8
The Old-Boy Network

In this chapter, we will cover the following recipes:

- Setting up the network
- Let us ping!
- Tracing IP routes
- Listing all available machines on a network
- Running commands on a remote host with SSH
- Running graphical commands on a remote machine
- Transferring files through the network
- Connecting to a wireless network
- Password-less auto-login with SSH
- Port forwarding using SSH
- Mounting a remote drive at a local mount point
- Network traffic and port analysis
- Measuring network bandwidth
- Creating arbitrary sockets
- Building a bridge
- Sharing an Internet connection
- Basic firewall using `iptables`
- Creating a Virtual Private Network

Introduction

Networking is the act of connecting computers to allow them to exchange information. The most widely used networking stack is TCP/IP, where each node is assigned a unique IP address for identification. If you are already familiar with networking, you can skip this introduction.

TCP/IP networks work by passing data packets from node to node. Each data packet contains the IP address of its destination and the port number of the application that can process this data.

When a node receives a packet, it checks to see if it is this packet's destination. If so, the node checks the port number and invokes the appropriate application to process the data. If this node is not the destination, it evaluates what it knows about the network and passes the packet to a node that is closer to the final destination.

Shell scripts can be used to configure the nodes in a network, test the availability of machines, automate execution of commands at remote hosts, and more. This chapter provides recipes that introduce tools and commands related to networking, and shows how to use them effectively.

Setting up the network

Before digging through recipes based on networking, it is essential to have a basic understanding of setting up a network, terminologies, and commands for assigning IP address, adding routes, and so on. This recipe provides an overview of commands used in GNU/Linux networks.

Getting ready

A network interface physically connects a machine to a network, either with a wire or a Wi-Fi link. Linux denotes network interfaces using names such as `eth0`, `eth1`, or `enp0s25` (referring to Ethernet interfaces). Other interfaces, namely `usb0`, `wlan0`, and `tun0`, are available for USB network interfaces, wireless LAN, and tunnels, respectively.

In this recipe, we will use these commands: `ifconfig`, `route`, `nslookup`, and `host`.

The `ifconfig` command is used to configure and display details about network interfaces, subnet mask, and so on. It should be available at `/sbin/ifconfig`.

How to do it...

1. List the current network interface configuration:

```
$ ifconfig
lo        Link encap:Local Loopback
inet addr:127.0.0.1  Mask:255.0.0.0
inet6addr: ::1/128 Scope:Host
   UP LOOPBACK RUNNING  MTU:16436  Metric:1
   RX packets:6078 errors:0 dropped:0 overruns:0 frame:0
   TX packets:6078 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
   RX bytes:634520 (634.5 KB)  TX bytes:634520 (634.5 KB)
wlan0     Link encap:EthernetHWaddr 00:1c:bf:87:25:d2
inet addr:192.168.0.82  Bcast:192.168.3.255  Mask:255.255.252.0
inet6addr: fe80::21c:bfff:fe87:25d2/64 Scope:Link
   UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
   RX packets:420917 errors:0 dropped:0 overruns:0 frame:0
   TX packets:86820 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
   RX bytes:98027420 (98.0 MB)  TX bytes:22602672 (22.6 MB)
```

   The leftmost column in the ifconfig output lists the names of network interfaces, and the right-hand columns show the details related to the corresponding network interface.

2. To set the IP address for a network interface, use the following command:

```
# ifconfig wlan0 192.168.0.80
```

   You will need to run the preceding command as root

   `192.168.0.80` is defined as the address for the wireless device, wlan0

   To set the subnet mask along with the IP address, use the following command:

```
# ifconfig wlan0 192.168.0.80  netmask 255.255.252.0
```

3. Many networks use **Dynamic Host Configuration Protocol** (**DHCP**) to assign IP addresses automatically when a computer connects to the network. The `dhclient` command assigns the IP address when your machine is connected to a network that assigns IP addresses automatically. If addresses are assigned via DHCP, use `dhclient` instead of manually choosing an address that might conflict with another machine on the network. Many Linux distributions invoke `dhclient` automatically when they sense a network cable connection

```
# dhclient eth0
```

There's more...

The `ifconfig` command can be combined with other shell tools to produce specific reports.

Printing the list of network interfaces

This one-line command sequence displays network interfaces available on a system:

```
$ ifconfig | cut -c-10 | tr -d ' ' | tr -s 'n'
lo
wlan0
```

The first ten characters of each line in `ifconfig` output is reserved for writing names of network interfaces. Hence, we use `cut` to extract the first ten characters of each line. `tr -d ' '` deletes every space character in each line. Now, the `n` newline character is squeezed using `tr -s 'n'` to produce a list of interface names.

Displaying IP addresses

The `ifconfig` command displays details of every active network interface available on the system. However, we can restrict it to a specific interface using the following command:

```
$ ifconfig iface_name
```

Consider this example:

```
$ ifconfig wlan0
wlan0     Link encap:EthernetHWaddr 00:1c:bf:87:25:d2
inet addr:192.168.0.82 Bcast:192.168.3.255 Mask:255.255.252.0
inet6 addr: fe80::3a2c:4aff:6e6e:17a9/64 Scope:Link
UP BROADCAST RUNNINT MULTICAST  MTU:1500 Metric:1
RX Packets...
```

To control a device, we need the IP address, broadcast address, hardware address, and subnet mask:

- `HWaddr 00:1c:bf:87:25:d2`: This is the hardware address (MAC address)
- `inet addr:192.168.0.82`: This is the IP address
- `Bcast:192.168.3.255`: This is the broadcast address
- `Mask:255.255.252.0`: This is the subnet mask

To extract the IP address from the `ifconfig` output, use this command:

```
$ ifconfig wlan0 | egrep -o "inetaddr:[^ ]*" | grep -o "[0-9.]*"
192.168.0.82
```

The `egrep -o "inetaddr:[^ ]*"` command returns `inet addr:192.168.0.82`. The pattern starts with `inetaddr:` and ends with any non-space character sequence (specified by `[^ ]*`). The next command, `grep -o "[0-9.]*"` reduces its input to only numbers and periods, and prints out an IP4 address.

Spoofing the hardware address (MAC address)

When authentication or filtering is based on the hardware address, we can use hardware address spoofing. The hardware address appears in the `ifconfig` output as `HWaddr 00:1c:bf:87:25:d2`.

The `hw` subcommand of `ifconfig` will define a devices class and the MAC address:

```
# ifconfig eth0 hw ether 00:1c:bf:87:25:d5
```

In the preceding command, `00:1c:bf:87:25:d5` is the new MAC address to be assigned. This is useful when we need to access the Internet through MAC-authenticated service providers that provide access to the Internet for a single machine.

Note: this definition only lasts until a machine restarts.

Name server and DNS (Domain Name Service)

The underlying addressing scheme for the Internet is the dotted decimal form (like `83.166.169.231`). Humans prefer to use words instead of numbers, so resources on the Internet are identified with strings of words called **URLs** or **domain names**. For example, `www.packtpub.com` is a domain name and it corresponds to an IP address. The site can be identified by the numeric or the string name.

This technique of mapping IP addresses to symbolic names is called **Domain Name Service** (**DNS**). When we enter `www.google.com`, our computer uses the DNS servers to resolve the domain name into the corresponding IP address. While on a local network, we set up the local DNS to name local machines with symbolic names.

Name servers are defined in `/etc/resolv.conf`:

```
$ cat /etc/resolv.conf
# Local nameserver
nameserver 192.168.1.1
# External nameserver
nameserver 8.8.8.8
```

We can add name servers manually by editing that file or with a one-liner:

```
# sudo echo nameserver IP_ADDRESS >> /etc/resolv.conf
```

The easiest method to obtain an IP address is to use the `ping` command to access the domain name. The reply includes the IP address:

```
$ ping google.com
PING google.com (64.233.181.106) 56(84) bytes of data.
```

The number `64.233.181.106` is the IP address of a google.com server.

A domain name may map to multiple IP addresses. In that case, `ping` shows one address from the list of IP addresses. To obtain all the addresses assigned to the domain name, we should use a DNS lookup utility.

