Speaking UNIX: Managing multitudes of machines the mild-mannered way

Skill Level: Intermediate

Martin Streicher (martin.streicher@gmail.com)
Software Developer
Pixel, Byte, and Comma

14 Sep 2010

Cloud computing may currently be all the rage, but there is a silicon lining to each calculating cumulus: hardware and software require very real upkeep. Learn how to manage gobs of machines right from the command line.

A cloud is ultimately constituted from hardware and software—components that require real and regular upkeep. Hardware failures demand repair or replacement; software requires patching, updates, and upgrades; and systems must be configured to keep ahead of demand and potential security threats. Application developers may find compute clouds soft, fluffy, and convenient, but a cloud administrator faces the grimy truth.

You don't have to manage a cloud to face gritty issues. A LAN, a small server farm, and a compute cluster pose many of the same systems administration challenges. When the number of machines climbs, workaday tools such as Secure Shell (ssh), `scp`, and `sftp` become cumbersome. This installment of Speaking UNIX looks at effective techniques to manage gobs of machines from the command line, starting with just a handful of systems and scaling upward.

A brute force approach

A simple and obvious approach to running a command on a collection of machines wraps the common ssh utility in a script. Assuming that you have distributed your public key to each remote system you want to access (to avoid typing a password each time), this script, named *mssh.sh*, runs a single command on each machine specified and prints the collected results at the end (see Listing 1).

Listing 1. mssh.sh

```
#!/bin/bash
# Usage: mssh.sh "machine1 [machine2...]" "command"

OUTPUT_LOG=/tmp/output-$$.log
ERROR_LOG=/tmp/error-$$.log
MACHINES=$1; shift
COMMAND=$1; shift

for machine in $MACHINES
do
    ssh $machine $COMMAND >>$OUTPUT_LOG.$machine 2>>$ERROR_LOG.$machine &
done

wait

cat $OUTPUT_LOG.*
cat $ERROR_LOG.* >&2
rm -f $OUTPUT_LOG.* $ERROR_LOG.*
```

For example, the command `mssh.sh "example.com joe@sample.com" "uptime -a">` runs `uptime -a` on two hosts: example.com and sample.com. The list of machine names is quoted to lump the names together into one argument, and the command is quoted for exactly the same reason. Each machine name must conform to the paradigms for ssh—either *`hostname`* if the remote user name is the same as the local user name, or *`username@hostname`* if the remote user name differs from the local login. Running `mssh.sh "example.com joe@sample.com" "uptime -a">` produces something akin to this:

```
$ mssh.sh "example.com joe@sample.com" "uptime"
example.com
08:34:35 up 66 days, 17:29,  0 users,  load average: 0.40,
0.19, 0.07
joe@sample.com
08:34:28 up 104 days, 10:18,  0 users,  load average:
0.15, 0.10, 0.10
```

This script is rudimentary but can be extended to include other features, such as a tunable timeout to prevent interminable delays when a single host is down (look at the `ssh -o` option) and a named directory to capture output. Indeed, many packages build on the spirit of this script to simplify distributed systems administration. One of those is the Distributed Shell (dsh).

A better tool for the task

Specifically designed to run shell commands on remote systems, dsh provides a handful of conveniences to make working with groups of machines easier. The shell is available in both binary and source form. For binaries, consult your Linux® or UNIX® distribution for the `libdshconfig` and `dsh` packages. For example, Ubuntu and Debian users can install dsh in one fell swoop with `apt-get`:

```
$ sudo apt-get install libdshconfig1 libdshconfig1-dev dsh
```

If you cannot find pre-built packages for your system, dsh is readily built from source code. Find the latest version of both the library and the utility, download and unpack both tarballs, then build and install both with the typical technique of `./configure; make; sudo make install` (see Listing 2).

