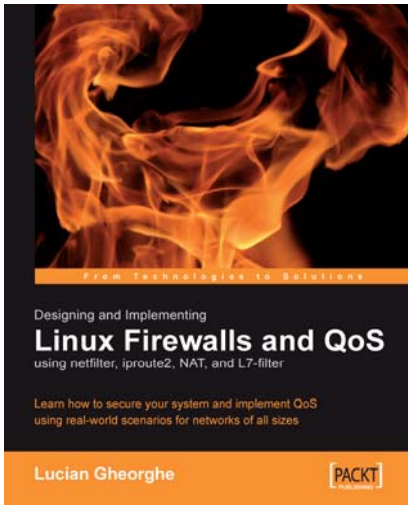
Designing and Implementing Linux Firewalls and QoS using netfilter, iproute2, NAT and l7-filter

Learn how to secure your system and implement QoS using real-world scenarios for networks of all sizes

Lucian Gheorghe



Chapter 3

"Prerequisites: netfilter and iproute2"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter 3 "Prerequisites: netfilter and iproute2"

A synopsis of the book's content

Information on where to buy this book

About the Author

Lucian Gheorghe

Lucian Gheorghe has just joined the Global NOC of Interoute, Europe's largest voice and data network provider. Before Interoute, he was working as a senior network engineer for Globtel Internet, a significant Internet and Telephony Services Provider to the Romanian market. He has been working with Linux for more than 8 years putting a strong accent on security for protecting vital data from hackers and ensuring good quality services for internet customers. Moving to VoIP services he had to focus even more on security as sensitive billing data is most often stored on servers with public IP addresses. He has been studying QoS implementations on Linux to build different types of services for IP customers and also to deliver good quality for them and for VoIP over the public Internet. Lucian has also been programming with Perl, PHP, and Smarty for over 5 years mostly developing in-house management interfaces for IP and VoIP services.

I would like to thank everyone who is reading this book and the people that run netfilter, iproute2, and L7-filter projects.Your feedback is very important to me, so drop me a line at lucian.firewallbook@gmail.com. The book is far from being perfect so please send me errata information on the same email address (I would love to receive erratas from readers because it will convince me that people who read this book actually learned something :-))

I want to dedicate this book to my father, my mother, and my sister—I love you very very much. Many thanks go to the team at Globtel who were like second family to me, to my girlfriend for understanding me and standing by me, to Louay and the rest of the team at Packt Publishing for doing a great job, to Nigel Coulson, Petr Klobasa and the rest of the people at Interoute for supporting me, to Claudiu Filip who is one of the most intelligent people I know, and last, but not least, to the greatest technical author alive—Cristian Darie.

About the Reviewers

Barrie Dempster

Barrie Dempster is currently employed as a Senior Security Consultant for NGS Software Ltd, a world-renowned security consultancy well known for its focus in enterprise-level application vulnerability research and database security. He has a background in Infrastructure and Information Security in a number of specialized environments such as financial services institutions, telecommunications companies, call centers, and other organizations across multiple continents. Barrie has experience in the integration of network infrastructure and telecommunications systems requiring high-caliber secure design, testing, and management. He has been involved in a variety of projects from the design and implementation of Internet banking systems to large-scale conferencing and telephony infrastructure, as well as penetration testing and other security assessments of business-critical infrastructure.

Designing and Implementing Linux Firewalls and QoS using netfilter, iproute2, NAT and l7-filter

A networking firewall is a logical barrier designed to prevent unauthorized or unwanted communications between sections of a computer network. Linux-based firewalls besides being highly customizable and versatile are also robust, inexpensive, and reliable.The two things needed to build firewalls and QoS with Linux are two packages named netfilter and iproute. While netfilter is a packet-filtering framework included in the Linux kernels 2.4 and 2.6, iproute is a package containing a few utilities that allow Linux users to do advanced routing and traffic shaping. L7-filter is a packet classifier for the Linux kernel that doesn't look up port numbers or Layer 4 protocols, but instead looks at the data in an IP packet and does a regular expression match on it to determine what kind of data it is, mainly what application protocol is being used. IP2P is an alternative to L7-filter, but has been designed for filtering only P2P applications while L7-filter takes into consideration a wider range of applications.

What This Book Covers

Chapter 1 is a brief introduction to networking concepts. It covers the OSI and TCP/IP networking models with explanations of their layers, TCP and UDP as Layer 4 protocols, and then rounds off the chapter with a discussion on IP addresses, Subnetting, and Supernetting.

Chapter 2 discusses possible security threats and vulnerabilities found at each of the OSI layers. The goal here is to understand where and how these threats can affect us and to stay protected from attackers. It then rounds off the discussion by sketching out the basic steps required to protect the services that run on our system.

Chapter 3 introduces two tools needed to build Linux firewalls and QoS. We first learn the workings of netfilter, which is a packet-filtering framework, and implement what we have learned to build a basic firewall for a Linux workstation. We then see how to perform advanced routing and traffic shaping using the IP and TC tools provided by the iproute2 package. The chapter ends with another example scenario where we implement the concepts learned in the chapter.

Chapter 4 discusses NAT, the types of NAT, how they work, and how they can be implemented with Linux by giving practical examples. It also describe packet mangling, when to use it, and why to use it.

Chapter 5 covers Layer 7 filtering in detail. We see how to install the L7-filter package, apply the necessary Linux kernel and iptables patches, and test our installation. We then learn the different applications of L7-filter and see how to put them to practical use. We also see how to install and use IPP2P, which is an alternative to the L7-filter package, but only for P2P traffic, and finally we set up a test between the two packages.

Chapter 6 raises two very popular scenarios, for which we design, implement, and test firewalls and a small QoS configuration. In the first scenario, we configure Linux as a SOHO router. Being a relatively smaller network with few devices, we learn how to adapt to what we have learned in the earlier chapters to suit this environment and build a secure network. We implement transparent proxies using squid and iptables so that children/minors cannot access malicious or pornographic web content. Our firewall setup implements NAT to redirect traffic from certain ports to other hosts using Linux. This configuration is tested by checking the NAT table and seeing how the kernel analyzes our rules.As part of QoS, we split the bandwidth between the devices in a SOHO environment using HTB. Assuming a 1Mbps connection, we design a policy to split it between the 4 devices creating 4 HTB child classes for the 4 devices. In the end, we test our QoS configuration using the tc class show command.In the second scenario, we configure Linux as router for a typical small to medium company.

Chapter 7 covers the design of a firewall system for a hypermarket having its headquarters in one location, one store in the same city, and several stores in other cities. The hypermarket has an application that uses MSSQL databases in each location, which are replicated at the headquarters. All locations have IP Analog Telephone Adapters with subscriptions at the main provider (the HQ provider). In this example we use, just like in the real H.323 as the VoIP protocol. We set up all remote locations to have an encrypted VPN connection using ip tunnel to connect to the headquarters. Users are shown how to create a QOS script with HTB that controls bandwidth usage based on priorities.

The next firewall taken up is that for a small ISP setup that has one internet connection, an access network, a server farm, and the internal departments. The setup of firewall scripts for each of them and methods to handle the tricky wireless server are covered. The QoS is handled by the intranet server, the wireless server, and the Core router.