DNS lookup

Several DNS lookup utilities provide name and IP address resolution from the command line. The `host` and `nslookup` commands are two commonly installed utilities.

The `host` command lists all of the IP addresses attached to a domain name:

```
$ host google.com
google.com has address 64.233.181.105
google.com has address 64.233.181.99
google.com has address 64.233.181.147
google.com has address 64.233.181.106
google.com has address 64.233.181.103
google.com has address 64.233.181.104
```

The `nslookup` command maps names to IP addresses and will also map IP addresses to names:

```
$ nslookup google.com
Server:    8.8.8.8
Address:  8.8.8.8#53

Non-authoritative answer:
Name:  google.com
Address: 64.233.181.105
Name:  google.com
Address: 64.233.181.99
Name:  google.com
Address: 64.233.181.147
Name:  google.com
Address: 64.233.181.106
Name:  google.com
Address: 64.233.181.103
Name:  google.com
Address: 64.233.181.104

Server:    8.8.8.8
```

The last line in the preceding command-line snippet corresponds to the default name server used for resolution.

It is possible to add a symbolic name to the IP address resolution by adding entries into the `/etc/hosts` file.

Entries in `/etc/hosts follow this format`:

```
IP_ADDRESS name1 name2 ...
```

You can update `/etc/hosts` like this:

```
# echo IP_ADDRESS symbolic_name>> /etc/hosts
```

Consider this example:

```
# echo 192.168.0.9 backupserver>> /etc/hosts
```

After adding this entry, whenever resolution to `backupserver` occurs, it will resolve to `192.168.0.9`.

If `backupserver` has multiple names, you can include them on the same line:

```
# echo 192.168.0.9 backupserver backupserver.example.com >> /etc/hosts
```

Showing routing table information

It is common to have interconnected networks. For example, different departments at work or school may be on separate networks. When a device on one network wants to communicate with a device on the other network, it needs to send packets through a device which is common to both networks. This device is called a `gateway` and its function is to route packets to and from different networks.

The operating system maintains a table called the `routing table`, which contains the information on how packets are to be forwarded through machines on the network. The `route` command displays the routing table:

```
$ route
Kernel IP routing table
Destination      Gateway   GenmaskFlags  Metric  Ref  UseIface
192.168.0.0         *      255.255.252.0  U     2      0     0wlan0
link-local          *      255.255.0.0    U     1000   0     0wlan0
default          p4.local  0.0.0.0        UG    0      0     0wlan0
```

Alternatively, you can also use this:

```
$ route -n
Kernel IP routing table
Destination   Gateway      Genmask       Flags Metric Ref  UseIface
192.168.0.0   0.0.0.0      255.255.252.0   U     2      0     0   wlan0
169.254.0.0   0.0.0.0      255.255.0.0     U     1000   0     0   wlan0
0.0.0.0       192.168.0.4  0.0.0.0         UG    0      0     0   wlan0
```

Using `-n` specifies to display the numeric addresses. By default, route will map the numeric address to a name.

When your system does not know the route to a destination, it sends the packet to a default gateway. The default gateway may be the link to the Internet or an inter-departmental router.

The `route add` command can add a default gateway:

```
# route add default gw IP_ADDRESS INTERFACE_NAME
```

Consider this example:

```
# route add default gw 192.168.0.1 wlan0
```

See also

- The *Using variables and environment variables* recipe of `Chapter 1`, *Shell Something Out*, explains the `PATH` variable
- The *Searching and mining text inside a file with grep* recipe of `Chapter 4`, *Texting and Driving*, explains the `grep` command

Let us ping!

The `ping` command is a basic network command, supported on all major operating systems. Ping is used to verify connectivity between hosts on a network and identify accessible machines.

How to do it...

The ping command uses **Internet Control Message Protocol** (**ICMP**) packets to check the connectivity of two hosts on a network. When these echo packets are sent to a target, the target responds with a reply if the connection is complete. A ping request can fail if there is no route to the target or if there is no known route from the target back to the requester.

Pinging an address will check whether a host is reachable:

```
$ ping ADDRESS
```

The `ADDRESS` can be a hostname, domain name, or an IP address itself.

By default, `ping` will continuously send packets and the reply information is printed on the terminal. Stop the pinging process by pressing *Ctrl* + *C*.

Consider the following example:

- When a host is reachable, the output will be similar to the following:

```
$ ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=1.44 ms
^C
--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.440/1.440/1.440/0.000 ms

$ ping google.com
PING google.com (209.85.153.104) 56(84) bytes of data.
64 bytes from bom01s01-in-f104.1e100.net (209.85.153.104):
icmp_seq=1 ttl=53 time=123 ms
^C
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 123.388/123.388/123.388/0.000 ms
```

- When a host is unreachable, the output will resemble this:

```
$ ping 192.168.0.99
PING 192.168.0.99 (192.168.0.99) 56(84) bytes of data.
From 192.168.0.82 icmp_seq=1 Destination Host Unreachable
From 192.168.0.82 icmp_seq=2 Destination Host Unreachable
```

If the target is not reachable, the ping returns with the `Destination Host Unreachable` error message.

Network administrators generally configure devices such as routers not to respond to `ping`. This is done to lower security risks, as `ping` can be used by attackers (using brute-force) to find out IP addresses of machines.

There's more...

In addition to checking the connectivity between two points in a network, the `ping` command returns other information. The round trip time and lost packet reports can be used to determine whether a network is working properly.

Round Trip Time

The `ping` command displays **Round Trip Time** (**RTT**) for each packet sent and returned. RTT is reported in milliseconds. On an internal network, a RTT of under 1ms is common. When pinging a site on the Internet, RTT are commonly 10-400 ms, and may exceed 1000 ms:

```
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 118.012/206.630/347.186/77.713 ms
```

Here, the minimum RTT is `118.012 ms`, the average RTT is `206.630` ms, and the maximum RTT is `347.186ms`. The `mdev` (`77.713ms`) parameter in the ping output stands for mean deviation.

Sequence number

Each packet that ping sends is assigned a number, sequentially from 1 until ping stops. If a network is near saturation, packets may be returned out of order because of collisions and retries, or may be completely dropped:

```
$> ping example.com
64 bytes from example.com (1.2.3.4): icmp_seq=1 ttl=37 time=127.2 ms
64 bytes from example.com (1.2.3.4): icmp_seq=3 ttl=37 time=150.2 ms
64 bytes from example.com (1.2.3.4): icmp_seq=2 ttl=30 time=1500.3 ms
```

In this example, the second packet was dropped and then retried after a timeout, causing it to be returned out of order and with a longer Round Trip Time.

Time to live

Each ping packet has a predefined number of hops it can take before it is dropped. Each router decrements that value by one. This value shows how many routers are between your system and the site you are pinging. The initial **Time To Live** (**TTL**) value can vary depending on your platform or ping revision. You can determine the initial value by pinging the loopback connection:

```
$> ping 127.0.0.1
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.049 ms
$> ping www.google.com
64 bytes from 173.194.68.99: icmp_seq=1 ttl=45 time=49.4 ms
```

In this example, we ping the loopback address to determine what the TTL is with no hops (in this case, 64). Then we ping a remote site and subtract that TTL value from our No-Hop value to determine how many hops are between the two sites. In this case, 64-45 is 19 hops.

The TTL value is usually constant between two sites, but can change when conditions require alternative paths.

Limiting the number of packets to be sent

The `ping` command sends echo packets and waits for the reply of echo indefinitely until it is stopped by pressing *Ctrl* + *C*. The `-c` flag will limit the count of echo packets to be sent:

```
-c COUNT
```

Consider this example:

```
$ ping 192.168.0.1 -c 2
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=4.02 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=1.03 ms

--- 192.168.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 1.039/2.533/4.028/1.495 ms
```

In the previous example, the `ping` command sends two echo packets and stops. This is useful when we need to ping multiple machines from a list of IP addresses through a script and check their statuses.

Return status of the ping command

The `ping` command returns the exit status `0` when it succeeds and returns non-zero when it fails. `Successful` means the destination host is reachable, whereas `Failure` is when the destination host is unreachable.