Listing 2. Building dsh from source

```
$ # Build and install the library first
$ wget http://www.netfort.gr.jp/~dancer/software/downloads/libdshconfig-0.20.13.tar.gz
$ tar xzvf libdshconfig-0.20.13.tar.gz
$ cd libshconfig-0.20.13
$ ./configure
$ make
$ sudo make install

$ # Then build and install the utility
$ wget http://www.netfort.gr.jp/~dancer/software/downloads/dsh-0.25.9.tar.gz
$ tar xzvf dsh-0.25.9.tar.gz
$ cd dsh-0.25.9
$ ./configure
$ make
$ sudo make install
```

The shell is a fairly small application; the `dsh` and `dsh.conf` man pages provide all the details required to master the utility. For instance, to run `uptime` across a set of hosts, as in the first example, you simply type:

```
$ dsh --show-machine-names -m example.com -m joe@sample.com -- uptime
example.com: 11:34:57 up 66 days, 20:29,  0 users,  load average: 0.04, 0.06, 0.01
joe@sample.com: 11:35  up 2 days, 14:59, 8 users, load averages: 0.46 0.35 0.31
```

You specify a machine with `-m`, and host names follow the same rules as ssh. The two dashes in the command line separate options for the `dsh` command itself from the command to run. Output appears in the order the machines are named. The command `--show-machine-names` prepends each machine name to whatever is emitted from the remote command.

If you tend to work with the same set or subset of machines, you can define one or more collections and specify a collection to operate on. You can create one global collection and any number of groups. The file $HOME/.dsh/machines.list is the global collection. If you specify `dsh -a`, the given command runs on all machines listed in machines.list. Hence, if machines.list contained:

```
example.com
joe@sample.com
```

the command:

```
dsh -a --show-machine-names -- uptime
```

would produce the same output as the previous command:

```
$ dsh -a --show-machine-names -- uptime
example.com:  11:57:03 up 66 days, 20:51,  0 users,  load average: 0.29, 0.18, 0.07
joe@sample.com: 11:57  up 2 days, 15:21, 8 users, load averages: 0.52 0.31 0.26
```

You can create smaller or specialized collections of machines in individual files named $HOME/.dsh/group/*groupname*, where *groupname* is a meaningful name you assign. For example, if you create a file named *$HOME/.dsh/group/servers*, the command `dsh -g servers -- uptime` runs `uptime` on all machines listed in the *servers* file.

Feel free to mix and match `-m` with `-a` and `-g` to extend the global list or a group, respectively. Additionally, you can use `--file filename` to add all machines listed in *`filename`* to the list of hosts. By default, dsh runs commands in parallel. Instead, if you would like to run a command sequentially, specify `--wait-shell`.

While handy, dsh has one substantial detractor: It cannot copy files. If you want to deploy data, say, to more than one machine, you'll have to write a new script, adopt a distribution infrastructure (such as `rsync`), or consider a more robust tool (such as Parallel SSH (pssh)).

Just like ssh, only in parallel

Like dsh, pssh aims to streamline the administration of lots of machines. In addition to all the capabilities of dsh, pssh is able to copy files to and from a central server and kill processes across a bank of systems. The shell and its underlying library are written in Python, and pssh is easy to install assuming that your system already has the Python interpreter and core libraries (see Listing 3).

Listing 3. Installing pssh

```
$ # For systems with apt-get (apt-get installs Python if necessary)
$ sudo apt-get install pssh

$ # For all others, install Python and then continue
$ wget http://peak.telecommunity.com/dist/ez_setup.py
$ sudo python ez_setup.py
$ wget http://parallel-ssh.googlecode.com/files/pssh-2.1.1.tar.gz
$ tar xzvf pssh-2.1.1.tar.gz
$ cd pssh-2.1.1
$ sudo python setup.py install
```

The pssh package installs five utilities: `parallel-ssh`, `parallel-scp`, `parallel-slurp`, `parallel-nuke`, and `parallel-rsync`. Each utility operates on multiple hosts in parallel.

- `parallel-ssh` runs a command on multiple hosts in parallel.
- `parallel-scp`, as its name implies, copies files to multiple remote hosts in parallel.
- `parallel-rsync`, true to its moniker, efficiently copies files to multiple hosts in parallel via the `rsync` protocol.
- `parallel-slurp` copies files from multiple remote hosts to a central host in parallel.
- `parallel-nuke` kills processes on multiple remote hosts in parallel.

