Transfer Tips, Part II

By Jerry Peek

Last month's column (available online at http://www.linux-mag.com/2003-03/power_01.html) presented a number of ways to transfer files by hand with *ssh* and *scp* and the power of the shell. This month's column looks at ways to transfer and synchronize sets of files *automatically* with *ftp*, *sftp*, and *wget*. Along the way, you'll also see tips on compressed *ssh* and detached processes. Let's dig in!

Transfer Tips Part I, Continued

First, a note about last month's column, where we used *gzip* and *bzip2* to compress data transferred through an *ssh* connection. Separate compression and decompression steps are often unnecessary because many *ssh* servers have compression built-in.

For example, if the *ssh* server on host *foo* supports compression, the command `scp -C afile foo:` compresses *afile*, transfers the compressed (intermediate) file to *foo*, and then uncompresses it into your home directory on *foo*. The `-C` option enables compression.

In the same way, the command `ssh -C foo` compresses all traffic traversing the interactive connection to *foo*. In fact, compression even works with X *forwarding* (tunneling X Window System connections through *ssh*), boosting the performance of tunneled X applications.

ssh compression is most useful over slow network connections like dialup modems. However, if you're transferring a lot of small files over a slow link, network latency can defeat the time savings from `scp -C`. That's because each file has to be negotiated and sent separately — and that "overhead" time can build up. Instead, *pipeline* the transfer — package all of the files into a single *tar* archive, and compress/decompress the *tar* file on-the-fly (with `tar -z` or `ssh -C`). And be careful using *ssh* compression over fast links. On a fast link, the CPU time needed to compress and decompress the data could make the transfer slower!

If you want to experiment with *ssh*, the *time* command can help. For example, try sending a big text file from your machine to some host *foo* with the command `time scp -C /usr/dict/words foo:/tmp`. Then do it again without the `-C` option.

For interactive logins, try emitting a lot of text, like `head -5000 /usr/dict/words | fmt` and compare the scrolling speed in compressed and uncompressed sessions.

By the way, if the *ssh* server you're connecting to doesn't support compression, you'll get a warning that says, "`Remote host refused compression.`"

ftp Revisited

Web browsers and newer utilities like *wget* and cURL have mostly superseded the old *ftp* utility. But *ftp* does transfers interactively, which lets you control its actions. Unlike many FTP protocol-enabled programs, *ftp* can also send files *to* an FTP server and even has a little-known configuration file that stores macros and enables automatic logins.

Unfortunately, the FTP protocol is insecure: your password and your data are transmitted in the clear. Traditional FTP is best used for transfers on your internal network or from anonymous-login sites on the Internet. While *ftp* is insecure, a similar command, *sftp*, performs FTP-like transfers over an encrypted SSH transport. It also supports the *ssh* `-C` option for compressed transfers. However, the remote host has to accept *sftp* connections; *sftp* can't encrypt connections to hosts that have only standard FTP servers.

Your system may have a newer client named *ncftp*. In fact, some sites have replaced *ftp* with *ncftp*. We're covering old *ftp* here, though, because it's widely available (even on non-Unix-like systems).

LISTING ONE: An interactive *ftp* session

```
$ ftp ftp.mycorp.biz
 ...
Name (ftp.mycorp.biz:jpeek): anonymous
331 Use your e-mail address as password.
Password:
ftp> cd /pub/foo/bar
ftp> dir
-rw-r—r— ... 4378 Sep 8 2000 README
-rw-r—r— ... 18312 Sep 7 2000 xyz1.rm
-rw-r—r— ... 88115 Sep 7 2000 xyz2.rm
 ...
ftp> lcd /tmp/foobar
Local directory now /tmp/foobar
ftp> prompt
Interactive mode off.
ftp> mget *.rm
 ...
226 BINARY Transfer complete
ftp> get README readme-xyz
local: readme-xyz remote: README
 ...
ftp> cd /incoming
ftp> put somefile
 ...
ftp> quit
$
```

Listing One shows an example *ftp* session, shortened and edited to fit the page (many of the status messages have been removed). We log in anonymously to the host `ftp.mycorp.biz`, get a prompt (`ftp>`), go to the remote directory `/pub/foo/bar` and the local directory `/tmp/foobar`, retrieve all files whose names end with `.rm`, then retrieve the file `README` and rename it `readme-xyz`. Finally, we upload a file to the server's `/incoming` directory.