The return status can be obtained as follows:

```
$ ping domain -c2
if [ $? -eq0 ];
then
  echo Successful ;
else
  echo Failure
fi
```

Tracing IP routes

When an application requests a service through the Internet, the server may be at a distant location and connected via many of gateways or routers. The `traceroute` command displays the address of all intermediate gateways a packet visits before reaching its destination. `traceroute` information helps us to understand how many hops each packet takes to reach a destination. The number of intermediate gateways represents the effective distance between two nodes in a network, which may not be related to the physical distance. Travel time increases with each hop. It takes time for a router to receive, decipher, and transmit a packet.

How to do it...

The format for the `traceroute` command is as follows:

```
traceroute destinationIP
```

`destinationIP` may be numeric or a string:

```
$ traceroute google.com
traceroute to google.com (74.125.77.104), 30 hops max, 60 byte packets
1  gw-c6509.lxb.as5577.net (195.26.4.1)  0.313 ms  0.371 ms  0.457 ms
2  40g.lxb-fra.as5577.net (83.243.12.2)  4.684 ms  4.754 ms  4.823 ms
3  de-cix10.net.google.com (80.81.192.108)  5.312 ms  5.348 ms  5.327 ms
4  209.85.255.170 (209.85.255.170)  5.816 ms  5.791 ms 209.85.255.172
(209.85.255.172)  5.678 ms
5  209.85.250.140 (209.85.250.140)  10.126 ms  9.867 ms  10.754 ms
6  64.233.175.246 (64.233.175.246)  12.940 ms 72.14.233.114 (72.14.233.114)
13.736 ms  13.803 ms
7  72.14.239.199 (72.14.239.199)  14.618 ms 209.85.255.166 (209.85.255.166)
12.755 ms 209.85.255.143 (209.85.255.143)  13.803 ms
8  209.85.255.98 (209.85.255.98)  22.625 ms 209.85.255.110 (209.85.255.110)
14.122 ms
*
9  ew-in-f104.1e100.net (74.125.77.104)  13.061 ms  13.256 ms  13.484 ms
```

Modern Linux distributions also ship with an `mtr` command, which is similar to traceroute but shows real-time data that keeps refreshing. It is useful for checking your network carrier quality.

Listing all available machines on a network

When we monitor a large network, we need to check the availability of all machines. A machine may not be available for two reasons: it is not powered on, or because of a problem in the network. We can write a shell script to determine and report which machines are available on the network.

Getting ready

In this recipe, we demonstrate two methods. The first method uses ping and the second method uses `fping`. The `fping` command is easier for scripts and has more features than the ping command. It may not be part of your Linux distribution, but can be installed with your package manager.

How to do it...

The next example script will find the visible machines on the network using the ping command:

```
#!/bin/bash
#Filename: ping.sh
# Change base address 192.168.0 according to your network.

for ip in 192.168.0.{1..255} ;
do
  ping $ip -c 2 &> /dev/null ;

  if [ $? -eq 0 ];
  then
    echo $ip is alive
  fi
done
```

The output resembles this:

```
$ ./ping.sh
192.168.0.1 is alive
192.168.0.90 is alive
```

How it works...

This script uses the `ping` command to find out the available machines on the network. It uses a `for` loop to iterate through a list of IP addresses generated by the expression `192.168.0.{1..255}`. The `{start..end}` notation generates values between start and end. In this case, it creates IP addresses from `192.168.0.1` to `192.168.0.255`.

`ping $ip -c 2 &> /dev/null` runs a `ping` command to the corresponding IP address. The `-c` option causes ping to send only two packets. The `&> /dev/null` redirects both `stderr` and `stdout to /dev/null`, so nothing is printed on the terminal. The script uses `$?` to evaluate the exit status. If it is successful, the exit status is `0`, and the IP address which replied to our ping is printed.

In this script, a separate `ping` command is executed for each address, one after the other. This causes the script to run slowly when an IP address does not reply, since each ping must wait to time out before the next ping begins.

There's more...

The next recipes show enhancements to the ping script and how to use `fping`.

Parallel pings

The previous script tests each address sequentially. The delay for each test is accumulated and becomes large. Running the ping commands in parallel will make this faster. Enclosing the body of the loop in `{}&` will make the `ping` commands run in parallel. `( )` encloses a block of commands to run as a subshell, and `&` sends it to the background:

```
#!/bin/bash
#Filename: fast_ping.sh
# Change base address 192.168.0 according to your network.

for ip in 192.168.0.{1..255} ;
do
   (
```

```
        ping $ip -c2 &> /dev/null ;

        if [ $? -eq0 ];
        then
         echo $ip is alive
        fi
    )&
  done
wait
```

In the `for` loop, we execute many background processes and come out of the loop, terminating the script. The `wait` command prevents the script from terminating until all its child processes have exited.

The output will be in the order that pings reply. This will not be the numeric order in which they were sent if some machines or network segments are slower than others.

Using fping

The second method uses a different command called `fping`. The `fping` command generates ICMP messages to multiple IP addresses and then waits to see which reply. It runs much faster than the first script.

The options available with `fping` include the following:

- The `-a` option with `fping` specifies to display the IP addresses for available machines
- The `-u` option with `fping` specifies to display unreachable machines
- The `-g` option specifies generating a range of IP addresses from the slash-subnet mask notation specified as IP/mask or start and end IP addresses:

  ```
  $ fping -a 192.160.1/24 -g
  ```

  Alternatively, this can be used:

  ```
  $ fping -a 192.160.1 192.168.0.255 -g
  ```

- `2>/dev/null` is used to dump error messages printed due to an unreachable host to a null device

It is also possible to manually specify a list of IP addresses as command-line arguments or as a list through `stdin`. Consider the following example:

```
$ fping –a 192.168.0.1 192.168.0.5 192.168.0.6
# Passes IP address as arguments
$ fping –a <ip.list
# Passes a list of IP addresses from a file
```

See also

- The *Playing with file descriptors and redirection* recipe in `Chapter 1`, *Shell Something Out*, explains the data redirection
- The *Comparisons and tests* recipe in `Chapter 1`, *Shell Something Out*, explains numeric comparisons

Running commands on a remote host with SSH

SSH stands for **Secure Shell**. It connects two computers across an encrypted tunnel. SSH gives you access to a shell on a remote computer where you can interactively run a single command and receive the results or start an interactive session.

Getting ready

SSH doesn't come preinstalled with all GNU/Linux distributions. You may have to install the `openssh-server` and `openssh-client` packages using a package manager. By default, SSH runs on port number `22`.

How to do it...

1. To connect to a remote host with the SSH server running, use the following command:

```
$ ssh username@remote_host
```

The options in this command are as follows:

- `username` is the user that exists at the remote host
- `remote_host` can be the domain name or IP address

Consider this example:

```
$ ssh mec@192.168.0.1
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be
established.
RSA key fingerprint is
2b:b4:90:79:49:0a:f1:b3:8a:db:9f:73:2d:75:d6:f9.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of
known hosts.
Password:

Last login: Fri Sep  3 05:15:21 2010 from 192.168.0.82
mec@proxy-1:~$
```

SSH will ask for a password, and upon successful authentication it will connect to the login shell on the remote machine.

SSH performs a fingerprint verification to make sure we are actually connecting to the remote computer we want. This is to avoid what is called a **man-in-the-middle attack**, where an attacker tries to impersonate another computer. SSH will, by default, store the fingerprint the first time we connect to a server and verify that it does not change for future connections.

By default, the SSH server runs at port `22`. However, certain servers run SSH service at different ports. In that case, use `-p port_num` with the `ssh` command to specify the port.

2. Connect to an SSH server running at port `422`:

```
$ ssh user@locahost -p 422
```

When using `ssh` in shell scripts, we do not want an interactive shell, we simply want to execute commands on the remote system and process the command's output.

Issuing a password every time is not practical for an automated script, so password-less login using SSH keys should be configured. The *Password-less auto-login with SSH* recipe in this chapter explains the SSH commands to set this up.