Chapter 8 covers the design of a three-layered network deployed at a large provider of Internet and IP telephony services, the three layers being Core, Distribution, and Access. It explains network configuration first on the core and distribution levels and then moves on to building firewalls. The huge size of the network also means that there is a need to tackle newer security threats. We have four Cores running BGP under Zebra and each one is peculiar in its own way. There are three data services that this ISP can provide to its customers: Internet access, national network access, and metropolitan network access. This chapter will show you how to handle QoS so as to limit this traffic as needed.

3
Prerequisites: netfilter and iproute2

The two things needed to build firewalls and Quality of Service (QoS) with Linux are two packages named netfilter and iproute. While netfilter is a packet filtering framework included in the Linux kernels 2.4 and 2.6, iproute is a package containing a few utilities that allow Linux users to do advanced routing and traffic shaping.

This chapter is intended to introduce the tools we will use throughout this book. However, netfilter and iproute are very large subjects; so what I'll try to do in this chapter is to introduce readers who are not familiar with the subject, along with building a nice overview for readers who already know the subject.

There are two websites with a lot of documentation on both projects—for netfilter, `http://www.netfilter.org`, and for iproute, `http://www.lartc.org`.

netfilter/iptables

netfilter is a very important part of the Linux kernel in terms of security, packet mangling, and manipulation. The front end for netfilter is iptables, which "tells" the kernel what the user wants to do with the IP packets arriving into, passing through, or leaving the Linux box.

The most used features of netfilter are packet filtering and network address translation, but there are a lot of other things that we can do with netfilter, such as packet mangling Layer 7 filtering.

A rough explanation on how netfilter works is like this:

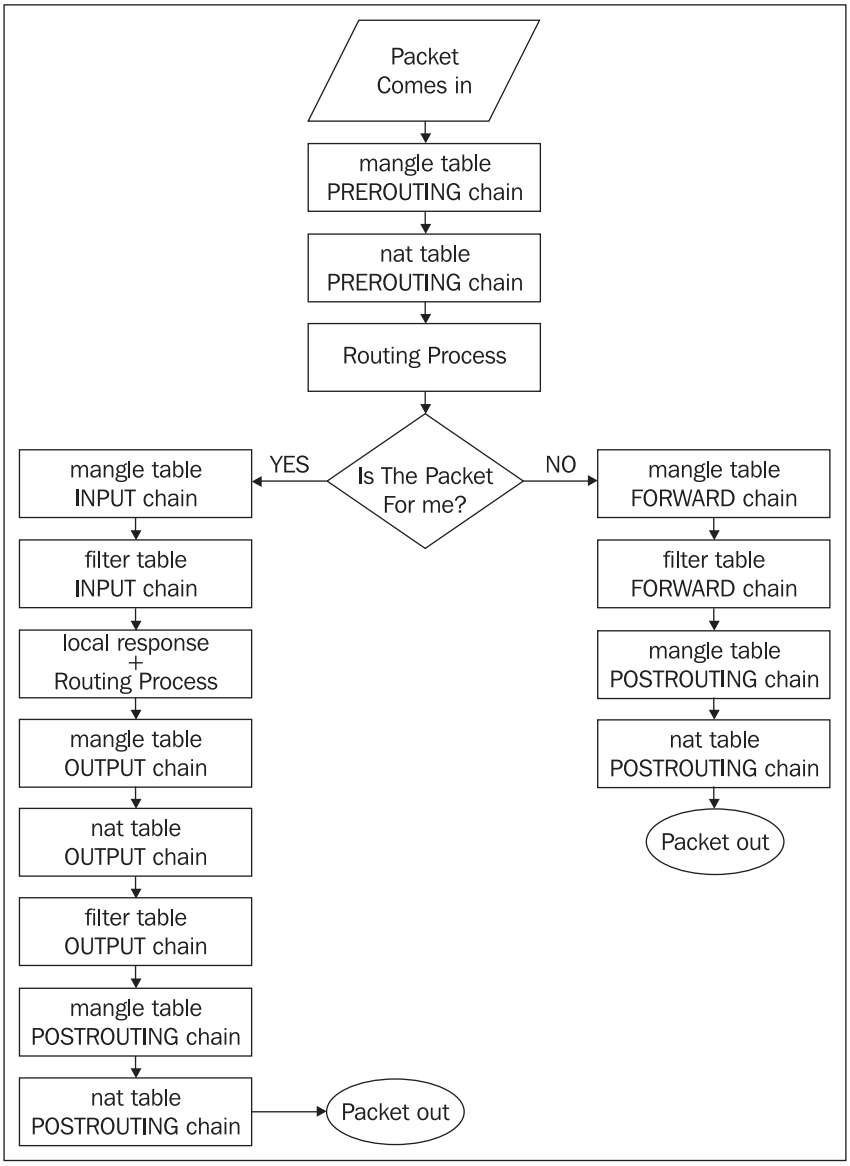
- The user instructs the kernel about what it needs to do with the IP packets that flow through the Linux box using the iptables tool.
- The Linux box then analyzes the IP headers on all packets flowing through it.
- If, when looking at the IP headers, the kernel finds matching rules, then the packet is manipulated according to the matching rule.

It might look very simple at the beginning, but actually is a lot more complicated process. netfilter has a few **tables**, each containing a default set of rules, which are called chains. The default table loaded into the kernel is the filter table, which contains three chains:

- **INPUT**: Contains rules for packets destined to the Linux machine itself.
- **FORWARD**: Contains rules for packets that the Linux machine routes to another IP address.
- **OUTPUT**: Contains rules for packets generated by the Linux machine.

By loading the NAT and mangle modules of netfilter, nat and mangle tables are automatically loaded. The nat and mangle tables, and their predefined chains, are detailed in Chapter 4.

Here's an overview of how packets travel through the tables and their chains:



This diagram shows how packets travel the tables and their chains when the NAT and mangle modules are loaded into the kernel. Immediately after a packet arrives at our Linux box, the mangle table PREROUTING chain is analyzed. At this point we can do all sorts of modifications on the IP packets supported by the mangle table (e.g. TOS byte modifications, marking packets, and so on) before the routing process takes place.

Next, the packets flow through the pre-routing chain of the nat table, where we can do DNAT, port redirection, etc.

It is only logical to be able to perform destination network address translation before the routing process occurs. As you will see in Chapter 4, where we discuss DNAT in more detail, DNAT is the process of translating one (usually public) IP address into another (usually private). This is done by modifying the destination IP address in the IP packet's header. netfilter must do that before the kernel makes a routing decision so that the kernel will look for the new destination IP address in the IP packet.

After passing through the two chains, the Linux kernel makes a routing decision. This is not the netfilter's job. By analyzing the destination IP address from the IP packet header, the Linux box knows if the packet needs to be routed elsewhere or it was destined for it.

If the Linux box is the destination for the IP packet, the packet goes through the mangle table's INPUT chain for packet mangling. Afterwards, the packet is passed to the filter table INPUT chain, where it can be accepted, rejected, or dropped. If the packet is accepted (e.g. a request to a web server running on our Linux box), the Linux box generates a response to that packet, which goes through the mangle table OUPUT chain first.

Next, the packet is passed through the nat table OUTPUT chain and the filter table OUTPUT chain. At this point, the mangle table POSTROUTING chain and the nat table POSTROUTING chain are analyzed and the packet is ready to be sent out on the corresponding interface.

