NAME

netrun − run a script over multiple hosts in parallel

SYNOPSIS

netrun [**−hqRS**] [**−c** *connect timeout*] [**−f** *max forks*] [**−i** *interpreter*] [**−s** *script file*] | [**−e** *script*] [**−d** *data file*] [**−l** *login name*] [**−L** *log dir*] [**−t** *timeout*] hosts ...

DESCRIPTION

Netrun provides a convenient and efficient way to run a single command or a script on a bunch of remote hosts. **Netrun** captures the output and error messages from the command or script for reporting and examination.

Netrun is powered by `ssh` and assumes that you have a setup like the following:

- You have created an SSH public/private key pair using `ssh−keygen`.
- You have copied your public key(s) to `$HOME/.ssh/authorized_keys` on all of the remote hosts on which you plan to run commands or scripts with **netrun**. If you plan to access remote hosts with different accounts, you'll need to make sure that your public key(s) have been added to each account's authorized_keys file. Be sure to check your local computer security policy and to get permission from the affected users before doing this!
- You have started an `ssh−agent` and loaded your private keys with `ssh−add`. Note that you can avoid having to run `ssh−agent` by creating private keys with no passphrase. Please do not do this! At many sites, this is grounds for disciplinary action (especially if the corresponding public keys are added to root's authorized_keys files).

For each hostname, IP address, or CIDR style subnet given on the command−line, **netrun** performs the following steps:

1. Connects to port 22 and captures the SSH version string. This is done to verify that the remote sshd is up and accessible. Hosts will be skipped if this step takes longer than 15 seconds (or the connection timeout specified by **−c**) to complete.
2. Attempts to establish an SSH connection and run an interpreter (the default is /bin/sh).
3. Feeds the script file (specified with **−s**) or command string (specified with **−e**) and an optional data file (specified with **−d**) to the standard input of the interpreter.
4. Captures stdout and stderr of the interpreter to log files. By default, these log files are stored in `./netrun.PID`, however an alternate log directory may be specified via **−L**. If the user running netrun does not have both write and search permissions to the log directory, **netrun** exits with an error message.
5. Displays a report that summarizes the status of the attempted actions. A copy of this summary is also saved in *logdir*/netrun.summary.

Netrun runs these steps on at most 25 hosts (by default) at a time. The level of parallelism can be adjusted using **−f**. The status report that is displayed once all of the parallel jobs have completed includes the following information:

Name/Address
: The hostname or IP address used to connect/login to the remote host

Exit
: The exit status of the ssh process (if an SSH connection was made). Usually, this is the exit status of the interpreter that was run on the remote system; however, the ssh program uses 255 to indicate that an error occurred while trying to establish the connection. A −1 exit status indicates that the remote script failed to complete before the timeout specified with **−t**.

Runtime
: The time in seconds required to ssh into the remote host and run the specified script or commands.

# Lines
: The number of lines of output produced by the remote script. This does not include lines written to stderr.

First Line of Output
: The first 30 or so bytes of the first line of output from the remote script.

If no script is specified (with **−s** or **−e**) on the command−line, **netrun** skips steps 2 − 4 above and just displays the status of the connection to the SSH port. Steps 2 − 4 are also skipped if **netrun** is unable to connect to the SSH port and retrieve the SSH version string from the remote host.

While **netrun** does not actually use the SSH version string, it retrieves it to verify that an SSH connection to the remote host is possible and is not being blocked by hosts.allow, etc. This prevents background processes from being tied up on ssh connections that can never succeed.

OPTIONS

−h Display a brief summary of the command-line options.

−q In Quick mode, **netrun** simply dumps script/command results from remote hosts to standard out. Each line is prefixed with the hostname or IP address of the host from which the output came. Hosts that fail to respond or to produce any output are excluded from the output. Also, log files are removed automatically.

This mode is handy for running a command or script on a bunch of hosts and grep'ing the output.

−R Don't randomize the hosts list. By default, all hosts and CIDR addresses are expanded and randomized. This evens out performance by creating a mix of slow and fast remote hosts. Specify **−R** if you need **netrun** to preserve the order of the hosts specified on the command−line.

−S Slow start. By default, **netrun** starts new processes as fast as it can, but when running large numbers of background processes (see **−f**), this can quickly overwhelm even a powerful workstation and may also trigger DoS settings in corporate firewalls. This option adds a 1/4 second sleep between each process start to pace things out a bit.

−c *connect timeout*
: In order avoid forking off lots of *ssh*(1) processes that may never come back, **netrun** tries to connect to port 22 on each host in the work list. If a host takes longer than a default of 15 seconds to respond, **netrun** skips it. This almost always works well, but if the SSH server is linked with the TCP Wrapper library and the nameservers are not responding, a timeout of at least two minutes will be necessary.

−f *max forks*
: Specify the maximum number of background processes. The default is 25.

−i *interpreter*
: Run the specified interpreter on the remote hosts. The default is `/bin/sh`. The interpreter is only run if a script file or command string is specified via **−s** or **−e** respectively.