Unlike dsh, the hosts are always named via a *manifest*, a file where each line takes the form *`host[:port] [user]`*. Here's how to run `uptime` across a swath of hosts with `parallel-ssh`:

```
$ parallel-ssh -h servers.txt uptime
[1] 16:15:14 [SUCCESS] example.com 22
16:15  up 2 days, 19:39, 9 users, load averages: 0.09 0.10 0.12
[2] 16:15:28 [SUCCESS] sample.com 22
16:15:28 up 67 days,  1:09,  0 users,  load average: 0.09, 0.07, 0.01
```

The file servers.txt has two lines:

```
example.com
sample.com joe
```

By default, output from each instance of the command appears in stdout. The output is divided into sections, one section per host. However, you can name a directory to capture the stdout of each instance. For example, if you run the previous command and add `--outdir /tmp/uptime`, a transcript of the command from each host is captured in a separate file in /tmp/uptime, as Listing 4 shows.

Listing 4. Capturing output in a separate file

```
$ parallel-ssh -h servers.txt uptime
[1] 16:15:14 [SUCCESS] example.com 22
[2] 16:15:28 [SUCCESS] sample.com 22

$ ls -1 /tmp/uptime
example.com
sample.com

$ cat /tmp/uptime/*
16:22  up 2 days, 19:46, 9 users, load averages: 0.47 0.28 0.19
```

```
16:22:32 up 67 days,  1:17,  0 users,  load average: 0.06, 0.04, 0.00
```

The `parallel-ssh` utility can spawn a maximum of 32 processes to connect to various nodes in parallel. If a remote command does not complete after 60 seconds, the connection is terminated. If your command requires more processing time, use `-t` to set a longer expiration time. (`parallel-scp` and `parallel-rsync` do not have a default expiration, but you can specify one using `-t`.)

You can use `parallel-scp` to copy one or more files or directories to many machines in parallel. It should seem familiar if you've mastered the traditional `scp`.

```
$ parallel-scp -h servers.txt /etc/hosts /tmp/hosts
[1] 16:49:38 [SUCCESS] example.com 22
[2] 16:49:55 [SUCCESS] sample.com 22
```

The previous command copies the local file /etc/hosts to /tmp/hosts on each machine listed in servers.txt. `parallel-rsync` works similarly, running `rsync` in parallel to manage files between the local host and the remote hosts listed in the manifest. `parallel-slurp` works something like `parallel-scp` in reverse but with one twist: It collects the named file from each remote machine but does not overwrite the local version of the file. Instead, `parallel-slurp` creates a subdirectory for each remote machine and copies the named file to that location.

As a demonstration, imagine that you want to copy the /etc/hosts file from each remote machine to the local machine. To achieve the goal, you'd execute `parallel-slurp -h servers.txt /etc/hosts`, as shown in Listing 5.

Listing 5. Copying /etc/hosts from remote machines to the local machine

```
$ parallel-slurp -h servers.txt -L /tmp/hosts /etc/hosts hosts_file
1] 17:03:32 [SUCCESS] example.com 22
[2] 17:03:50 [SUCCESS] dcauto.gotdns.com 22

$ ls -R /tmp/hosts
/tmp/hosts/example.com:
hosts_file

/tmp/hosts/sample.com:
hosts_file
```

The `parallel-slurp` utility copies the named remote file to the local machine and stores each copy in a specific file in an individual subdirectory named after the remote host. Here, the remote file was /etc/hosts; each local copy is named *hosts_file*. The `-L` option specifies where to create the subdirectories. Here, the target was /tmp/hosts yielding subdirectories /tmp/hosts/example.com and /tmp/hosts/sample.com.

Finally, `parallel-nuke` is the equivalent of running `ssh host killall`. The argument to `parallel-nuke` is a pattern. Any process running on the remote machine whose name matches the pattern is killed. The command is handy for stopping the same daemon on a collection of servers.

To use the pssh tools, you must configure public key access to each remote server you want to administer. If a pssh utility yields `[FAILURE]`, verify your configuration by connecting with vanilla ssh. If you are prompted for a password, rectify the problem by installing your public key on the remote host and try again. (For instructions, see the `ssh` and `ssh-keygen` man pages.)

Machinery en masse

For 5, 10, or more machines, the tools described here likely suffice, especially for infrequent and ad hoc administration tasks. However, when the number of machines climbs higher or you repeat the same chores often, it may be prudent to consider other tools and subsystems designed to automate the maintenance of many machines. Reflexively, some software intended for large networks can be applied to a handful of machines, too. Finding the right tools and a balance between manual intervention and automation is a perennial challenge.

