Linux 2.4 Advanced Routing HOWTO

Table of Contents

Table of Contents

Table of Contents

Linux 2.4 Advanced Routing HOWTO

Netherlabs BV (bert hubert <bert.hubert@netherlabs.nl>)
Gregory Maxwell <greg@linuxpower.cx>
Remco van Mook <remco@virtu.nl>
Martijn van Oosterhout <kleptog@cupid.suninternet.com>
Paul B Schroeder <paulsch@us.ibm.com>
howto@ds9a.nl

v0.1.0 $Date: 2000/10/08 20:57:49 $

A very hands−on approach to iproute2, traffic shaping and a bit of netfilter

---

1.Dedication

2.Introduction

3.Introduction to iproute2

4.Rules − routing policy database

5.GRE and other tunnels

6.IPsec: secure IP over the internet

7.Multicast routing

8.Using Class Based Queueing for bandwidth management

9.More queueing disciplines

10.Netfilter & iproute − marking packets

11.More classifiers

12.Kernel network parameters

13.Backbone applications of traffic control

14.Cookbook

15.Advanced Linux Routing

16.Dynamic routing − OSPF and BGP

17.Further reading

18.Acknowledgements

1.Dedication

This document is dedicated to lots of people, and is my attempt to do something back. To list but a few:

- Rusty Russell
- Alexey N. Kuznetsov
- The good folks from Google
- The staff of Casema Internet

2.Introduction

Welcome, gentle reader.

This document hopes to enlighten you on how to do more with Linux 2.2/2.4 routing. Unbeknownst to most users, you already run tools which allow you to do spectacular things. Commands like 'route' and 'ifconfig' are actually very thin wrappers for the very powerful iproute2 infrastructure

I hope that this HOWTO will become as readable as the ones by Rusty Russell of (amongst other things) netfilter fame.

You can always reach us by writing the HOWTO team.

2.1 Disclaimer & License

1. Send your derivative work (in the most suitable format such as sgml) to the LDP (Linux Documentation Project) or the like for posting on the Internet. If not the LDP, then let the LDP know where it is available.
2. License the derivative work with this same license or use GPL. Include a copyright notice and at least a pointer to the license used.
3. Give due credit to previous authors and major contributors.

If you're considering making a derived work other than a translation, it's requested that you discuss your plans with the current maintainer.

It is also requested that if you publish this HOWTO in hardcopy that you send the authors some samples for 'review purposes' :−)

2.2 Prior knowledge

As the title implies, this is the 'Advanced' HOWTO. While by no means rocket science, some prior knowledge is assumed. This document is meant as a sequel to the Linux 2.4 Networking HOWTO by the same authors. You should probably read that first.

Here are some orther references which might help learn you more:

Rusty Russell's networking−concepts−HOWTO

Very nice introduction, explaining what a network is, and how it is connected to other networks

Linux Networking−HOWTO (Previously the Net−3 HOWTO)

Great stuff, although very verbose. It learns you a lot of stuff that's already configured if you are able to connect to the internet. Should be located in /usr/doc/HOWTO/NET3−4−HOWTO.txt but can be also be found online

2.3 What Linux can do for you

A small list of things that are possible:

- Throttle bandwidth for certain computers
- Throttle bandwidth to certain computers
- Help you to fairly share your bandwidth
- Protect your network from DoS attacks
- Protect the internet from your customers
- Multiplex several servers as one, for load balancing or enhanced availability
- Restrict access to your computers
- Limit access of your users to other hosts
- Do routing based on user id (yes!), MAC address, source IP address, port, type of service, time of day

or content

Currently not many people are using these advanced features. This has several reasons. While the provided documentation is verbose, it is not very hands on. Traffic control is almost undocumented.

2.4 Housekeeping notes

There are several things which should be noted about this document. While I wrote most of it, I really don't want it to stay that way. I am a strong believer in Open Source, so I encourage you to send feedback, updates, patches etcetera. Do not hesitate to inform me of typos or plain old errors. If my English sounds somewhat wooden, please realise that I'm not a native speaker. Feel free to send suggestions.