3. To run a command at the remote host and display its output on the local shell, use the following syntax:

```
$ sshuser@host 'COMMANDS'
```

Consider this example:

```
$ ssh mec@192.168.0.1 'whoami'
mec
```

You can submit multiple commands by separating the commands with a semicolon:

```
$ ssh user@host "command1 ; command2 ; command3"
```

Consider the following example:

```
$ ssh mec@192.168.0.1  "echo user: $(whoami);echo OS: $(uname)"
Password:
user: mec
OS: Linux
```

In this example, the commands executed at the remote host are as follows:

```
echo user: $(whoami);
echo OS: $(uname)
```

We can pass a more complex subshell in the command sequence using the `( )` subshell operator.

3. The next example is an SSH-based shell script to collect the uptime of a list of remote hosts. Uptime is the length of time since the last power-on. It's returned by the `uptime` command.

It is assumed that all systems in `IP_LIST` have a common user `test`.

```
#!/bin/bash
#Filename: uptime.sh
#Description: Uptime monitor

IP_LIST="192.168.0.1 192.168.0.5 192.168.0.9"
```

```
USER="test"

for IP in $IP_LIST;
do
utime=$(ssh ${USER}@${IP} uptime  |awk '{ print $3 }' )
  echo $IP uptime:  $utime
done
```

Expected output:

```
$ ./uptime.sh
192.168.0.1 uptime: 1:50,
192.168.0.5 uptime: 2:15,
192.168.0.9 uptime: 10:15,
```

There's more...

The `ssh` command can be executed with several additional options.

SSH with compression

The SSH protocol supports compressing the data transfer. This feature comes in handy when bandwidth is an issue. Use the `-C` option with the `ssh` command to enable compression:

```
$ ssh -C user@hostname COMMANDS
```

Redirecting data into stdin of remote host shell commands

SSH allows you to use output from a task on your local system as input on the remote system:

```
$ echo 'text' | ssh user@remote_host 'echo'
text
```

Alternatively, this can be used:

```
# Redirect data from file as:
$ ssh user@remote_host 'echo'  < file
```

`echo` on the remote host prints the data received through `stdin`, which in turn is passed to `stdin` from localhost.

This facility can be used to transfer tar archives from a local host to the remote host. This is described in detail in `Chapter 7`, *The Backup plan*:

```
$> tar -czf - LOCALFOLDER | ssh 'tar -xzvf-'
```

Running graphical commands on a remote machine

If you attempt to run a command on a remote machine that uses a graphical window, you will see an error similar to `cannot open display`. This is because the `ssh` shell is attempting (and failing) to connect to the X server on the remote machine.

How to do it...

To run an graphical application on a remote server, you need to set the `$DISPLAY` variable to force the application to connect to the X server on your local machine:

```
ssh user@host "export DISPLAY=:0 ; command1; command2"""
```

This will launch the graphical output on the remote machine.

If you want to show the graphical output on your local machine, use SSH's X11 forwarding option:

```
ssh -X user@host "command1; command2"
```

This will run the commands on the remote machine, but it will display graphics on your machine.

See also

- The *Password-less auto-login with SSH* recipe in this chapter explains how to configure auto-login to execute commands without prompting for a password

Transferring files through the network

A major use for networking computers is resource sharing. Files are a common shared resource. There are different methods for transferring files between systems, ranging from a USB stick and `sneakernet` to network links such as NFS and Samba. These recipes describe how to transfer files using the common protocols FTP, SFTP, RSYNC, and SCP.

Getting ready

The commands for performing file transfer over the network are mostly available by default with Linux installation. Files can be transferred via FTP using the traditional `ftp` command or the newer `lftp`, or via an SSH connection using `scp` or `sftp`. Files can be synchronized across systems with the `rsync` command.

How to do it...

File Transfer Protocol (**FTP**) is old and is used in many public websites to share files. The service usually runs on port `21`. FTP requires that an FTP server be installed and running on the remote machine. We can use the traditional `ftp` command or the newer `lftp` command to access an FTP-enabled server. The following commands are supported by both `ftp` and `lftp`. FTP is used in many public websites to share files.

To connect to an FTP server and transfer files to and from it, use the following command:

```
$ lftpusername@ftphost
```

It will prompt for a password and then display a logged in prompt:

```
lftp username@ftphost:~>
```

You can type commands in this prompt, as shown here:

- `cd directory`: This will change directory on the remote system
- `lcd`: This will change the directory on the local machine
- `mkdir`: This will create a directory on the remote machine
- `ls`: This will list files in the current directory on the remote machine

- `get FILENAME`: This will download a file to the current directory on the local machine:

```
lftp username@ftphost:~> get filename
```

- `put filename`: This will upload a file from the current directory on the remote machine:

```
lftp username@ftphost:~> put filename
```

- The `quit` command will terminate an `lftp` session

Autocompletion is supported in the `lftp` prompt

There's more...

Let's go through additional techniques and commands used for file transfer through a network.

Automated FTP transfer

The `lftp` and the `ftp` commands open an interactive session with the user. We can automate FTP file transfers with a shell script:

```
#!/bin/bash

#Automated FTP transfer
HOST=example.com'
USER='foo'
PASSWD='password'
lftp  -u ${USER}:${PASSWD} $HOST <<EOF

binary
cd /home/foo
put testfile.jpg

quit
EOF
```

The preceding script has the following structure:

```
<<EOF
DATA
EOF
```

This is used to send data through `stdin` to the `lftp` command. The *Playing with file descriptors and redirection* recipe of `Chapter 1`, *Shell Something Out*, explains various methods for redirection to *stdin*.

The `-u` option logs in to the remote site with our defined `USER` and `PASSWD`. The `binary` command sets the file mode to binary.

SFTP (Secure FTP)

SFTP is a file transfer system that runs on the top of an SSH connection and emulates an FTP interface. It requires an SSH server on the remote system instead of an FTP server. It provides an interactive session with an `sftp` prompt.

Sftp supports the same commands as `ftp` and `lftp`.

To start an `sftp` session, use the following command:

```
$ sftp user@domainname
```

Similar to `lftp`, an `sftp` session can be terminated by typing the `quit` command.

Sometimes, the SSH server will not be running at the default port `22`. If it is running at a different port, we can specify the port along with `sftp` as `-oPort=PORTNO`. Consider this example:

```
$ sftp -oPort=422 user@slynux.org
```

`-oPort` should be the first argument of the `sftp` command.

The rsync command

The `rsync` command is widely used for copying files over networks and for taking backup snapshots. This is described in detail in the *Backing up snapshots with rsync*

recipe of `Chapter 7`, *The Backup Plan*.

SCP (secure copy program)

SCP is a secure file copy command similar to the older, insecure remote copy tool called `rcp`. The files are transferred through an encrypted channel using SSH:

```
$ scp filename user@remotehost:/home/path
```

This will prompt for a password. Like `ssh`, the transfer can be made password-less with the auto-login SSH technique. The *Password-less auto-login with SSH* recipe in this chapter explains SSH auto-login. Once SSH login is automated, the scp command can be executed without an interactive password prompt.

The `remotehost` can be an IP address or domain name. The format of the `scp` command is as follows:

```
$ scp SOURCE DESTINATION
```

`SOURCE` or `DESTINATION` can be in the format `username@host:/path`:

```
$ scp user@remotehost:/home/path/filename filename
```

The preceding command copies a file from the remote host to the current directory with the given filename.

If SSH is running at a different port than `22`, use `-oPort` with the same syntax, `sftp`.

Recursive copying with scp

The `-r` parameter tells `scp` to recursively copy a directory between two machines:

```
$ scp -r /home/usernameuser@remotehost:/home/backups
# Copies the directory /home/usernameto the remote backup
```

The `-p` parameter will cause `scp` to retain permissions and modes when copying files.

See also

- The *Playing with file descriptors and redirection* recipe in `Chapter 1`, *Shell Something Out*, explains the standard input using EOF

Connecting to a wireless network

An Ethernet connection is simple to configure, since it is connected through wired cables with no special requirements like authentication. However, wireless LAN requires an **Extended Service Set IDentification** network identifier (**ESSID**) and may also require a pass-phrase.