The chains presented here are the predefined chains of each table (filter, nat, and mangle). However, users can set up custom chains with custom names, and pass packets to those chains from the corresponding predefined chain. For example, if we want to create some rules for SSH access into the Linux box, we can create a custom chain named SSH, and insert one rule in the INPUT chain that instructs the kernel to analyze the SSH chain for incoming packets on port 22/TCP.

The predefined chains cannot be deleted or renamed.

This packet flow diagram is one basic thing to have in mind for all people who work with netfilter/iptables on a daily basis. It's recommended to memorize it or at least keep it handy if you are just starting out.

Iptables — Operations

iptables has a syntax somewhat similar to the old ipchains (netfilter for 2.2 kernels). However, the concepts of netfilter for 2.4+ kernels are totally different from netfilters' concepts for 2.2 kernels.

The operations iptables can do with chains are:

- List the rules in a chain (`iptables -L CHAIN`).
- Change the policy of a chain (`iptables -P CHAIN ACCEPT`).
- Create a new chain (`iptables -N CHAIN`).
- Flush a chain; delete all rules (`iptables -F CHAIN`).
- Delete a chain (`iptables -D CHAIN`), only if the chain is empty.
- Zero counters in a chain (`iptables -Z CHAIN`). Every rule in every chain keeps a counter of the number of packets and bytes it matched. This command resets those counters.

For the `-L`, `-F`, `-D`, and `-Z` operations, if the chain name is not specified, the operation is applied to the entire table, which if not specified is by default the filter table.

To specify the table on which we do operations, we must use the `-t` switch like so `iptables -t filter` ...

Operations that iptables can execute on rules are:

- Append rules to a chain (`iptables -A`)
- Insert rules in a chain (`iptables -I`)
- Replace a rule from a chain (`iptables -R`)
- Delete a rule from a chain (`iptables -D`)

The most used switches are `-A` and `-D` (append and delete rules). Usually, when designing firewalls, the rules are appended to chains.

During run time, users use `-I` more than `-A` because often they need to insert temporary rules in the chain.

`iptables -A` places the rule at the end of the chain, while `iptables -I` places the rule on the top of the other rules in the chain. However, you can insert a rule anywhere in the chain by specifying the position where you want the rule to be in the chain with the `-I` switch: `iptables -I CHAIN 4` will insert a rule at the fourth position of the specified chain.

`iptables -D` can be used by specifying the position of the rule you want to delete or by specifying the entire rule.

The syntax for adding a rule to a chain is:

```
iptables -A <CHAIN_NAME> …<filtering specifications>… -j <TARGET>
```

Filtering specifications is a part of an iptables rule that is used by the kernel to identify IP packets for which the kernel does the action specified by `TARGET`.

Filtering Specifications

IP packets can be identified in a large number of ways by specifying interfaces, protocols, ports, etc., to iptables rules. The beauty of it is that we can mix any of those specifications, having a high flexibility and a wide range of selectors. I'm not planning to cover all those selectors in depth, but keep in mind that if you think about something logical about IP packets, you have every chance to identify those packets using iptables rules.

Filtering specifications for Layer2: Interfaces can be specified as selectors with `-i` and `-o` switches.

`-i` stands for "`--in-interface`", and `-o` for "`--out-interface`". `+` can be used to specify only the beginning string of the interface—for example `-i eth+` will match all interfaces beginning with the string `eth`; so we've specified all Ethernet interfaces as input interfaces for one rule.

Short version switches (e.g `-i`) and long version switches (e.g. "`--in-interface`") have absolutely the same effect. Some people prefer using short switches for command lines and long switches for scripts as they can offer better readability, but we will use only short switches in this book even in the scripts to get used to the command lines better.

The exclamation mark "!" represents a negation and can be used to specify on which interface(s) not to apply this filter (e.g. `-i ! eth1` will *not* match packets coming in on eth1).

Packets analyzed in the OUTPUT and POSTROUTING chains don't have input interfaces, and so it is not allowed to use the `-i` switch on those chains.

Also, INPUT and PREROUTING chains don't have output interfaces, and so you can't use the `-o` switch for rules in those chains.

Layer 3: Source IP address(es) can be specified using `-s`, `--src`, or `--source`, and destination IP address(es) with `-d`,`--dst`, or `--destination`. Sources or destinations can be IP addresses, subnets, or canonical names (e.g, "`-s 217.207.125.58`", "`-s www.packtpub.com`", or "`-s 217.207.125.58/32`" have the same effect). Specifying canonical names for hosts that have multiple IP addresses will result in adding the same number of rules as the number of IP addresses the DNS server resolves for that host at the time the rules are added.

Don't use canonical names on rules with high risk. For example, don't allow SSH access from `ahost.anotherisp.com`, as this will easily allow a man-in-the-middle attack.

Layer 4: Protocol can be specified using the `-p` switch, which stands for "`--protocol`". Protocols can be specified by their corresponding numbers or by their names—`tcp`, `udp`, or `icmp` (case insensitive).

For the ICMP protocol, you can specify ICMP message types using "`--icmp-type`". The list of ICMP messages can be found by using the command "`iptables -p icmp --help`".

For the UDP protocol, you can specify source or destination ports with "`--source-port`" or "`--sport`" and "`--destination-port`" and "`--dport`".

TCP, being the most complete Layer 4 protocol, has more options. You can specify, besides source or destination ports as for the UDP protocol, "`--tcp-flags`", "`--syn`" and "`--tcp-option`". TCP flags can be "`SYN ACK FIN RST URG PSH ALL NONE`". "`--syn`" is used to identify the initiating connections and is equivalent to "`--tcp-flags SYN,RST,ACK SYN`". "`--tcp-option`" followed by a number matches TCP packets with the option set to that number.

Filtering specifications can combine all of the features just mentioned; so we can have a combination of Layers 2, 3, and 4 specifications in the same rule.

Another beautiful thing about netfilter/iptables is that matching extensions can be developed separately and added later. On the netfilter site, there is a large repository of matching extensions called "patch-o-matic", at `http://www.netfilter.org/projects/patch-o-matic/index.html`.

A new and "daring" extension to iptables plans to extend its capabilities from the lower layers to the upper layer of the OSI model, Layer 7—application. The project is called l7-filter and it will be explained later in this book, in Chapter 6.

Target Specifications

For the filter table, the most used targets for firewall rules are DROP and ACCEPT. If a rule matches the filtering specifications and has a DROP target, the packet will simply be discarded. If a packet matches a rule with a DROP target, the Linux kernel will drop the packet without consulting other rules in the firewall. If the target is ACCEPT, then the packet is accepted without further consultation of other firewall rules.

An alternative to DROP is the REJECT target, which drops the packet but sends an ICMP packet to the source IP of the packet. By default, the REJECT target will send an ICMP 'port unreachable' message to the sender, but that can be overwritten using the "`--reject-with`" switch.