If you feel to you are better qualified to maintain a section, or think that you can author and maintain new sections, you are welcome to do so. The SGML of this HOWTO is available via CVS, I very much envision more people working on it.

In aid of this, you will find lots of FIXME notices. Patches are always welcome! Wherever you find a FIXME, you should know that you are treading unknown territory. This is not to say that there are no errors elsewhere, but be extra careful. If you have validated something, please let us know so we can remove the FIXME notice.

About this HOWTO, I will take some liberties along the road. For example, I postulate a 10Mbit internet connection, while I know full well that those are not very common.

2.5 Access, CVS & submitting updates

The canonical location for the HOWTO is here.

We now have anonymous CVS access available for the world at large. This is good in several ways. You can easily upgrade to newer versions of this HOWTO and submitting patches is no work at all.

Furthermore, it allows the authors to work on the source independently, which is good too.

```
$ export CVSROOT=:pserver:anon@outpost.ds9a.nl:/var/cvsroot
$ cvs login
CVS password: [enter 'cvs' (without 's)]
$ cvs co 2.4routing
cvs server: Updating 2.4routing
U 2.4routing/2.4routing.sgml
```

If you spot an error, or want to add something, just fix it locally, and run cvs diff −u, and send the result off to us.

A Makefile is supplied which should help you create postscript, dvi, pdf, html and plain text. You may need to install sgml−tools, ghostscript and tetex to get all formats.

2.6 Mailing list

The authors receive an increasing amount of mail about this HOWTO. Because of the clear interest of the community, it has been decided to start a mailinglist where people can talk to each other about Advanced Routing and Traffic Control. You can subscribe to the list here.

It should be pointed out that the authors are very hesitant of answering questions asked not on the list. We would like the archive of the list to become some kind of knowledge base. If you have a question, please search the archive, and then post to the mailinglist.

2.7 Layout of this document

We will be doing interesting stuff almost immediately, which also means that there will initially be parts that are explained incompletely or are not perfect. Please gloss over these parts and assume that all will become clear.

Routing and filtering are two distinct things. Filtering is documented very well by Rusty's HOWTOs, available here:

- Rusty's Remarkably Unreliable Guides

We will be focusing mostly on what is possible by combining netfilter and iproute2.

3.Introduction to iproute2

3.1 Why iproute2?

Most Linux distributions, and most UNIX's, currently use the venerable 'arp', 'ifconfig' and 'route' commands. While these tools work, they show some unexpected behaviour under Linux 2.2 and up. For example, GRE tunnels are an integral part of routing these days, but require completely different tools.

With iproute2, tunnels are an integral part of the tool set.

The 2.2 and above Linux kernels include a completely redesigned network subsystem. This new networking code brings Linux performance and a feature set with little competition in the general OS arena. In fact, the new routing, filtering, and classifying code is more featureful then that provided by many dedicated routers and firewalls and traffic shaping products.

As new networking concepts have been invented, people have found ways to plaster them on top of the existing framework in existing OSes. This constant layering of cruft has lead to networking code that is filled with strange behaviour, much like most human languages. In the past, Linux emulated SunOS's handling of many of these things, which was not ideal.

This new framework makes it possible to clearly express features previously beyond Linux's reach.

3.2 iproute2 tour

Linux has a sophisticated system for bandwidth provisioning called Traffic Control. This system supports various method for classifying, prioritizing, sharing, and limiting both inbound and outbound traffic.

We'll start off with a tiny tour of iproute2 possibilities.

3.3 Prerequisites

You should make sure that you have the userland tools installed. This package is called 'iproute' on both RedHat and Debian, and may otherwise be found at `ftp://ftp.inr.ac.ru/ip-routing/iproute2-2.2.4-now-ss??????.tar.gz"`. Some parts of iproute require you to have certain kernel options enabled.