Getting ready

To connect to a wired network, we simply assign an IP address and subnet mask with the `ifconfig` utility. A wireless network connection requires the `iwconfig` and `iwlist` utilities.

How to do it...

This script will connect to a wireless LAN with **WEP** (**Wired Equivalent Privacy**):

```
#!/bin/bash
#Filename: wlan_connect.sh
#Description: Connect to Wireless LAN

#Modify the parameters below according to your settings
######### PARAMETERS ###########
IFACE=wlan0
IP_ADDR=192.168.1.5
SUBNET_MASK=255.255.255.0
GW=192.168.1.1
HW_ADDR='00:1c:bf:87:25:d2'
#Comment above line if you don't want to spoof mac address

ESSID="homenet"
WEP_KEY=8b140b20e7
FREQ=2.462G
################################

KEY_PART=""

if [[ -n $WEP_KEY ]];
then
  KEY_PART="key $WEP_KEY"
fi

if [ $UID -ne 0 ];
then
  echo "Run as root"
  exit 1;
fi

# Shut down the interface before setting new config
/sbin/ifconfig $IFACE down


if [[ -n $HW_ADDR  ]];
```

```
then
  /sbin/ifconfig $IFACE hw ether $HW_ADDR
  echo Spoofed MAC ADDRESS to $HW_ADDR
fi

/sbin/iwconfig $IFACE essid $ESSID $KEY_PART freq $FREQ

/sbin/ifconfig $IFACE $IP_ADDR netmask $SUBNET_MASK

route add default gw $GW $IFACE

echo Successfully configured $IFACE
```

How it works...

The `ifconfig`, `iwconfig`, and `route` commands must be run as root. Hence, a check for the root user is performed before performing any actions in the scripts.

Wireless LAN requires parameters such as `essid`, `key`, and `frequency` to connect to the network. `essid` is the name of the wireless network to connect to. Some networks use a WEP key for authentication, which is usually a five- or ten-letter hex passphrase. The frequency assigned to the network is required by the `iwconfig` command to attach the wireless card with the proper wireless network.

The `iwlist` utility will scan and list the available wireless networks:

```
# iwlist scan
wlan0     Scan completed :
          Cell 01 - Address: 00:12:17:7B:1C:65
                    Channel:11
                    Frequency:2.462 GHz (Channel 11)
                    Quality=33/70  Signal level=-77 dBm
                    Encryption key:on
                    ESSID:"model-2"
```

The `Frequency` parameter can be extracted from the scan result, from the `Frequency:2.462 GHz (Channel 11)` line.

WEP is used in this example for simplicity. Note that WEP is insecure. If you are administering the wireless network, use a variant of **Wi-Fi Protected Access2** (**WPA2**).

See also

- The *Comparisons and tests* recipe of `Chapter 1`, *Shell Something Out*, explains string comparisons

Password-less auto-login with SSH

SSH is widely used with automation scripting, as it makes it possible to remotely execute commands at remote hosts and read their outputs. Usually, SSH is authenticated with username and password, which are prompted during the execution of SSH commands. Providing passwords in automated scripts is impractical, so we need to automate logins. SSH has a feature which SSH allows a session to auto-login. This recipe describes how to create SSH keys for auto-login.

Getting ready

SSH uses an encryption technique called asymmetric keys consisting of two keys–a public key and a private key for automatic authentication. The `ssh-keygen` application creates an authentication key pair. To automate the authentication, the public key must be placed on the server (by appending the public key to the `~/.ssh/authorized_keys` file) and the private key file of the pair should be present at the `~/.ssh` directory of the user at the client machine. SSH configuration options (for example, path and name of the `authorized_keys` file) can be modified by altering the `/etc/ssh/sshd_config` configuration file.

How to do it...

There are two steps to implement automatic authentication with SSH. They are as follows:

- Creating the SSH key on the local machine
- Transferring the public key to the remote host and appending it to `~/.ssh/authorized_keys` (which requires access to the remote machine)

To create an SSH key, run the `ssh-keygen` command with the encryption algorithm type specified as RSA:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/username/.ssh/id_rsa):
```

```
Created directory '/home/username/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/username/.ssh/id_rsa.
Your public key has been saved in /home/username/.ssh/id_rsa.pub.
The key fingerprint is:
f7:17:c6:4d:c9:ee:17:00:af:0f:b3:27:a6:9c:0a:05 username@slynux-laptop
The key'srandomart image is:
+--[ RSA 2048]----+
|            .    |
|             o . .|
|     E        o o.|
|      ...oo |
|       .S .+  +o.|
|      .  . .=....|
|     .+.o...|
|      . . + o.  .|
|       ..+       |
+-----------------+
```

You need to enter a passphrase to generate the public-private key pair. It is possible to generate the key pair without entering a passphrase, but it is insecure.

If you intend to write scripts that use automated login to several machines, you should leave the passphrase empty to prevent the script from asking for a passphrase while running.

The `ssh-keygen` program creates two files. `~/.ssh/id_rsa.pub` and `~/.ssh/id_rsa:id_rsa.pub` is the generated public key and `id_rsa` is the private key. The public key has to be appended to the `~/.ssh/authorized_keys` file on remote servers where we need to auto-login from the current host.

This command will append a key file:

```
$ ssh USER@REMOTE_HOST \
    "cat >> ~/.ssh/authorized_keys" < ~/.ssh/id_rsa.pub
Password:
```

Provide the login password in the previous command.

The auto-login has been set up from now onwards, so SSH will not prompt for passwords during execution. Test this with the following command:

```
$ ssh USER@REMOTE_HOST uname
Linux
```

You will not be prompted for a password. Most Linux distros include `ssh-copy-id`, which will append your private key to the appropriate `authorized_keys` file on the remote server. This is shorter than the `ssh` technique described earlier:

```
ssh-copy-id USER@REMOTE_HOST
```

Port forwarding using SSH

Port forwarding is a technique which redirects an IP connection from one host to another. For example, if you are using a Linux/Unix system as a firewall you can redirect connections to port `1234` to an internal address such as `192.168.1.10:22` to provide an `ssh` tunnel from the outside world to an internal machine.

How to do it...

You can forward a port on your local machine to another machine and it's also possible to forward a port on a remote machine to another machine. In the following examples, you will get a shell prompt once the forwarding is complete. Keep this shell open to use the port forward and exit it whenever you want to stop the port forward.

1. This command will forward port `8000` on your local machine to port `80` on `www.kernel.org`:

   ```
   ssh -L 8000:www.kernel.org:80user@localhost
   ```

   Replace user with the username on your local machine.

2. This command will forward port 8000 on a remote machine to port `80` of `www.kernel.org`:

   ```
   ssh -L 8000:www.kernel.org:80user@REMOTE_MACHINE
   ```

   Here, replace `REMOTE_MACHINE` with the hostname or IP address of the remote machine and `user` with the username you have SSH access to.

There's more...

Port forwarding is more useful when using non-interactive mode or reverse port forwarding.

Non-interactive port forward

If you want to just set port forwarding instead of having a shell kept open while port forwarding is effective, use the following form of `ssh`:

```
ssh -fL8000:www.kernel.org:80user@localhost -N
```

The `-f` option instructs `ssh` to fork to background before executing the command. `-N` tells `ssh` that there is no command to run; we only want to forward ports.

Reverse port forwarding

Reverse port forwarding is one of the most powerful features of SSH. This is most useful in situations where you have a machine which isn't publicly accessible from the Internet, but you want others to be able to access a service on this machine. In this case, if you have SSH access to a remote machine which is publicly accessible on the Internet, you can set up a reverse port forward on that remote machine to the local machine which is running the service.