The *ftp* client reads a setup file named *.netrc* in your home directory, which must be readable only by you (use `chmod 600 .netrc` to set the correct permissions). For each remote host you access, you can make a *.netrc* entry with the remote username, password (saved in clear text, so use it only for anonymous logins!), and a set of macros (named shortcuts for a series of *ftp* commands). The special macro *init* has commands that run each time you connect to the host.

Let's see a sample *.netrc* entry that runs the first few commands used in the previous example. (Although we've broken the first line for printing, all of it should be entered on one line.) You must place an empty line after each macro.

```
machine ftp.mycorp.biz login anonymous
  password jpeek@jpeek.com
macdef init
cd /pub/foo/bar
dir
lcd /tmp/foobar
prompt
```

You don't need to run *ftp* interactively, though. *ftp* reads commands from the standard input, and writes its prompts and results to standard output. That lets you generate commands on-the-fly — from a shell script, for instance — to drive *ftp* non-interactively.

The next example shows a fragment of a Bourne shell script that opens a connection to the host `$ftpto` (with help from a *.netrc* file). It writes four commands to *ftp*'s standard input, using the shell's "here-document" operator `<<`*`pattern`*. (A here-document feeds text from the script file, line by line, into *ftp*, until the shell finds the matching *`pattern`* on a line by itself. The standard output and standard error are merged (using the shell's operator `2>&1`) and written to the file named in `$logfile`:

```
#!/bin/sh
 ...
ftp $ftpto <<END_FTP_CMDS >$logfile 2>&1
ascii
verbose
put $xfrfile
dir $xfrfile
END_FTP_CMDS
```

The first command sets the `ascii` mode to do automatic end-of-line corrections for text files sent between different types of hosts (from a Linux box to an MS Windows system, for instance). When *ftp* isn't reading commands from a prompt, it uses its non-verbose mode; the second command enables `verbose` mode. The third command sends the file `$xfrfile`. The fourth command, `dir`, gets a listing like `ls -l`, which should show that `$xfrfile` has been written to the remote host. A `quit` command isn't needed; when *ftp*'s standard input ends, it quits.

Why Not *ftp*?

Running *ftp* interactively lets you see and control the transactions. Running *ftp* from a script is handy for short jobs, but it isn't robust. If something goes wrong, the script typically can't recover. It also doesn't scale well: writing a script with lots of `put` or `get` commands is tedious and not very efficient. The handy *Expect* utility (covered previously in the January 2001 issue, available online at http://www.linux-mag.com/ 2001-01/guru_01.html) can do a robust job of managing interactive processes such as *ftp*. *ftp*'s verbose status messages (like `226 BINARY Transfer complete`) make error-checking fairly easy for *Expect*.

Of course, there are other ways to transfer files, including *scp* and a program in a network-savvy scripting language like Perl. There's also *wget*, which is designed to do non-interactive FTP transfers.

What *wget* Does

The GNU *wget* utility gets files from the Web with the HTTP and HTTPS protocols. It also supports FTP.

For instance, the command `wget http://www.linux-mag.com/` gets a copy of the Linux Magazine home page, which is a file named *index.html*. By default, *wget* won't get other files that make a web page display properly, such as image files and stylesheets. The `–p` option makes *wget* analyze an HTML page and retrieve those extra files.

Although you can use *wget* interactively to get one file at a time, it's even handier when it retrieves files on its own. For instance, you can log into your office server from home, start a long transfer, then log out and let the transfer complete, as shown in the sidebar "Running a Detached Process" on page 59.