You can also try here for the latest version.

3.4 Exploring your current configuration

This may come as a surprise, but iproute2 is already configured! The current commands `ifconfig` and `route` are already using the advanced syscalls, but mostly with very default (ie. boring) settings.

The `ip` tool is central, and we'll ask it do display our interfaces for us.

`ip` shows us our links

```
[ahu@home ahu]$ ip link list
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast qlen 10
    link/ppp
```

Your mileage may vary, but this is what it shows on my NAT router at home. I'll only explain part of the output as not everything is directly relevant.

We first see the loopback interface. While your computer may function somewhat without one, I'd advise against it. The MTU size (Maximum Transfer Unit) is 3924 octets, and it is not supposed to queue. Which makes sense because the loopback interface is a figment of your kernel's imagination.

I'll skip the dummy interface for now, and it may not be present on your computer. Then there are my two physical network interfaces, one at the side of my cable modem, the other serves my home ethernet segment. Furthermore, we see a ppp0 interface.

Note the absence of IP addresses. iproute disconnects the concept of 'links' and 'IP addresses'. With IP aliasing, the concept of 'the' IP address had become quite irrelevant anyhow.

It does show us the MAC addresses though, the hardware identifier of our ethernet interfaces.

`ip` shows us our IP addresses

```
[ahu@home ahu]$ ip address show
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/8 brd 10.255.255.255 scope global eth0
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast qlen 10
    link/ppp
    inet 212.64.94.251 peer 212.64.94.1/32 scope global ppp0
```

This contains more information. It shows all our addresses, and to which cards they belong. 'inet' stands for Internet (IPv4). There are lots of other address families, but these don't concern us right now.

Let's examine eth0 somewhat closer. It says that it is related to the inet address '10.0.0.1/8'. What does this mean? The /8 stands for the number of bits that are in the Network Address. There are 32 bits, so we have 24 bits left that are part of our network. The first 8 bits of 10.0.0.1 correspond to 10.0.0.0, our Network Address, and our netmask is 255.0.0.0.

The other bits are connected to this interface, so 10.250.3.13 is directly available on eth0, as is 10.0.0.1 for example.

With ppp0, the same concept goes, though the numbers are different. Its address is 212.64.94.251, without a subnet mask. This means that we have a point−to−point connection and that every address, with the exception of 212.64.94.251, is remote. There is more information however, it tells us that on the other side of the link is yet again only one address, 212.64.94.1. The /32 tells us that there are no 'network bits'.

It is absolutely vital that you grasp these concepts. Refer to the documentation mentioned at the beginning of this HOWTO if you have trouble.

You may also note 'qdisc', which stands for Queueing Discipline. This will become vital later on.

`ip` shows us our routes

Well, we now know how to find 10.x.y.z addresses, and we are able to reach 212.64.94.1. This is not enough however, so we need instructions on how to reach the world. The internet is available via our ppp connection, and it appears that 212.64.94.1 is willing to spread our packets around the world, and deliver results back to us.

```
[ahu@home ahu]$ ip route show
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

This is pretty much self explanatory. The first 4 lines of output explicitly state what was already implied by `ip address show`, the last line tells us that the rest of the world can be found via 212.64.94.1, our default gateway. We can see that it is a gateway because of the word via, which tells us that we need to send packets to 212.64.94.1, and that it will take care of things.

For reference, this is what the old 'route' utility shows us:

```
[ahu@home ahu]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
212.64.94.1     0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
0.0.0.0         212.64.94.1     0.0.0.0         UG    0      0        0 ppp0
```

3.5 ARP

ARP is the Address Resolution Protocol as described in RFC 826. ARP is used by a networked machine to resolve the hardware location/address of another machine on the same local network. Machines on the Internet are generally known by their names which resolve to IP addresses. This is how a machine on the foo.com network is able to communicate with another machine which is on the bar.net network. An IP address, though, cannot tell you the physical location of a machine. This is where ARP comes into the picture.

Let's take a very simple example. Suppose I have a network composed of several machines. Two of the machines which are currently on my network are foo with an IP address of 10.0.0.1 and bar with an IP address of 10.0.0.2. Now foo wants to ping bar to see that he is alive, but alas, foo has no idea where bar is. So when foo decides to ping bar he will need to send out an ARP request. This ARP request is akin to foo shouting out on the network "Bar (10.0.0.2)! Where are you?" As a result of this every machine on the network will hear foo shouting, but only bar (10.0.0.2) will respond. Bar will then send an ARP reply directly back to foo which is akin bar saying, "Foo (10.0.0.1) I am here at 00:60:94:E9:08:12." After this simple transaction used to locate his friend on the network foo is able to communicate with bar until he (his arp cache) forgets where bar is.

Now let's see how this works. You can view your machines current arp/neighbor cache/table like so:

```
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

As you can see my machine espa041 (9.3.76.41) knows where to find espa042 (9.3.76.42) and espagate (9.3.76.1). Now let's add another machine to the arp cache.

```
[root@espa041 /home/paulsch/.gnome-desktop]# ping -c 1 espa043
PING espa043.austin.ibm.com (9.3.76.43) from 9.3.76.41 : 56(84) bytes of data.
64 bytes from 9.3.76.43: icmp_seq=0 ttl=255 time=0.9 ms

--- espa043.austin.ibm.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.9/0.9/0.9 ms

[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0 lladdr 00:06:29:21:80:20 nud reachable
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

As a result of espa041 trying to contact espa043, espa043's hardware address/location has now been added to the arp/nieghbor cache. So until the entry for espa043 times out (as a result of no communication between the two) espa041 knows where to find espa043 and has no need to send an ARP request.

Now let's delete espa043 from our arp cache:

```
[root@espa041 /home/src/iputils]# ip neigh delete 9.3.76.43 dev eth0
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0  nud failed
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud stale
```

Now espa041 has again forgotten where to find espa043 and will need to send another ARP request the next time he needs to communicate with espa043. You can also see from the above output that espagate (9.3.76.1) has been changed to the "stale" state. This means that the location shown is still valid, but it will have to be confirmed at the first transaction to that machine.

4.Rules − routing policy database

If you have a large router, you may well cater for the needs of different people, who should be served differently. The routing policy database allows you to do this by having multiple sets of routing tables.

If you want to use this feature, make sure that your kernel is compiled with the "IP: advanced router" and "IP: policy routing" features.

When the kernel needs to make a routing decision, it finds out which table needs to be consulted. By default, there are three tables. The old 'route' tool modifies the main and local tables, as does the ip tool (by default).

The default rules:

```
[ahu@home ahu]$ ip rule list
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

This lists the priority of all rules. We see that all rules apply to all packets ('from all'). We've seen the 'main' table before, it's output by `ip route ls`, but the 'local' and 'default' table are new.

If we want to do fancy things, we generate rules which point to different tables which allow us to override system wide routing rules.

For the exact semantics on what the kernel does when there are more matching rules, see Alexey's ip−cref documentation.

4.1 Simple source routing

Let's take a real example once again, I have 2 (actually 3, about time I returned them) cable modems, connected to a Linux NAT ('masquerading') router. People living here pay me to use the internet. Suppose one of my house mates only visits hotmail and wants to pay less. This is fine with me, but you'll end up using the low−end cable modem.

The 'fast' cable modem is known as 212.64.94.251 and is an PPP link to 212.64.94.1. The 'slow' cable modem is known by various ip addresses, 212.64.78.148 in this example and is a link to 195.96.98.253.