Here are some tools to consider:

- **`rsync`**. This is an excellent tool for distributing files from a central server and keeping distributed file systems in sync. A prior installment of Speaking UNIX covered `rsync` in detail.
- **Puppet**. Puppet is an increasingly popular subsystem for UNIX and Linux® that automates configuration maintenance. According to its web site, "[Puppet] provides a powerful framework to simplify the majority of technical tasks that [systems administrators] need to perform. [A chore] is written as code in Puppet’s custom language, which is shareable just like any other code." Puppet can describe dependencies between components, define the proper state of a file, query the state of the system, and more. If you ever perform a chore more than once, chances are it's best to capture it as a Puppet task.
- **Capistrano**. Capistrano is another popular tool for remote systems administration. Its home page describes the tool well: "Simply put, Capistrano is a tool for automating tasks on one or more remote servers. It executes commands in parallel on all targeted machines and provides a mechanism for rolling back changes across multiple machines. It is ideal for anyone doing any kind of system administration, either professionally or incidentally." Like Puppet, Capistrano is scripted. Scripts are based on the Ruby programming language and the Capistrano domain-specific extensions. Here's an example:

```
task :search_libs, :hosts => "www.capify.org" do
  run "ls -x1 /usr/lib | grep -i xml"
end
```

This task is named `search_libs`. It connects to www.capify.org and runs the command `ls -x1 /usr/lib | grep -i xml`. Capistrano supports groups of machines via *roles*, among hundreds of other features. Tasks are launched via the `cap` command, as in `cap search_libs`. Capistrano is used widely among Ruby and Rails developers to deploy code to servers, but it's an excellent choice for automating most distributed systems administration tasks. Tutorials explain how to mix Capistrano with the Java™ language, Perl, Python, and other programming languages and how to use Capistrano with application engines, such as Drupal and Expression Engine. Capistrano works best when paired with a source control system, but it's not required. You can distribute binaries with the `put` operation.

- **Nagios**. Maintenance is important, but so too is monitoring. Outages and errors can wreak havoc on a network, especially when many systems are running identical configurations. Nagios is an open source monitor that watches servers, services, resources, and more. It's easy to install and deploy and can be used via any web browser.

You may also want to look at compute cluster tools such as Oak Ridge National Laboratories' (ORNL) Cluster Command and Control (C3) and pdsh. C3 operates a massive compute cluster at ORNL and offers a number of command-line tools that increase systems manager productivity by reducing the time and effort required to operate and manage a cluster. The pdsh shell is similar in many ways to pssh but can also manage system images.

So many machines, so little time

Using tools such as dsh and pssh save time and reduce errors. You can run the same command on a large number of systems and see the combined results almost instantly. Manifests lump like machines together, too, reducing the risk of omissions. Puppet and Capistrano can capture oft-repeated tasks in scripts. If you manage more then a handful of machines, automation is key. See, even compute clouds can have a silver lining.

Resources

Learn

- Speaking UNIX: Check out other parts in this series.
- The rsync family (Federico Kereki, developerWorks, April 2009): Get to know `rsync` and a variety of utilities based on the `rsync` protocols.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- dsh: This shell can run a shell command on many machines in parallel.
- pssh. Parallel ssh runs commands, copies files, and manages processes on many machines in parallel.
- Puppet. Learn more about Puppet, one of the most popular distributed systems administration tools.
- Capistrano. Visit the Capistrano site to learn more about scripting administration with Ruby and the Capistrano domain-specific language.
- Nagios. This open source monitoring platform can be combined with automated maintenance tools to improve the overall stability of any size network.
- ORNL Cluster Command and Control. The C3 tools manage hundreds of compute nodes.
- pdsh: This compute cluster management tool can run a command on multiple machines in parallel.

Discuss

- developerWorks blogs: Check out our blogs and get involved in the developerWorks community.
- Follow developerWorks on Twitter.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX forums:
    - AIX Forum

- AIX Forum for developers
- Cluster Systems Management
- IBM Support Assistant Forum
- Performance Tools Forum
- Virtualization Forum
- More AIX and UNIX Forums

About the author

Martin Streicher

Martin Streicher is a freelance Ruby on Rails developer and the former Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has programmed UNIX-like systems since 1986. He collects art and toys. You can reach Martin at martin.streicher@gmail.com.