And *wget* does much more. For instance, it can retrieve all files from an FTP area or a web site, recursively. On an FTP site, that job isn't too tough. For a web site, though, HTML pages have to be parsed and interpreted. The links, which could point to other pages on any other server, may need to be followed. Even more complicated, local copies of HTML files

See Power Tools, *pg. 59*

Power Tools, *from pg. 39*

may need global changes so their links point to the local versions of linked-to files. That poses some questions: if image files are in a completely separate directory, should *wget* store them in the current directory or create a new directory? Should *wget* follow links from a web page to a parent or sibling directory or to another host, and should *wget* recursively retrieve from those places too? As it turns out, that's all configurable from the command line, as well as from two setup files: the system-wide *wgetrc* and the *.wgetrc* in your home directory. This helps to explain why *wget* has so many options! Those options and all the details, are in *wget*'s info pages. Type `info wget`. In this column, we'll see just a few examples.

Mirroring with *wget* and *cron*

wget can mirror files from another host — finding and retrieving only the remote files that differ from the local ones. For instance, *wget* can make a complete copy of the */pub* directory (which typically contains the data file tree) from the FTP server *ftp.mycorp.biz* and write it into your local directory */mirrors/ftp.mycorp.biz/pub*.

The next example shows the *wget* command to do just that, running nightly from *cron*. (The command must be entered on a single *crontab* line, not split as we've shown it here.)

```
12 2 * * * cd /mirrors; su ftp -c
  'wget —append-output=/var/log/mirroring
  —mirror ftp://ftp.mycorp.biz/pub/'
```

System administrators often run jobs from *root*'s *crontab* file, but running *wget* unattended as *root* can cause trouble. Unexpectedly-huge files may overfill the local filesystem, crackers can make symbolic links in the local filesystem that cause *wget* to overwrite system files in other directories, and more. To avoid those problems, we've used the command `su ftp –c 'wget ....'` so the *root cron* job runs *wget* as the user *ftp*.

One feature *wget* uses during mirroring is *time-stamping*. *wget* will transfer a remote file if it doesn't exist locally, or if it's newer than the local version, or if the remote and local files have different sizes. This is a good example of a task that's hard to manage with a Web browser or *ftp*.

Another advantage of a non-interactive command like *wget* is that you can manage it just like other Linux utilities: you can invoke *wget* from a script or from the command line. You could start the next example over a dialup connection, then leave home to go shopping while the transfer completes.

```
$ wget -r ftp://bar.com/pub/
```

cURL and *rsync* and Beyond

If you like *wget*, you may like cURL even more because it handles URLs for protocols other than HTTP, HTTPS, and FTP. Also, unlike *wget*, cURL can send files to a server. It's available from http://curl.haxx.se.

wget's mirroring is one-way: making your local files the same as remote files. It doesn't work in the other direction, though. However, you can do that with *rsync*, a non-interactive utility that we'll cover next month. *rsync* has a smart difference-finding algorithm, uses pipelining to make transfers efficient, and more. But, for now, happy transfers!

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: Googling Linux Magazine

The Linux Magazine website is packed with information. The online tables of contents give a good overview, but they can't possibly list every topic or program mentioned in every article. That's where a Google search comes in. By using Google's `site:` operator to restrict the search to linux-mag.com, you can avoid inferior `;-)` sources of information. For instance, to find information on the *Expect* utility, type the following search into the searchbox at http://www.google.com/.

```
expect Libes site:linux-mag.com
```

Adding the name of the author of Expect is a guess (which works!) to keep Google from matching every column that contains the generic word "expect."

RUNNING A DETACHED PROCESS

A process can run detached from its parent shell. This means the shell won't wait for the spawned process to finish or kill it when you log out.

Let's see how to run *wget* detached so you can log out and let it run. You could do this when you're logged in from home to your office — over a dialup modem, for instance — and you want *wget* to download a huge file that will be waiting on your office server in the morning:

```
mysys$ ssh server.mycorp.biz
server$ (wget -o wlog ftp://foo.com/hugef &)
server$ exit
Connection closed by remote host.
mysys$
```

If you use `ps -l` (lowercase "L") before logging out, you should see that `wget`'s *PPID* — its parent process ID — is 1 (the Linux *init* process), not the shell. Some *ssh* servers still won't let you log out after starting a detached process, though. In that case, try submitting an *at* job to start *wget* soon after you log out. (The February "Power Tools" column, available online at http://www.linux-mag.com/2003-02/power_01.html, describes *at*.)