each file.

```
$ rsync -e ssh --times --perms --recursive --delete doc destination:
```

This variation of `rsync` mirrors the local `doc` directory in your home directory on *destination*. File times are maintained, as are permissions, and extraneous files (that is, files in the remote directory that do not exist within the local directory) are removed.

Because `rsync` can make some significant changes, you might prefer to add the `--dry-run` option to the command line to preview what `rsync` plans to do. `--dry-run` does not make any changes -- it merely shows you what happens, as shown in [Listing 3](#) below.

### Listing 3. Preview what rsync does

```
$ rsync -e ssh --dry-run --times --perms --recursive --delete bin destination:
building file list ... done
bin/
skipping non-regular file "bin/HTML.pl"
skipping non-regular file "bin/Quark.pl"
bin/Responses/
bin/Responses/DBI.pm
bin/Responses/Response.pm
skipping non-regular file "bin/XML.pl"
bin/backupdca.sh
bin/lib/
bin/report.pl
bin/report.txt

sent 724 bytes  received 108 bytes  554.67 bytes/sec
total size is 168879  speedup is 202.98
```

`rsync` has a multitude of options:

- **-a** is invaluable, as it's shorthand for `--group --owner --perms --times --devices --links --recursive`. `--devices` recreates device files, and `--links` copies symbolic links as symbolic links rather than copying what the symbolic link points to.
- **--update** prevents `rsync` from overwriting newer files. If the remote system has a newer file than the local system, the remote system's file is retained.
- Try **--verbose** to watch `rsync` in action.

Again, read the man page for `rsync` to learn more of its tricks. One significant feature specifically includes or excludes files according to criteria you define.

---

## Data, data everywhere

UNIX has been used in networked environments for more than 20 years. In that time, the hardware has changed dramatically, but much of the software remains the same, as do the challenges for users and system administrators. One of the biggest problems, keeping track of all the data, grows worse as disk capacity grows to gargantuan sizes. Utilities such as `tar`, `sftp/scp`, and `rsync` can tame even the most savage disk.

Part 6 of this series takes a look at automation -- yet another way to save time and effort and reduce human error.

## Resources

### Learn

- [Speaking UNIX](#): Check out other parts in this series.
- [zsh Mailing List Archive](#): Read this mailing list to learn more Z shell tricks and tips.
- [AIX and UNIX](#): Visit the developerWorks AIX and UNIX zone to expand your UNIX skills.
- [New to AIX and UNIX](#): Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [AIX 5L Wiki](#): A collaborative environment for technical information related to AIX.
- [Technology bookstore](#): Browse this site for books and other technical topics.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

### Get products and technologies

- [Z shell](#): Download the latest version of Z shell from the [Z shell home page](#).
- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

### Discuss

- [zsh](#): Collaborate, discuss, and share your zsh expertise on the zsh wiki.
- Participate in the AIX and UNIX forums:
  - [AIX 5L -- technical](#)
  - [AIX for Developers Forum](#)
  - [Cluster Systems Management](#)
  - [IBM Support Assistant](#)
  - [Performance Tools -- technical](#)
  - [Virtualization -- technical](#)

- [More AIX and UNIX forums](#)
- Participate in the [developerWorks blogs](#) and get involved in the developerWorks community.

## About the author



Martin Streicher is the Chief Technology Officer of McClatchy Interactive and the Editor-in-Chief of *[Linux Magazine](#)*. Martin holds a Masters of Science degree in computer science from Purdue University and has been programming UNIX-like systems since 1986. You can reach Martin at [martin.streicher@gmail.com](mailto:martin.streicher@gmail.com).

## Share this....



[Digg this story](#)



[del.icio.us](#)



[Slashdot it!](#)

UNIX is a registered trademark of The Open Group in the United States and other countries. Other company, product, or service names may be trademarks or service marks of others.