The target in an iptables rule can also be used to pass a packet to a user-defined chain. For example, if we create a new chain like "`iptables -N SSH`", we need to tell the kernel to look for this chain for all incoming TCP connections on port 22 like this:

```
iptables -A INPUT -p tcp --dport 22 -j SSH
```

Another useful target is LOG, which can be used to log packets matching a filtering specification in the kernel log, which can be read with dmesg or syslogd. LOG target options are:

- `--log-level level`: The level of logging can be a name or a number. The valid names are `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert`, and `emerg` with corresponding numbers from 7 to 0.
- `--log-prefix prefix`: Log prefix is followed by a string of up to 29 characters, placed at the beginning of the log message.
- `--log-tcp-sequence`: Logs TCP sequence numbers.
- `--log-tcp-options`: Logs the option field of TCP packet headers.
- `--log-ip-options`: Logs the option field of the IP packet headers.
- `--log-uid`: Logs the user ID of the process that generated the packet.

The LOG target is not a terminating target like ACCEPT, DROP, and REJECT. This means that if a packet matches a rule that has the LOG target, the kernel looks up the rules that follow to also match this packet. A limit match for rules with LOG targets would be a good idea to prevent flooding the log files.

As an example, earlier we created the SSH chain and passed packets coming in on port 22/TCP. Now, we want to accept incoming SSH connections from 192.168.0.0/27 and 10.10.15.0/24, for example, and log all other attempts, but we will limit logging to 5/s, because in the case of a SYN flood on port 22/TCP, the logs would fill quickly.

First, we will append the rules to the SSH chain to allow connections from the trusted hosts:

```
iptables -A SSH -s  192.168.0.0/27 -j ACCEPT
iptables -A SSH -s  10.10.15.0/24 -j ACCEPT
```

Next, we will add the logging rule:

```
iptables -A SSH  -m limit --limit 5/s -j LOG
```

And then DROP all other connections:

```
iptables -A SSH -j DROP
```

We need to verify the configuration, and we will use `iptables -L -n` for that. We will see in the INPUT chain:

```
root@router:~/lucix# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
SSH        tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:22
```

And we will see the SSH chain:

```
Chain SSH (1 references)
target     prot opt source               destination
ACCEPT     all  --  192.168.0.0/27       0.0.0.0/0
ACCEPT     all  --  10.10.15.0/24        0.0.0.0/0
LOG        all  --  0.0.0.0/0            0.0.0.0/0
limit: avg 5/sec burst 5 LOG flags 0 level 4
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

To test the SSH chain we will try to telnet port 22 from an unauthorized host. Using `iptables -L -n -v`, we will see that the packet matched the LOG and DROP rules:

```
Chain SSH (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     all  --  *      *       192.168.0.0/27       0.0.0.0/0
    0     0 ACCEPT     all  --  *      *       10.10.15.0/24        0.0.0.0/0
    1    48 LOG        all  --  *      *       0.0.0.0/0            0.0.0.0/0
limit: avg 5/sec burst 5 LOG flags 0 level 4
    1    48 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0
```

Now, if you look at the logs using `dmesg` command, you will see:

```
IN=eth0 OUT= MAC=00:d0:b7:a7:6f:74:00:04:23:cf:14:e6:08:00
SRC=192.168.168.168 DST=192.168.0.1 LEN=48 TOS=0x00 PREC=0x00
TTL=109 ID=54250 DF PROTO=TCP SPT=27276 DPT=22 WINDOW=16384
RES=0x00 SYN URGP=0
```

which tells us that 192.168.168.168 tried to connect on port 22 TCP.

For the nat and mangle tables, we will discuss the targets in the following chapter.

The targets presented here are the most commonly used targets of iptables. There are a lot of add-ons published on patch-o-matic, which can provide new targets for iptables.

A Basic Firewall Script—Linux as a Workstation

So far, we've learned mostly about the usage of iptables filtering options. I will now build up a small firewall script that I think should be default when installing any Linux distribution.

By default, all Linux distributions have the default policy ACCEPT on all filter chains. Also, on a default installation, most Linux distributions leave a lot of services running. If you install an old Linux distribution and decide to go for lunch after you have just booted up without any firewall and with a public IP address, good chances are that by the time you've eaten your soup, a rootkit is already installed on your computer.

Let's take a look at the following simple script:

```
#!/bin/bash

#assign variable $IPT with the iptables command
IPT=/sbin/iptables

#set policies on each chain
$IPT -P INPUT DROP

$IPT -P FORWARD DROP

$IPT -P OUTPUT ACCEPT #default, but set it anyway
#flush all rules in the filter table
$IPT -F

#allow traffic on the loopback interface
```

```
$IPT -A INPUT -i lo -j ACCEPT

#allow icmp traffic
$IPT -A INPUT -p icmp -j ACCEPT

#allow incoming DNS traffic
$IPT -A INPUT -p udp --sport 53 -j ACCEPT

#allow established TCP connections
$IPT -A INPUT -p tcp ! --syn -j ACCEPT
```

So, what we did here was to set the INPUT and FORWARD chains policy to DROP. The OUTPUT chain policy is set to ACCEPT, which is the default policy for this chain.

We will not append any rules in the FORWARD chain because this is a personal computer and not a router, and so the forwarding will be off. We will also not append any rules in the OUTPUT chain—anything we originate is OK.

Next, we flush all existing rules out from the filter table. At this point, nothing really works. Some applications use TCP/IP connections on the loopback interface; so it's safe to allow packets that come in on the interface "lo".

We learned about ICMP attacks in Chapter 2. However, it is my opinion that ICMP should be allowed. Filtering ICMP will not allow you to test your internet connection using `ping`, `traceroute`, `mtr`, etc., and also path MTU discovery will not work, which is a very important protocol in many cases.

DNS responses use the UDP protocol and source port 53. Keep in mind that the line:

```
$IPT -A INPUT -p udp --sport 53 -j ACCEPT
```

is a potential security breach. We left it like this because we earlier stated that this is what we think the default firewall should look like. However, if you're not running a DNS server (which is not recommended for a personal computer), accept incoming UDP connections with source port 53 only from your provider's DNS servers (the ones you have in `/etc/resolv.conf`). For example, if the provider's DNS servers are 1.1.1.1 and 1.1.2.1, replace the earlier line with:

```
$IPT -A INPUT -s 1.1.1.1 -p udp --sport 53 -j ACCEPT
$IPT -A INPUT -s 1.1.2.1 -p udp --sport 53 -j ACCEPT
```

This way, you will be safer.

The last thing we need for our internet connection to work is to allow incoming TCP traffic for already established TCP connections. Better phrased, deny any incoming TCP traffic that doesn't belong to a TCP connection that this computer initiated (deny TCP SYN packets).

This rough introduction on netfilter/iptables covers only some basic parts of the subject. The intention of this book is not to teach the reader the complete syntax of iptables. For that, there is a manpage at `http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html`.

iproute2 and Traffic Control

iproute2 is a software package that provides various tools for advanced routing, tunnels, and traffic control.

iproute2 was originally designed by Alexey Kuznetsov, and is well known for implementing QoS in Linux kernels and is now maintained by Stephen Hemminger. The primary site for iproute2 is `http://linux-net.osdl.org/index.php/Iproute2` and its main documentation site is `http://www.lartc.org`.

The most important tools that iproute2 provides are `ip` and `tc`.

Network Configuration: "ip" Tool

The `ip` tool provides most of the networking configuration a Linux box needs. You can configure interfaces, ARP, policy routing, tunnels, etc.

Now, with IPv4 and IPv6, `ip` can do pretty much anything (including a lot that we don't need in our particular situations). The syntax of `ip` is not difficult, and there is a lot of documentation on this subject. However, the most important thing is knowing what we need and when we need it.

First of all, `ip` is the main tool we need for dynamic routing protocols (BGP, OSPF, and RIP) on Linux provided by Zebra, which will be discussed later in this book.

Let's have a look at the `ip` command help to see what `ip` knows:

```
root@router:~# ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { link | addr | route | rule | neigh | tunnel |
                   maddr | mroute | monitor | xfrm }
```

```
        OPTIONS := { -V[ersion] | -s[tatistics] | -r[esolve] |
                     -f[amily] { inet | inet6 | ipx | dnet | link } |
-o[neline] }
root@router:~#
```

The `ip link` command shows the network device's configurations that can be changed with `ip link set`. This command is used to modify the device's proprieties and not the IP address.

The IP addresses can be configured using the `ip addr` command. This command can be used to add a primary or secondary (alias) IP address to a network device (`ip addr add`), to display the IP addresses for each network device (`ip addr show`), or to delete IP addresses from interfaces (`ip addr del`). IP addresses can also be flushed using different criteria, e.g. `ip addr flush dynamic` will flush all routes added to the kernel by a dynamic routing protocol.

Neighbor/Arp table management is done using `ip neighbor`, which has a few commands expressively named `add`, `change`, `replace`, `delete`, and `flush`.

`ip tunnel` is used to manage tunneled connections. Tunnels can be `gre`, `ipip`, and `sit`. We will include an example later in the book on how to build IP tunnels.

The `ip` tool offers a way for monitoring routes, addresses, and the states of devices in real-time. This can be accomplished using `ip monitor`, `rtmon`, and `rtacct` commands included in the iproute2 package.

One very important and probably the most used object of the `ip` tool is `ip route`, which can do any operations on the kernel routing table. It has commands to add, change, replace, delete, show, flush, and get routes.

One of the things iproute2 introduced to Linux that ensured its popularity was policy routing. This can be done using `ip rule` and `ip route` in a few simple steps.

Traffic Control: tc

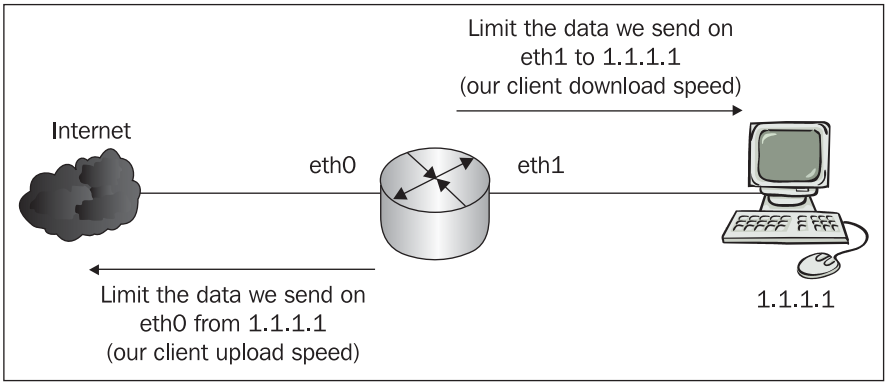
The `tc` command allows administrators to build different QoS policies in their networks using Linux instead of very expensive dedicated QoS machines. Using Linux, you can implement QoS in all the ways any dedicated QoS machine can and even more. Also, one can make a bridge using a good PC running Linux that can be transformed into a very powerful and very cheap dedicated QoS machine.

For that, QoS support must be configured in the Linux kernel (`CONFIG_NET_QOS="Y"` and `CONFIG_NET_SCHED="Y"`).

Queuing Packets

First of all, queuing is used to determine the way data is sent; so with queuing, we can control how much data is sent, and with what priority we send data that matches some criteria.

Please keep in mind that there is no way to queue incoming data. When we talk about limiting upload and download speeds for some IP address, for example, we talk about limiting the data our Linux router sends to that IP address on the interface that IP address is connected to (download) and the data our Linux router sends over the Internet from that IP address (upload), as in the following figure:



This is quite satisfying because TCP has flow control, which actually negotiates the speed of the packet flow between two communicating hosts depending on the capabilities of each host. UDP doesn't have flow control, but most of the applications that use UDP as transport protocol implement flow control within themselves.

Well, things look pretty good, but this is how things work in the "perfect world", where there aren't people with bad intentions (or stupid people without bad intentions) that generate flood attacks because we can't limit the incoming data.

So, what's the problem? Well, put 99 computers near the 1.1.1.1 computer in the earlier figure! Let's say there are 100 users on a FastEthernet connection (with more switches, as the router has one Ethernet cable in one switch). We can limit each computer to 1Mbps upload / 1Mbps download; so we're using 100 Mbps when everyone is on the top of their limits. Now, if 1.1.1.1 wants to disrupt service to the other users, it's very simple. Because there is no way of limiting incoming traffic, if 1.1.1.1 floods one or many random hosts on the Internet with a 100Mbps data stream, the router limits the outgoing data from 1.1.1.1 to 1Mbps, but it still receives 100Mbps on its eth1 interface. This results in denial of service, and there isn't really much to do about it. If the switches are unmanaged, the only thing you can do about it is to plug out the cable from the port in which 1.1.1.1 is connected.

Now, to get back to the subject, queuing disciplines are of two kinds: classless and classful.

Classless Queuing Disciplines (Classless qdiscs)

Classless qdiscs are the simplest ones because they only accept, drop, delay or reschedule data. They can be attached to one interface and can only shape the entire interface.

