Last update; 3/8/2018

A brief tutorial on the various ways to do networking things from Bash

Topics

- Redirection
- Bash scripts related to networking

3.6 Redirections

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

Each redirection that may be preceded by a file descriptor number may instead be preceded by a word of the form {*varname*}. In this case, for each redirection operator except >&- and <&-, the shell will allocate a file descriptor greater than 10 and assign it to {*varname*}. If >&- or <&- is preceded by {*varname*}, the value of *varname* defines the file descriptor to close.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is '<', the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is '>', the redirection refers to the standard output (file descriptor 1).

The word following the redirection operator in the following descriptions, unless otherwise noted, is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, filename expansion, and word splitting. If it expands to more than one word, Bash reports an error.

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file *dirlist*, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file *dirlist*, because the standard error was made a copy of the standard output before the standard output was redirected to *dirlist*.

Bash handles several filenames specially when they are used in redirections, as described in the following table. If the operating system on which Bash is running provides these special files, bash will use them; otherwise it will emulate them internally with the behavior described below.

`/dev/fd/fd`
: If *fd* is a valid integer, file descriptor *fd* is duplicated.

`/dev/stdin`
: File descriptor 0 is duplicated.

`/dev/stdout`
: File descriptor 1 is duplicated.

`/dev/stderr`
: File descriptor 2 is duplicated.

`/dev/tcp/host/port`
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open the corresponding TCP socket.

`/dev/udp/host/port`
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open the corresponding UDP socket.

A failure to open or create a file causes the redirection to fail.

Redirections using file descriptors greater than 9 should be used with care, as they may conflict with file descriptors the shell uses internally.

Last update; 3/8/2018

cat < /dev/tcp/"time.nist.gov/13"

58186 18-03-09 01:20:54 53 0 0 332.9 UTC(NIST) *

```
exec 3<>/dev/tcp/www.google.com/80
echo -e "GET / HTTP/1.1\r\nhost: http://www.google.com\r\nConnection:
close\r\n\r\n" >&3
cat <&3
```

Bash has a built in command that looks like:

Cat > /dev/tcp/ip/port and it sends the steam out.

Very similar to wc.

If you saw yesterday's **Tech Tip** and were looking for more on using TCP/IP with bash's built-in `/dev/tcp` *device file* then read on. Here, we'll both read from, and write to a socket.

Before I go any further, let me state that this is based on something I discovered **here** on **Dave Smith's Blog**. All I've done here is added a few improvements based on the comments to the original post. I've also added a bit of additional explanation.

The following script fetches the front page from Google:

```
exec 3<>/dev/tcp/www.google.com/80
echo -e "GET / HTTP/1.1\r\nhost: http://www.google.com\r\nConnection:
close\r\n\r\n" >&3
cat <&3
```

Pretty simple, just 3 lines. The first line may be a bit confusing if you haven't seen this type of thing before. This line causes file descriptor 3 to be opened for reading and writing on the specified TCP/IP socket. This is a special form of the `exec`statement. From the bash man page:

exec [-cl] [-a name] [command [arguments]]

... If command is not specified, any redirections take effect in the current shell, and the return status is 0.

So using `exec` without a command is a way to open files in the current shell.

After the socket is open we send our HTTP request out the socket with the `echo ...>&3` command. The request consists of:

```
GET / HTTP/1.1
host: http://www.google.com
Connection: close
```

Each line is followed by a carriage-return and newline, and all the headers are followed by a blank line to signal the end of the request (this is all standard HTTP stuff).

Next we read the response out of the socket using `cat <&3`, which reads the response and prints it out. The response being the main HTML page from Google:

```
$ bash tcp.sh
HTTP/1.1 200 OK
Date: Wed, 30 Sep 2009 17:28:36 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=...
Set-Cookie: NID=27=...
Server: gws
X-XSS-Protection: 0
Transfer-Encoding: chunked
Connection: close

fef
<!doctype html><html><head><meta ...
```

And that's it, with just a few more lines of code you could have your own bash based browser... well maybe not.

```
printf "GET / HTTP/1.1\n\n" > /dev/tcp/74.125.225.19/80
echo -e "GET / HTTP/1.1\n\n" | nc 74.125.225.19 80
```

Example to get web content

```
telnet                                  ipecho.net                                  80 Enter
GET                                    /plain                                HTTP/1.1 Enter
HOST:                                                                    ipecho.net Enter
BROWSER:                                                                     web-kit Enter
Enter
```

```
curl ipecho.net/plain
```

For this, STUN was invented. As a client you can send a request to a publicly available STUN server and have it give back the IP address it sees. Sort of the low level whatismyip.com as it uses no HTTP and no smartly crafted DNS servers but the blazingly fast STUN protocol.

Using stunclient

If you have stunclient installed (apt-get install stuntman-client on debian/ubuntu) you can simply do:

$stunclient stun.services.mozilla.com

Binding test: success

Local address: A.B.C.D:42541

Mapped address: W.X.Y.Z:42541

where A.B.C.D is the IP address of your machine on the local net and W.X.Y.Z is the IP address servers like websites see from the outside (and the one you are looking for). Using sed you can reduce the output above to only an IP address:

stunclient stun.services.mozilla.com |

sed -n -e "s/^Mapped address: \(.*\):.*$/\1/p"

However, your question was how to find it using the command line, which might exclude using a STUN client. So I wonder...

Using bash

A STUN request can be handcrafted, sent to an external STUN server using netcat and be post-processed using dd, hexdump and sed like so:

$echo -en '\x00\x01\x00\x08\xc0\x0c\xee\x42\x7c\x20\x25\xa3\x3f\x0f\xa1\x7f\xfd\x7f\x00\x00\x00\x03\x00\x04\x00\x00\x00\x00' |

```
nc -u -w 2 stun.services.mozilla.com 3478 |
dd bs=1 count=4 skip=28 2>/dev/null |
hexdump -e '1/1 "%u."' |
sed 's/\.$/\n/'
```

The echo defines a binary STUN request (0x0001 indicates Binding Request) having length 8 (0x0008) with cookie 0xc00cee and some pasted stuff from wireshark. Only the four bytes representing the external IP are taken from the answer, cleaned and printed.

Working, but not recommended for production use :-)

P.S. Many STUN servers are available as it is a core technology for SIP and WebRTC. Using one from Mozilla should be safe privacy-wise but you could also use another: STUN server list

To use only Bash:

```
$ exec 3<> /dev/tcp/icanhazip.com/80 && # open connection
  echo 'GET /' >&3 &&                   # send http 0.9 request
  read -u 3 && echo $REPLY &&           # read response
  exec 3>&-                             # close fd
```

Grep for an ip address:

```
grep -Eo '\<[[:digit:]]{1,3}(\.[[:digit:]]{1,3}){3}\>'
```

```
echo $(ip route get 8.8.8.8 | awk '{print $NF; exit}')
sudo traceroute -I google.com | awk -F '[ ]' '{ if ( $2 ="2" ) { print $5 } }'
```

shows interfaces and IPV4

```
ifconfig | sed -nre '/^[^ ]+/{N;s/^([^ ]+).*addr: *([^ ]+).*/\1,\2/p}'
ifconfig | sed -nre '/^[^ ]+/{N;N;s/^([^ ]+).*addr: *([^ ]+).*addr: *([^ ]+).*/\1,\2,\3/p}'
```