```
ssh -R 8000:localhost:80 user@REMOTE_MACHINE
```

This command will forward port `8000` on the remote machine to port `80` on the local machine. Don't forget to replace `REMOTE_MACHINE` with the hostname of the IP address of the remote machine.

Using this method, if you browse to `http://localhost:8000` on the remote machine, you will connect to a web server running on port `80` of the local machine.

Mounting a remote drive at a local mount point

Having a local mount point to access the remote host filesystem facilitates read and write data transfer operations. SSH is the common transfer protocol. The `sshfs` application uses SSH to enable you to mount a remote filesystem on a local mount point.

Getting ready

`sshfs` doesn't come by default with GNU/Linux distributions. Install `sshfs` with a package manager. `sshfs` is an extension to the FUSE filesystem package that allows users to mount a wide variety of data as if it were a local filesystem. Variants of FUSE are supported on Linux, Unix, Mac OS/X, Windows, and more.

For more information on FUSE, visit its website at `http://fuse.sourceforge.net/`.

How to do it...

To mount a filesystem location at a remote host to a local mount point:

```
# sshfs -o allow_otheruser@remotehost:/home/path /mnt/mountpoint
Password:
```

Issue the password when prompted. After the password is accepted, the data at `/home/path` on the remote host can be accessed via a local mount point, `/mnt/mountpoint`.

To unmount, use the following command:

```
# umount /mnt/mountpoint
```

See also

- The *Running commands on a remote host with SSH* recipe in this chapter explains the `ssh` command

Network traffic and port analysis

Every application that accesses the network does it via a port. Listing the open ports, the application using a port and the user running the application is a way to track the expected and unexpected uses of your system. This information can be used to allocate resources as well as checking for rootkits or other malware.

Getting ready

Various commands are available for listing ports and services running on a network node. The `lsof` and `netstat` commands are available on most GNU/Linux distributions.

How to do it...

The `lsof` (list open files) command will list open files. The `-i` option limits it to open network connections:

```
$ lsof -i
COMMAND    PID   USER   FD   TYPE DEVICE SIZE/OFF NODE
    NAME

firefox-b 2261 slynux   78u  IPv4  63729      0t0  TCP
    localhost:47797->localhost:42486 (ESTABLISHED)

firefox-b 2261 slynux   80u  IPv4  68270      0t0  TCP
    slynux-laptop.local:41204->192.168.0.2:3128 (CLOSE_WAIT)

firefox-b 2261 slynux   82u  IPv4  68195      0t0  TCP
    slynux-laptop.local:41197->192.168.0.2:3128 (ESTABLISHED)

ssh       3570 slynux    3u  IPv6  30025      0t0  TCP
    localhost:39263->localhost:ssh (ESTABLISHED)

ssh       3836 slynux    3u  IPv4  43431      0t0  TCP
    slynux-laptop.local:40414->boney.mt.org:422 (ESTABLISHED)

GoogleTal 4022 slynux   12u  IPv4  55370      0t0  TCP
    localhost:42486 (LISTEN)

GoogleTal 4022 slynux   13u  IPv4  55379      0t0  TCP
    localhost:42486->localhost:32955 (ESTABLISHED)
```

Each entry in the output of `lsof` corresponds to a service with an active network port. The last column of output consists of lines similar to this:

```
laptop.local:41197->192.168.0.2:3128
```

In this output, `laptop.local:41197` corresponds to the `localhost` and `192.168.0.2:3128` corresponds to the remote host. `41197` is the port used on the current machine, and `3128` is the port to which the service connects at the remote host.

To list the opened ports from the current machine, use the following command:

```
$ lsof -i | grep ":[0-9a-z]+->" -o | grep "[0-9a-z]+" -o  | sort | uniq
```

How it works...

The `:[0-9a-z]+->` regex for grep extracts the host port portion `(:34395-> or :ssh->)` from the `lsof` output. The next `grep` removes the leading colon and trailing arrow leaving the port number (which is alphanumeric). Multiple connections may occur through the same port and hence, multiple entries of the same port may occur. The output is sorted and passed through `uniq` to display each port only once.

There's more...

There are more utilities that report open port and network traffic related information.

Opened port and services using netstat

`netstat` also returns network service statistics. It has many features beyond what is covered in this recipe.

Use `netstat -tnp` to list opened port and services:

```
$ netstat -tnp
Proto Recv-Q Send-Q Local Address           Foreign Address
    State       PID/Program name

tcp        0      0 192.168.0.82:38163      192.168.0.2:3128
    ESTABLISHED 2261/firefox-bin

tcp        0      0 192.168.0.82:38164      192.168.0.2:3128
    TIME_WAIT   -

tcp        0      0 192.168.0.82:40414      193.107.206.24:422
    ESTABLISHED 3836/ssh

tcp        0      0 127.0.0.1:42486         127.0.0.1:32955
    ESTABLISHED 4022/GoogleTalkPlug

tcp        0      0 192.168.0.82:38152      192.168.0.2:3128
    ESTABLISHED 2261/firefox-bin
```

```
tcp6       0      0 ::1:22                  ::1:39263
    ESTABLISHED -

tcp6       0      0 ::1:39263               ::1:22
    ESTABLISHED 3570/ssh
```

Measuring network bandwidth

The previous discussion of `ping` and `traceroute` was on measuring the latency of a network and the number of hops between nodes.

The `iperf` application provides more metrics for a networks' performance. The `iperf` application is not installed by default, but it is provided by most distributions' package manager.

How to do it...

The `iperf` application must be installed on both ends of a link (a host and a client). Once `iperf` is installed, start the server end:

```
$ iperf -s
```

Then run the client side to generate throughput statistics:

```
$ iperf -c 192.168.1.36
------------------------------------------------------------
Client connecting to 192.168.1.36, TCP port 5001
TCP window size: 19.3 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.44 port 46526 connected with 192.168.1.36 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   113 MBytes  94.7 Mbits/sec
```

The `-m` option instructs `iperf` to also find the **Maximum Transfer Size** (**MTU**):

```
$ iperf -mc 192.168.1.36
------------------------------------------------------------
Client connecting to 192.168.1.36, TCP port 5001
TCP window size: 19.3 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.44 port 46558 connected with 192.168.1.36 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   113 MBytes  94.7 Mbits/sec
[  3] MSS size 1448 bytes (MTU 1500 bytes, ethernet)
```

Creating arbitrary sockets

For operations such as file transfer and secure shell, there are prebuilt tools such as ftp and `ssh`. We can also write custom scripts as network services. The next recipe demonstrates how to create simple network sockets and use them for communication.

Getting ready

The `netcat` or `nc` command will create network sockets to transfer data over a TCP/IP network. We need two sockets: one listens for connections and the other connects to the listener.

How to do it...

1. Set up the listening socket using the following command:

   ```
   nc –l 1234
   ```

   This will create a listening socket on port `1234` on the local machine.

2. Connect to the socket using the following command:

   ```
   nc HOST 1234
   ```

   If you are running this on the same machine as the listening socket, replace `HOST` with localhost, otherwise replace it with the IP address or hostname of the machine.

3. Type something and press *Enter* on the terminal where you performed step 2. The message will appear on the terminal where you performed step 1.

There's more...

Network sockets can be used for more than just text communication, as shown in the following sections.

Quickly copying files over the network

We can exploit `netcat` and shell redirection to copy files over the network. This command will send a file to the listening machine:

1. On the listening machine, run the following command:

```
nc -l 1234 >destination_filename
```

2. On the sender machine, run the following command:

```
nc HOST 1234 <source_filename
```

Creating a broadcasting server

You can use `netcan` to create a custom server. The next recipe demonstrates a server that will send the time every 10 seconds. The time can be received by connecting to the port with a client `nc` session of telnet:

```
# A script to echo the date out a port
while [ 1 ]
do
  sleep 10
  date
done | nc -l 12345
echo exited
```

How it works...

Copying files with `nc` works because ns echoes the input from the input of one socket to the output at the other.

The broadcasting server is a bit more complicated. The `while [ 1 ]` loop will run forever. Within the loop, the script sleeps for 10 seconds, then invokes the date command and pipes the output to the `nc` command.

You can use `nc` to create a client, as follows:

```
$ nc 127.0.0.1 12345
```

Building a bridge

If you have two separate networks, you may need a way to pass data from one network to the other. This is commonly done by connecting the two subnets with a router, hub, or switch.

A Linux system can be used for a network bridge.

A bridge is a low-level connection that passes packets based on their MAC address instead of being identified by the IP address. As such it requires fewer machine resources and is more efficient.

You can use a bridge to link virtual machines on private, non-routed networks, or to link separate subnets in a company, for instance, to link a manufacturing subnet to the shipping sub-net so production information can be shared.

Getting ready

The Linux kernel has supported network bridges since the 2.2 kernel. The current tool to define a bridge, is the iproute2 (`ip`) command. This is standard in most distributions.

How to do it...

The ip command performs several actions using the command/subcommand model. To create a bridge, we use the `ip link` commands.

The Ethernet adapter being attached to the bridge should not be configured with an IP address when it is added to the bridge. The bridge is configured with an address, not the NIC.

In this example, there are two NIC cards: `eth0` is configured and connected to the `192.168.1.0` subnet, while eth1 is not configured but will be connected to the `10.0.0.0` subnet via the bridge:

```
# Create a new bridge named br0
ip link add br0 type bridge

# Add an Ethernet adapter to the bridge
ip link set dev eth1 master br0

# Configure the bridge's IP address
```

```
ifconfig br0 10.0.0.2

# Enable packet forwarding
echo 1 >/proc/sys/net/ipv4/ip_forward
```

This creates the bridge allowing packets to be sent from `eth0` to `eth1` and back. Before the bridge can be useful, we need to add this bridge to the routing tables.

On machines in the `10.0.0.0/24` network, we add a route to the `192.168.1.0/16` network:

```
route add –net 192.168.1.0/16 gw 10.0.0.2
```

Machines on the `192.168.1.0/16` subnet need to know how to find the `10.0.0.0/24` subnet. If the `eth0` card is configured for IP address `192.168.1.2`, the route command is as follows:

```
route add –net 10.0.0.0/24 gw 192.168.1.2
```

Sharing an Internet connection

Most firewall/routers have the ability to share an Internet connection with the devices in your home or office. This is called **Network Address Translation** (**NAT**). A Linux computer with two **Network Interface Cards** (**NIC**) can act as a router, providing firewall protection and connection sharing.

Firewalling and NAT support are provided by the support for iptables built into the kernel. This recipe introduces `iptables` with a recipe that shares a computer's Ethernet link to the Internet through the wireless interface to give other wireless devices access to the Internet via the host's Ethernet NIC.

Getting ready

This recipe uses `iptables` to define a **Network Address Translation** (**NAT**), which lets a networking device share a connection with other devices. You will need the name of your wireless interface, which is reported by the `iwconfig` command.

How to do it...

1. Connect to the Internet. In this recipe, we are assuming that the primary wired network connection, `eth0`, is connected to the Internet. Change it according to your setup.
2. Using your distro's network management tool, create a new ad hoc wireless connection with the following settings:
    - IP address: 10.99.66.55
    - Subnet mask: 255.255.0.0 (16)
3. Use the following shell script to share the Internet connection:

```
#!/bin/bash
#filename: netsharing.sh

echo 1 > /proc/sys/net/ipv4/ip_forward

iptables -A FORWARD -i $1 -o $2 \
    -s 10.99.0.0/16 -m conntrack --ctstate NEW -j ACCEPT

iptables -A FORWARD -m conntrack --ctstate \
    ESTABLISHED,RELATED -j ACCEPT

iptables -A POSTROUTING -t nat -j MASQUERADE
```

4. Run the script:

```
./netsharing.sh eth0 wlan0
```

   Here `eth0` is the interface that is connected to the Internet and `wlan0` is the wireless interface that is supposed to share the Internet with other devices.

5. Connect your devices to the wireless network you just created with the following settings:
    - IP address: 10.99.66.56 (and so on)
    - Subnet mask: 255.255.0.0

To make this more convenient, you might want to install a DHCP and DNS server on your machine, so it's not necessary to configure IPs on devices manually. A handy tool for this is `dnsmasq`, which performs both DHCP and DNS operations.

How it works

There are three sets of IP addresses set aside for non-routing use. That means that no network interface visible to the Internet can use them. They are only used by machines on a local, internal network. The addresses are `10.x.x.x`, `192.168.x.x`, and `172.16.x.x->172.32.x.x`. In this recipe, we use a portion of the `10.x.x.x` address space for our internal network.

By default, Linux systems will accept or generate packets, but will not echo them. This is controlled by the value `in/proc/sys/net/ipv4/ip_forward`.

Echoing a `1` to that location tells the Linux kernel to forward any packet it doesn't recognize. This allows the wireless devices on the `10.99.66.x` subnet to use `10.99.66.55` as their gateway. They will send a packet destined for an Internet site to `10.99.66.55`, which will then forward it out its gateway on `eth0` to the Internet to be routed to the destination.

The `iptables` command is how we interact with the Linux kernel's iptables subsystem. These commands add rules to forward all packets from the internal network to the outside world and to forward expected packets from the outside world to our internal network.

The next recipe will discuss more ways to use iptables.

Basic firewall using iptables

A firewall is a network service that is used to filter network traffic for unwanted traffic, block it, and allow the desired traffic to pass. The standard firewall tool for Linux is `iptables`, which is integrated into the kernel in recent versions.

How to do it...

`iptables` is present by default on all modern Linux distributions. It's easy to configure for common scenarios:

1. If don't want to contact a given site (for example, a known malware site), you can block traffic to that IP address:

```
#iptables –A OUTPUT –d 8.8.8.8 –j DROP
```

If you use `PING 8.8.8.8` in another terminal, then by running the `iptables` command, you will see this:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=56 time=221 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=56 time=221 ms
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
```

Here, the ping fails the third time because we used the `iptables` command to drop all traffic to `8.8.8.8`.

2. You can also block traffic to a specific port:

```
#iptables –A OUTPUT –p tcp –dport 21 –j DROP
$ ftp ftp.kde.org
ftp: connect: Connection timed out
```

If you find messages like this in your `/var/log/secure` or `var/log/messages` file, you have a small problem:

```
Failed password for abel from 1.2.3.4 port 12345 ssh2
Failed password for baker from 1.2.3.4 port 12345 ssh2
```

These messages mean a robot is probing your system for weak passwords. You can prevent the robot from accessing your site with an INPUT rule that will drop all traffic from that site.

```
#iptables –I INPUT –s 1.2.3.4 –j DROP
```

How it works...

`iptables` is the command used to configure the firewall on Linux. The first argument in `iptables` is -A, which instructs `iptables` to append a new rule to the chain, or -I, which places the new rule at the start of the ruleset. The next parameter defines the chain. A chain is a collection of rules, and in earlier recipes we used the `OUTPUT` chain, which is evaluated for outgoing traffic, whereas the last recipes used the `INPUT` chain, which is evaluated for incoming traffic.

The `-d` parameter specifies the destination to match with the packet being sent, and `-s` specifies the source of a packet. Finally, the `-j` parameter instructs `iptables` to jump to a particular action. In these examples, we used the DROP action to drop the packet. Other actions include `ACCEPT` and `REJECT`.

In the second example, we use the `-p` parameter to specify that this rule matches only TCP on the port specified with `-dport`. This blocks only the outbound `FTP` traffic.

There's more...

You can clear the changes made to the `iptables` chains with the `-flush` parameter:

```
#iptables -flush
```

Creating a Virtual Private Network

A **Virtual Private Network** (**VPN**) is an encrypted channel that operates across public networks. The encryption keeps your information private. VPNs are used to connect remote offices, distributed manufacturing sites, and remote workers.

We've discussed copying files with `nc`, or `scp`, or `ssh`. With a VPN network, you can mount remote drives via NFS and access resources on the remote network as if they were local.

Linux has clients for several VPN systems, as well as client and server support for OpenVPN.

This section's recipes will describe setting up an OpenVPN server and client. This recipe is to configure a single server to service multiple clients in a hub and spoke model. OpenVPN supports more topologies that are beyond the scope of this chapter.

Getting ready

OpenVPN is not part of most Linux distributions. You can install it using your package manager:

```
apt-get install openvpn
```

Alternatively, this command can also be used:

```
yum install openvpn
```

Note that you'll need to do this on the server and each client.

Confirm that the tunnel device (`/dev/net/tun`) exists. Test this on server and client systems. On modern Linux systems, the tunnel should already exist:

```
ls /dev/net/tun
```

How to do it...