There are several qdisc implementations on Linux, most of them included in the Linux kernel.

- **FIFO** (**pfifo and bfifo**): The simplest qdisc, which functions by the First In, First Out rule. FIFO algorithms have a queue size limit (buffer size), which can be defined in packets for pfifo or in bytes for bfifo.
- **pfifo_fast**: The default qdisc on all Linux interfaces. It's important to know how pfifo_fast works; so we'll explain it soon.
- **Token Bucket Filter** (**tbf**): A simple qdisc that is perfect for slowing down an interface to a specified rate. It can allow short bursts over the specified rate and is very processor friendly.
- **Stochastic Fair Queuing** (**SFQ**): One of the most widely used qdiscs. SFQ tries to fairly distribute the transmitting data among a number of flows.
- **Enhanced Stochastic Fair Queuing** (**ESFQ**): Not included in the Linux kernel, it works in the same manner as SFQ with the exception that the user can control more of the algorithm's parameters such as depth (flows) limit, hash table size options (hardcoded in original SFQ) and hash types.
- **Random Early Detection and Generic Random Early Detection** (RED and GRED): qdiscs suitable for backbone data queuing, with data rates over 100 Mbps.

There are more qdiscs than the ones I have stated here. However, from my experience, SFQ and ESFQ do a great job, and are the qdiscs that I have got the best results with.

As I said earlier, the default qdisc on Linux for all interfaces is pfifo_fast. Normally, one would think that this is just like pfifo, meaning there is a buffer and packets pass through the buffer using the First In First Out rule. Actually, it's not quite true. pfifo_fast has 3 bands—0, 1, and 2—in which packets are placed according to their TOS byte. Packets are sent out from those bands as follows:

- Packets in the 0 band have the highest priority
- Packets in the 1 band are sent out only if there aren't any packets in the 0 band

- Packets in the 2 band have the lowest priority and are sent out only if there aren't any packets in the 0 and 1 bands.

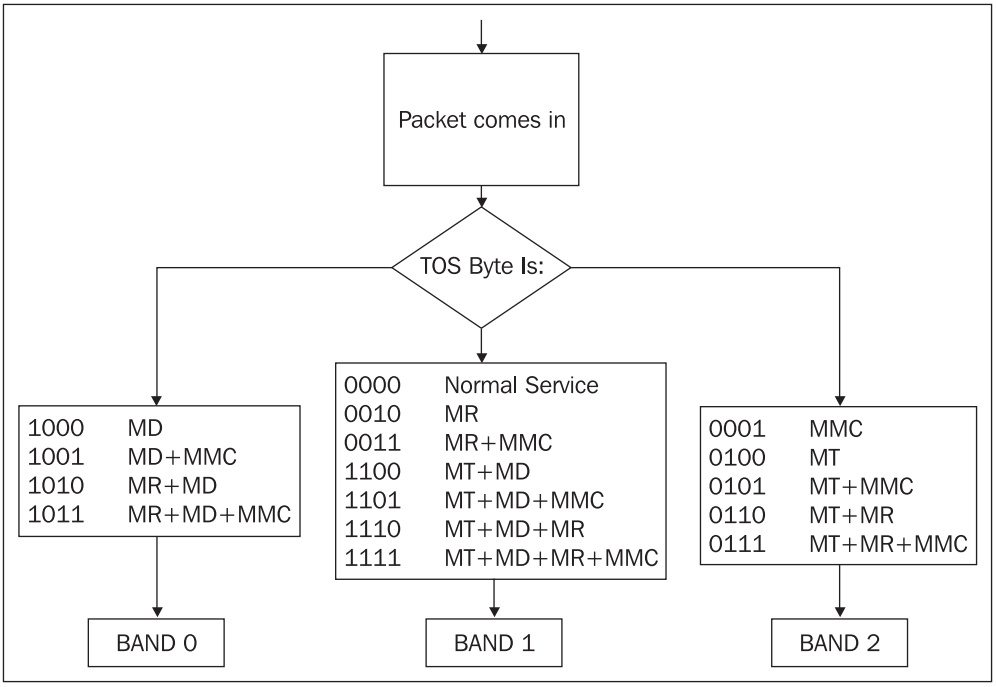
It's important to know this because this can be a way to optimize how packets travel through the network interfaces of our Linux routers. The TOS byte looks like this:

0	1	2	3	4	5	6	7
PRECEDENCE			Type of Service — TOS				MBZ

The TOS bits are defined as follows:

- 0000 Normal Service
- 0001 Minimize Monetary Cost (MMC)
- 0010 Maximize Reliability (MR)
- 0100 Maximize throughput (MT)
- 1000 Minimize Delay (MD)

Based on the TOS byte, the packets are placed in one of the three bands as follows:



This means that, by default, Linux is smart enough to prioritize traffic according to the TOS bytes. Usually, applications like Telnet, FTP, SMTP modify the TOS byte to work in an optimal way. We will see later in this book how to optimize the traffic ourselves.

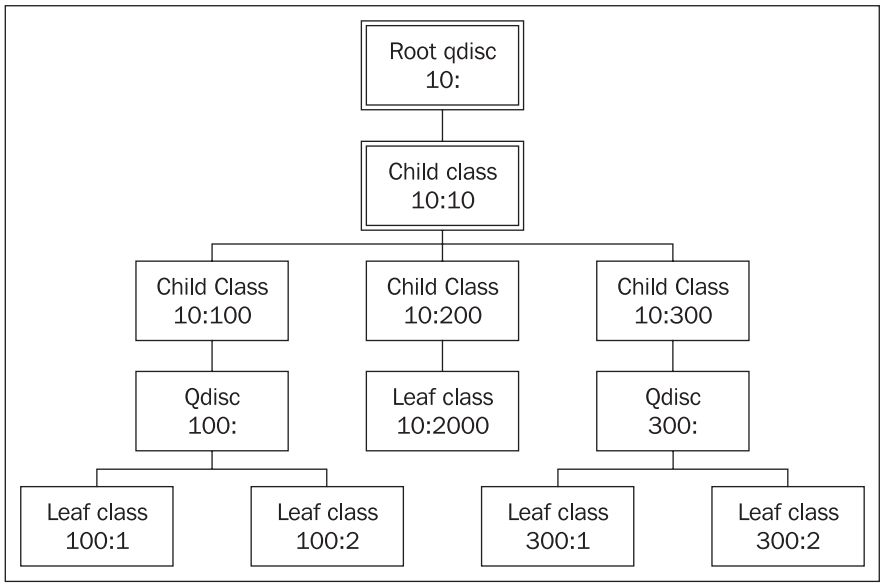
Classful Queuing Disciplines

These qdiscs are used for shaping different types of data. The commonly used classful qdiscs are CBQ (Class Based Queuing) and HTB (Hierarchical Token Bucket).

First of all, we need to learn how classful queuing disciplines work. The whole process is not difficult; so I'll try to explain it as simply as possible.

Everything is based on a hierarchy. First, every interface has one root qdisc that talks to the kernel. Second, there is a child class attached to the root qdisc. The child class further has child classes that have qdiscs attached to schedule the data and leaf classes, which are child classes of the child classes.

All confused? Have a look at the following image, which will explain away the confusion:



So, basically CBQ or HTB qdiscs allow us to create child CBQ or HTB classes, which we can set up to shape some kind of data. For each child class, we can attach a qdisc for scheduling packets within that child class. Next, we can create leaf classes, which are child classes of the qdiscs we attached to the child classes, or we can create leaf classes as child classes' child classes attached to the root qdisc.

tc qdisc, tc class, and tc filter

To build the tree configuration in the earlier figure, we need to use the `tc` command:

- `tc qdisc` manipulates queuing disciplines.
- `tc class` manipulates classes.
- `tc filter` manipulates filters used to identify data.

Both CBQ and HTB have a few parameters that can be adjusted to optimize their performance. Throughout this book we will use different values to suit the applications we are building. There is a lot of tuning to be done with these parameters, and I'm not going to explain all of them as there are some that you will probably never need.

CBQ qdiscs and classes have the following parameters:

```
root@router:~# tc qdisc add cbq help
Usage: ... cbq bandwidth BPS avpkt BYTES [ mpu BYTES ]
               [ cell BYTES ] [ ewma LOG ]
root@router:~# tc class add cbq help
Usage: ... cbq bandwidth BPS rate BPS maxburst PKTS [ avpkt BYTES ]
               [ minburst PKTS ] [ bounded ] [ isolated ]
               [ allot BYTES ] [ mpu BYTES ] [ weight RATE ]
               [ prio NUMBER ] [ cell BYTES ] [ ewma LOG ]
               [ estimator INTERVAL TIME_CONSTANT ]
               [ split CLASSID ] [ defmap MASK/CHANGE ]
```

and HTB qdiscs and classes' parameters are:

```
root@router:~# tc class add htb help
Usage: ... qdisc add ... htb [default N] [r2q N]
 default  minor id of class to which unclassified packets are sent {0}
 r2q      DRR quantums are computed as rate in Bps/r2q {10}
 debug    string of 16 numbers each 0-3 {0}

... class add ... htb rate R1 [burst B1] [mpu B] [overhead O]
                      [prio P] [slot S] [pslot PS]
                      [ceil R2] [cburst B2] [mtu MTU] [quantum Q]
 rate     rate allocated to this class (class can still borrow)
 burst    max bytes burst which can be accumulated during idle
period {computed}
 mpu      minimum packet size used in rate computations
 overhead per-packet size overhead used in rate computations
 ceil     definite upper class rate (no borrows) {rate}
 cburst   burst but for ceil {computed}
 mtu      max packet size we create rate map for {1600}
```

```
 prio     priority of leaf; lower are served first {0}
 quantum  how much bytes to serve from leaf at once {use r2q}

TC HTB version 3.3
```

I will try to explain a few of these parameters while using them in the actual example that follows.

Filters are used to identify the data we need to shape. We can identify the data based on the way the firewall marked it using the `fw` classifier, based on fields of the IP header using the `u32` classifier, based on the kernel's routing decision using the `route` classifier, or based on RSVP using `rsvp` or `rsvp6` classifiers.

The `tc filter` command has the following parameters:

```
root@router:~# tc filter help
Usage: tc filter [ add | del | change | get ] dev STRING
       [ pref PRIO ] [ protocol PROTO ]
       [ estimator INTERVAL TIME_CONSTANT ]
       [ root | classid CLASSID ] [ handle FILTERID ]
       [ [ FILTER_TYPE ] [ help | OPTIONS ] ]

       tc filter show [ dev STRING ] [ root | parent CLASSID ]
Where:
FILTER_TYPE := { rsvp | u32 | fw | route | etc. }
FILTERID := ... format depends on classifier, see there
OPTIONS := ... try tc filter add <desired FILTER_KIND> help
```

The most used classifier is `u32`, because most people desire to identify data by IP addresses, source or destination ports, etc. However, we will use the fw classifier along with u32 throughout the book. The u32 parameters are:

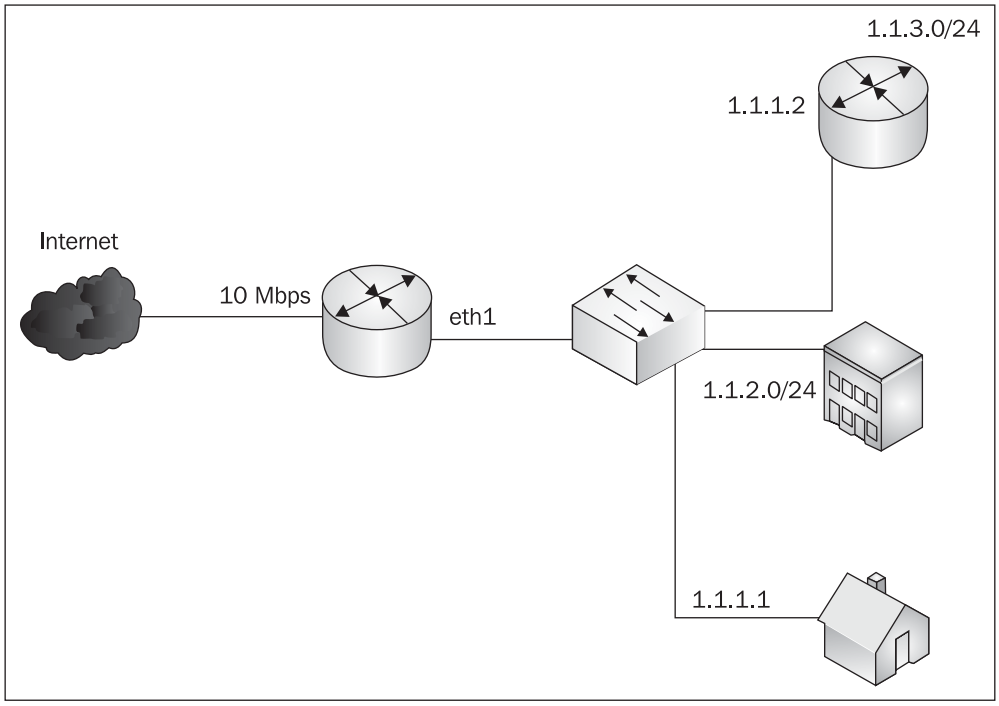
```
root@router:~# tc filter add u32 help
Usage: ... u32 [ match SELECTOR ... ] [ link HTID ] [ classid CLASSID
]
               [ police POLICE_SPEC ] [ offset OFFSET_SPEC ]
               [ ht HTID ] [ hashkey HASHKEY_SPEC ]
               [ sample SAMPLE ]
or         u32 divisor DIVISOR

Where: SELECTOR := SAMPLE SAMPLE ...
    SAMPLE := { ip | ip6 | udp | tcp | icmp | u{32|16|8} } SAMPLE_ARGS
    FILTERID := X:Y:Z
```

And for the fw classifier:

```
root@router:~# tc filter add fw help
Usage: ... fw [ classid CLASSID ] [ police POLICE_SPEC ]
       POLICE_SPEC := ... look at TBF
       CLASSID := X:Y
```

A Real Example

In the following example we will try to divide a 10Mbps bandwidth between three entities: a home-user, an office, and another ISP, as shown in the following figure:



Let's assume we want to give the home user 1Mbps of our bandwidth, the office 4Mbps, and the ISP 5Mbps.

First, let's see how this looks using CBQ. First, we need to add the root qdisc to the eth1 interface on which the clients are connected:

```
tc qdisc add dev eth1 root handle 10: cbq bandwidth 100Mbit avpkt 1000
```

So, the command used is `tc qdisc add` with the `dev` parameter set to `eth1` to define the interface we will attach the qdisc to. The `root` parameter specifies that this is the root qdisc. We will assign `handle 10` for the root qdisc. After specifying

the handle, we specified `cbq` as the type of the qdisc, followed by the parameters for `cbq`. `bandwidth` is set to `100Mbit`, which is the physical bandwidth of the device, and `avpkt`, which specifies the average packet size is set to `1000`.

Next, we need to create a child class that will be the parent of all classes. This class will have the bandwidth parameter equal to that of the root qdisc, equal to the physical bandwidth of the interface:

```
tc class add dev eth1 parent 10:0 classid 10:10 cbq bandwidth 100Mbit
rate \
  100Mbit allot 1514 weight 10Mbit prio 5 maxburst 20 avpkt 1000
bounded
```

For the child classes, we need to specify the parent class, which in this case is `10:0`—the root class. `classid` specifies the ID of the class, and `bandwidth` is the physical bandwidth of the interface (100Mbit). The speed limit is specified with the `rate` parameter, followed by the rate in bits (in this case, `100Mbit`). The `allot` parameter is the base unit for how much data the class can send in one round. `weight` is a parameter used by CBQ with `allot` to calculate how much data is sent in one round. Actually, from our experience and tests, `weight` pretty much specifies the rate in bytes for the class.

We will be using in this book parameters that gave the best results in our tests. Except `bandwidth`, `rate`, and `weight`, we don't recommend learning about all the other parameters. However, there is a more detailed explanation at: `http://www.lartc.org/howto/lartc.qdisc.classful.html#AEN939`.

For each client, we will create leaf classes, qdiscs, and filters. Let's start with the home user:

```
tc class add dev eth1 parent 10:10 classid 10:100 cbq bandwidth
100Mbit rate \
  1Mbit allot 1514 weight 128Kbit prio 5 maxburst 20 avpkt 1000
bounded

tc qdisc add dev eth1 parent 10:100 sfq quantum 1514b perturb 15

tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.1.1 flowid 10:100
```

So we created the 10:100 class with a rate of 1Mbit and 128Kbit weight. Next, we attached an `sfq` qdisc and a `u32` filter to match all traffic with the destination IP address 1.1.1.1. The `bounded` argument of the `tc class add cbq` command means

that the class isn't allowed to borrow bytes from other classes, meaning that there is no way that data for this class will go over 1Mbps.

A lot of documentation explains that `weight` should be `rate/10`. In our case, weight would be 100Kbit and the user wouldn't get data with speed above 100KB/s which is not 1Mbps. We've been always using `weight` as `rate/8` because this seems more fair to me.

Now, the other classes, qdiscs, and filters look like this:

```
#the office
tc class add dev eth1 parent 10:10 classid 10:200 cbq bandwidth
100Mbit rate \
  4Mbit allot 1514 weight 512Kbit prio 5 maxburst 20 avpkt 1000
bounded

tc qdisc add dev eth1 parent 10:200 sfq quantum 1514b perturb 15

tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.2.0/24 flowid 10:200

#the ISP
tc class add dev eth1 parent 10:10 classid 10:300 cbq bandwidth
100Mbit rate \
  5Mbit allot 1514 weight 640Kbit prio 5 maxburst 20 avpkt 1000
bounded

tc qdisc add dev eth1 parent 10:300 sfq quantum 1514b perturb 15

tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.1.2 flowid 10:300
tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.3.0/24 flowid 10:300
```

As you can see in the ISP case, we can add as many filters as we want to a class.

To verify the configuration, we can use `tc class show dev eth1` and see the classes:

```
root@router:~# tc class show dev eth1
class cbq 10: root rate 100000Kbit (bounded,isolated) prio no-transmit
class cbq 10:100 parent 10:10 leaf 806e: rate 1000Kbit (bounded) prio 5
class cbq 10:10 parent 10: rate 100000Kbit (bounded) prio 5
class cbq 10:200 parent 10:10 leaf 806f: rate 4000Kbit (bounded) prio 5
class cbq 10:300 parent 10:10 leaf 8070: rate 5000Kbit (bounded) prio 5
```

Now, to see that a class is actually shaping packets, we send three `ping` packets to 1.1.1.1, and check to see if the CBQ class matched those packets using `tc -s class show dev eth1`:

```
root@router:~# tc -s class show dev eth1 | fgrep -A 2 10:100
class cbq 10:100 parent 10:10 leaf 806e: rate 1000Kbit (bounded) prio 5
 Sent 294 bytes 3 pkts (dropped 0, overlimits 0)
  borrowed 0 overactions 0 avgidle 184151 undertime 0
```

Now everything looks OK; so let's move on to HTB. Before we do that, we need to delete the CBQ root qdisc using:

```
root@router:~# tc qdisc del root dev eth1
```

Using HTB looks a bit simpler than CBQ. First, the root qdisc looks like this:

```
tc qdisc add dev eth1 root handle 10: htb
```

Next, we will create the child class:

```
tc class add dev eth1 parent 10:0 classid 10:10 htb rate 100Mbit
```

Now, the qdiscs and filters within the client classes are the same as in the CBQ example. The only thing that differs is how the classes are built. Let's see the home-user class, qdisc, and filter:

```
tc class add dev eth1 parent 10:10 classid 10:100 htb rate 1Mbit

tc qdisc add dev eth1 parent 10:100 sfq quantum 1514b perturb 15

tc filter add dev eth1 protocol ip parent 10:0 prio 5 u32 match ip dst
1.1.1.1 flowid 10:100
```

So much simple, isn't it? Let's create the other two entities' classes, qdiscs, and filters:

```
#the office
tc class add dev eth1 parent 10:10 classid 10:200 htb rate 4Mbit

tc qdisc add dev eth1 parent 10:200 sfq quantum 1514b perturb 15

tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.2.0/24 flowid 10:200


#the ISP
tc class add dev eth1 parent 10:10 classid 10:300 htb rate 5Mbit

tc qdisc add dev eth1 parent 10:300 sfq quantum 1514b perturb 15

tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
```

```
1.1.1.2 flowid 10:300
tc filter add dev eth1 parent 10:0 protocol ip prio 5 u32 match ip dst
1.1.3.0/24 flowid 10:300
```

Now it's time to verify the configuration using `tc class show dev eth1`:

```
root@router:~# tc class show dev eth1
class htb 10:10 root rate 100000Kbit ceil 100000Kbit burst 126575b
cburst 126575b
class htb 10:100 parent 10:10 leaf 8072: prio 0 rate 1000Kbit ceil
1000Kbit burst 2849b cburst 2849b
class htb 10:200 parent 10:10 leaf 8073: prio 0 rate 4000Kbit ceil
4000Kbit burst 6599b cburst 6599b
class htb 10:300 parent 10:10 leaf 8074: prio 0 rate 5000Kbit ceil
5000Kbit burst 7849b cburst 7849b
```

and after sending three `ping` packets to 1.1.1.1, we should see them on the 10:100 class:

```
root@router:~# tc -s class show dev eth1 | fgrep -A 4 10:100
class htb 10:100 parent 10:10 leaf 8072: prio 0 rate 1000Kbit ceil
1000Kbit burst 2849b cburst 2849b
 Sent 294 bytes 3 pkts (dropped 0, overlimits 0)
 rate 24bit
 lended: 3 borrowed: 0 giants: 0
 tokens: 18048 ctokens: 18048
```

There is no catch in all of this—HTB looks simpler and it really is. CBQ has more parameters that can be adjusted by the user, while HTB does much of the adjustments internally.

Summary

This chapter introduced netfilter/iptables and iproute2. A very important thing for anyone building firewalls is to know how and where packets are analyzed. For that, we introduced a diagram of how packets traverse the chains in the filter, nat, and mangle tables for netfilter.

For beginners, a first look the iptables syntax might seem a bit difficult. An iptables rule contains the table on which we make an operation (filter table being default), a command (`append`, `insert`, `delete`, `list`), some filtering specifications to match the packets we want, and a target (DROP, ACCEPT, REJECT, LOG) that specifies what we want to do with the packet.

The iproute2 package introduces two complex tools. One is `ip`, which can be used to set up Layer 3 communication like IP addresses and routing. `tc` stands for *traffic control*, and it is used to implement QoS.

Before digging into `tc` commands, we learned a bit of theory on classless and classful queuing disciplines. The best and most popular classful qdiscs are CBQ and HTB, which we will use throughout this book.

We saw that HTB is simpler to use than CBQ because the command lines for CBQ must contain a lot of parameters. On the other hand, CBQ can be tuned for more advanced configurations, but the needs for these tunings are very rare.

We made a lot of tests with CBQ, and we will use in this book the parameters that produced the best results for us.

Where to buy this book

You can buy Designing and Implementing Linux Firewalls and QoS using netfilter, iproute2, NAT and l7-filter from the Packt Publishing website:
`http://www.packtpub.com/linux-firewalls/book`

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com