−s *script file*
: Provide a local file containing a script to run on the remote hosts. This script is sent to the specified interpreter's standard input. The default interpreter is `/bin/sh`.

−e *script*
: Like **−s**, but the script source code is provided on the command−line, usually inside of single−quotes. This is handy for small one-off runs. If both **−e** and **−s** are specified, the command string from **−e** is sent first, followed by the contents of the file specified with **−s**.

−d *data file*
: Specify a local file to be appended to the end of the script. When the interpreter is `perl`, this is actually a convenient way to pass parameters or data to the remote script. Just make the first line of the data file contains only `__DATA__`, and have the script read from the `DATA` file handle. It's actually necessary to do things this way because the script source code is sent as standard input to the remote

interpreter; therefore there is no way to pass arguments on the command−line. If `−` is passed as the argument to `−d`, **netrun** reads the data from standard input instead of a file.

−l *login name*
Specifies the user to log in as on the remote hosts. By default, this is the same as the user running **netrun**.

−L *log dir*
Create log files in *log dir* instead of `./netrun.PID`. **Netrun** creates two log files for each remote host: *hostname*.err and *hostname*.out

The *hostname*.err file contains a few special fields written by **netrun** (used to create the status summary) as well as any messages sent to standard error by the script while it was running on the remote host.

The *hostname*.out file simply contains any text that the remote script wrote to standard output.

−t *timeout*
Kill the remote script after timeout seconds. By default, there is no time limit. The actual SSH process is sent a `SIGHUP`, which causes it to shutdown the remote shell and kill the interpreter. The exit status reported by **netrun** in the summary report will be `−1` in this case.

EXAMPLES

The first few examples illustrate how **netrun** works in terms of equivalent shell operations. Once you understand that, you'll know the limitations of **netrun** and how to best take advantage of its capabilities.

The following two commands do basically the same thing, run `uptime` on a remote host:

```
$ echo "uptime" | ssh fred.mydomain.com /bin/sh
$ netrun -qe uptime fred.mydomain.com
```

Netrun invokes an interpreter (default is `/bin/sh`) on each remote host and then sends commands, scripts, and/or data to the remote interpreter's standard input. It works this way because most of the time this eliminates the need to maintain a copy of your script on each remote host. For example:

```
$ cat my_script.pl | ssh fred.mydomain.com perl
$ netrun -qi perl -s my_script.pl fred.mydomain.com
```

The chief limitation of this approach is that you can't send command-line arguments to your script. In the case of **perl** scripts; however, you can send `DATA`. For example:

```
$ cat my_script.pl my_data.txt | ssh fred.mydomain.com perl
$ netrun -qi perl -d my_data.txt -s my_script.pl fred.mydomain.com
```

In this case, `my_data.txt` probably contains something like this:

```
__DATA__
@ARGV = split " ", "-a -f /etc/init.d -v";
```

In `my_script.pl`, there might be some (slightly dangerous) code like this:

```
while ( <DATA> ) { eval $_ }
```

Here is another somewhat silly example which shows that **netrun** runs the interpreter specified by **−i** on each remote host, sending to that interpreter's standard input the value of **−e** followed by the contents of the local file specified by **−s** followed by the contents of the local file specified by **−d**:

```
$ netrun -i perl -e 'print "Hello, ''uname -n''";' \
                 -s my_script.pl \
                 -d my_data.txt  fred.mydomain.com
```

The above example prepends a **perl** print statement to the `my_script.pl` script and tacks the contents of my_data.txt to the end of it. It sends the resulting concatenation to the standard input of the **perl** process on each remote host, where hopefully it will do something useful.

With that introduction, here are some hopefully more useful examples:

Tell `inetd` to re-read its configuration file on a bunch of Solaris hosts:

```
$ test -n "$SSHS_AGENT_PID" && kill -0 $SSHS_AGENT_PID \
  || eval ‘ssh-agent‘
$ ssh-add -l | grep ’no identities’ && ssh-add
$ netrun -l root -e ’pkill -1 inetd’ ‘cat hosts.lst‘
$ ssh-add -d
```

Create a list of all systems on the local LAN that are running Oracle:

```
$ netrun -e ’ps -ef’ -L Oracle 192.168.1.0/24
$ grep -li oracle Oracle/*.out | sed ’s/\.out$//’
$ rm -rf Oracle
```

Same thing but shorter:

```
$ netrun -q -e ’ps -ef’ 192.168.1.0/24 | grep -i oracle | cut -d: -f1
```

Clone the local `/etc/sudoers` file out to a bunch of hosts (assumes that you can sudo run `tee` and `cksum` and that you have the NOPASSWD flag set):

```
$ sudo cat /etc/sudoers | netrun -q -d - -i ’
     sudo tee /etc/sudoers > /dev/null
     sudo cksum /etc/sudoers’ \
  ‘cat hosts.lst‘
```

SEE ALSO

ssh, sh, perl, grep, wfrun

AUTHOR

David C. Snyder <David.Snyder@turner.com>