The first step in setting up an OpenVPN network is to create the certificates for the server and at least one client. The simplest way to handle this is to make self-signed certificates with the `easy-rsa` package included with pre-version 2.3 releases of OpenVPN. If you have a later version of OpenVPN, `easy-rsa` should be available via the package manager.

This package is probably installed in `/usr/share/easy-rsa`.

Creating certificates

First, make sure you've got a clean slate with nothing left over from previous installations:

```
# cd /usr/share/easy-rsa
# . ./vars
# ./clean-all
```

NOTE: If you run `./clean-all`, I will be doing a `rm -rf` on `/usr/share/easy-rsa/keys`.

Next, create the **Certificate Authority** key with the `build-ca` command. This command will prompt you for information about your site. You'll have to enter this information several times. Substitute your name, e-mail, site name, and so on for the values in this recipe. The required information varies slightly between commands. Only the unique sections will be repeated in these recipes:

```
# ./build-ca
Generating a 2048 bit RSA private key
......+++
.....................................................+++
writing new private key to 'ca.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For somefieldsthere will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [CA]:MI
Locality Name (eg, city) [SanFrancisco]:WhitmoreLake
Organization Name (eg, company) [Fort-Funston]:Example
Organizational Unit Name (eg, section) [MyOrganizationalUnit]:Packt
Common Name (eg, your name or your server's hostname) [Fort-Funston
CA]:vpnserver
Name [EasyRSA]:
Email Address [me@myhost.mydomain]:admin@example.com

Next, build the server certificate with the build-key command:
# ./build-key server
Generating a 2048 bit RSA private key
..............................+++
.....................+++
writing new private key to 'server.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request....




Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
```

Create a certificate for at least one client. You'll need a separate client certificate for each machine that you wish to connect to this OpenVPN server:

```
# ./build-key client1
Generating a 2048 bit RSA private key
.....................+++
.................................................+++
writing new private key to 'client1.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
...
```

```
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
Using configuration from /usr/share/easy-rsa/openssl-1.0.0.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName  :PRINTABLE:'US'
stateOrProvinceName  :PRINTABLE:'MI'
localityName  :PRINTABLE:'WhitmoreLake'
organizationName  :PRINTABLE:'Example'
organizationalUnitName:PRINTABLE:'Packt'
commonName  :PRINTABLE:'client1'
name                  :PRINTABLE:'EasyRSA'
emailAddress:IA5STRING:'admin@example.com'
Certificate is to be certified until Jan  8 15:24:13 2027 GMT (3650 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Finally, generate the **Diffie-Hellman** with the `build-dh` command. This will take several seconds and will generate a few screens filled with dots and plusses:

```
# ./build-dh
Generating DH parameters, 2048 bit long safe prime, generator 2
This is going to take a long time
.....................+............+........
```

These steps will create several files in the keys folder. The next step is to copy them to the folders where they'll be used.

Copy server keys to `/etc/openvpn`:

```
# cp keys/server* /etc/openvpn
# cp keys/ca.crt /etc/openvpn
# cp keys/dh2048.pem /etc/openvpn
```

Copy the client keys to the client system:

```
# scp keys/client1* client.example.com:/etc/openvpn
# scp keys/ca.crt client.example.com:/etc/openvpn
```

Configuring OpenVPN on the server

OpenVPN includes sample configuration files that are almost ready to use. You only need to customize a few lines for your environment. The files are commonly found in `/usr/share/doc/openvpn/examples/sample-config-files`:

```
# cd /usr/share/doc/openvpn/examples/sample-config-files
# cp server.conf.gz /etc/openvpn
# cd /etc/openvpn
# gunzip server.conf.gz
# vim server.conf
```

Set the local IP address to listen on. This is the IP address of the NIC attached to the network you intend to allow VPN connections through:

```
local 192.168.1.125
Modify the paths to the certificates:

ca /etc/openvpn/ca.crt
cert /etc/openvpn/server.crt
key /etc/openvpn/server.key  # This file should be kept secret
```

Finally, check that the `diffie-hellman` parameter file is correct. The OpenVPN sample `config` file may specify a 1024-bit length key, while the `easy-rsa` creates a 2048-bit (more secure) key.

```
#dh dh1024.pem
dh dh2048.pem
```

Configuring OpenVPN on the client

There is a similar set of configurations to do on each client.

Copy the client configuration file to `/etc/openvpn`:

```
# cd /usr/share/doc/openvpn/examples/sample-config-files
# cpclient.conf /etc/openvpn
```

Edit the `client.conf` file:

```
# cd /etc/openvpn
# vim client.conf
```

Change the paths for the certificates to the point to correct folders:

```
ca /etc/openvpn/ca.crt
cert /etc/openvpn/server.crt
key /etc/openvpn/server.key  # This file should be kept secret
```

Set the remote site for your server:

```
#remote my-server-1 1194
remote server.example.com 1194
```

Starting the server

The server can be started now. If everything is configured correctly, you'll see it output several lines of output. The important line to look for is the `Initialization Sequence Completed` line. If that is missing, look for an error message earlier in the output:

```
# openvpnserver.conf
Wed Jan 11 12:31:08 2017 OpenVPN 2.3.4 x86_64-pc-linux-gnu [SSL (OpenSSL)]
[LZO] [EPOLL] [PKCS11] [MH] [IPv6] built on Nov 12 2015
Wed Jan 11 12:31:08 2017 library versions: OpenSSL 1.0.1t  3 May 2016, LZO
2.08...


Wed Jan 11 12:31:08 2017 client1,10.8.0.4
Wed Jan 11 12:31:08 2017 Initialization Sequence Completed
```

Using `ifconfig`, you can confirm that the server is running. You should see the tunnel device (tun) listed:

```
$ ifconfig
tun0      Link encap:UNSPECHWaddr
00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
inet addr:10.8.0.1  P-t-P:10.8.0.2  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Starting and testing a client

Once the server is running, you can start a client. Like the server, the client side of OpenVPN is created with the `openvpn` command. Again, the important part of this output is the `Initialization Sequence Completed` line:

```
# openvpn client.conf
Wed Jan 11 12:34:14 2017 OpenVPN 2.3.4 i586-pc-linux-gnu [SSL (OpenSSL)]
[LZO] [EPOLL] [PKCS11] [MH] [IPv6] built on Nov 19 2015
Wed Jan 11 12:34:14 2017 library versions: OpenSSL 1.0.1t  3 May 2016, LZO
2.08...




Wed Jan 11 12:34:17 2017 /sbin/ipaddr add dev tun0 local 10.8.0.6 peer
10.8.0.5
Wed Jan 11 12:34:17 2017 /sbin/ip route add 10.8.0.1/32 via 10.8.0.5
Wed Jan 11 12:34:17 2017 Initialization Sequence Completed
```

Using the `ifconfig` command, you can confirm that the tunnel has been initialized:

```
$ /sbin/ifconfig

tun0      Link encap:UNSPECHWaddr 00-00-00-00-00-00-00-00...00-00-00-00
inet addr:10.8.0.6  P-t-P:10.8.0.5  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:168 (168.0 B)  TX bytes:336 (336.0 B)
```

Use the `netstat` command to confirm that the new network is routed correctly:

```
$ netstat -rn
Kernel IP routing table
Destination   Gateway       Genmask         Flags   MSS Window  irttIface
0.0.0.0       192.168.1.7   0.0.0.0         UG        0 0          0 eth0
10.8.0.1      10.8.0.5      255.255.255.255 UGH       0 0          0 tun0
10.8.0.5      0.0.0.0       255.255.255.255 UH        0 0          0 tun0
192.168.1.0   0.0.0.0       255.255.255.0   U         0 0          0 eth0
```

This output shows the tunnel device connected to the `10.8.0.x` network, and the gateway is `10.8.0.1`.

Finally, you can test connectivity with the `ping` command:

```
$ ping 10.8.0.1
PING 10.8.0.1 (10.8.0.1) 56(84) bytes of data.
64 bytes from 10.8.0.1: icmp_seq=1 ttl=64 time=1.44 ms
```