The local table:

```
[ahu@home ahu]$ ip route list table local
broadcast 127.255.255.255 dev lo  proto kernel  scope link  src 127.0.0.1
local 10.0.0.1 dev eth0  proto kernel  scope host  src 10.0.0.1
broadcast 10.0.0.0 dev eth0  proto kernel  scope link  src 10.0.0.1
local 212.64.94.251 dev ppp0  proto kernel  scope host  src 212.64.94.251
broadcast 10.255.255.255 dev eth0  proto kernel  scope link  src 10.0.0.1
broadcast 127.0.0.0 dev lo  proto kernel  scope link  src 127.0.0.1
local 212.64.78.148 dev ppp2  proto kernel  scope host  src 212.64.78.148
local 127.0.0.1 dev lo  proto kernel  scope host  src 127.0.0.1
```

```
local 127.0.0.0/8 dev lo  proto kernel  scope host  src 127.0.0.1
```

Lots of obvious things, but things that need to specified somewhere. Well, here they are. The default table is empty.

Let's view the 'main' table:

```
[ahu@home ahu]$ ip route list table main
195.96.98.253 dev ppp2  proto kernel  scope link  src 212.64.78.148
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

We now generate a new rule which we call 'John', for our hypothetical house mate. Although we can work with pure numbers, it's far easier if we add our tables to /etc/iproute2/rt_tables.

```
# echo 200 John >> /etc/iproute2/rt_tables
# ip rule add from 10.0.0.10 table John
# ip rule ls
0:      from all lookup local
32765:  from 10.0.0.10 lookup John
32766:  from all lookup main
32767:  from all lookup default
```

Now all that is left is to generate Johns table, and flush the route cache:

```
# ip route add default via 195.96.98.253 dev ppp2 table John
# ip route flush cache
```

And we are done. It is left as an exercise for the reader to implement this in ip−up.

5.GRE and other tunnels

There are 3 kinds of tunnels in Linux. There's IP in IP tunneling, GRE tunneling and tunnels that live outside the kernel (like, for example PPTP).

5.1 A few general remarks about tunnels:

Tunnels can be used to do some very unusual and very cool stuff. They can also make things go horribly wrong when you don't configure them right. Don't point your default route to a tunnel device unless you know **exactly** what you are doing :−). Furthermore, tunneling increases overhead, because it needs an extra set of IP headers. Typically this is 20 bytes per packet, so if the normal packet size (MTU) on a network is

1500 bytes, a packet that is sent through a tunnel can only be 1480 bytes big. This is not necessarily a problem, but be sure to read up on IP packet fragmentation/reassembly when you plan to connect large networks with tunnels. Oh, and of course, the fastest way to dig a tunnel is to dig at both sides.

5.2 IP in IP tunneling

This kind of tunneling has been available in Linux for a long time. It requires 2 kernel modules, ipip.o and new_tunnel.o.

Let's say you have 3 networks: Internal networks A and B, and intermediate network C (or let's say, Internet). So we have network A:

```
network 10.0.1.0
netmask 255.255.255.0
router  10.0.1.1
```

The router has address 172.16.17.18 on network C.

and network B:

```
network 10.0.2.0
netmask 255.255.255.0
router  10.0.2.1
```

The router has address 172.19.20.21 on network C.

As far as network C is concerned, we assume that it will pass any packet sent from A to B and vice versa. You might even use the Internet for this.

Here's what you do:

First, make sure the modules are installed:

```
insmod ipip.o
insmod new_tunnel.o
```

Then, on the router of network A, you do the following:

```
ifconfig tunl0 10.0.1.1 pointopoint 172.19.20.21
route add -net 10.0.2.0 netmask 255.255.255.0 dev tunl0
```

And on the router of network B:

```
ifconfig tunl0 10.0.2.1 pointopoint 172.16.17.18
route add -net 10.0.1.0 netmask 255.255.255.0 dev tunl0
```

And if you're finished with your tunnel:

```
ifconfig tunl0 down
```

Presto, you're done. You can't forward broadcast or IPv6 traffic through an IP−in−IP tunnel, though. You just connect 2 IPv4 networks that normally wouldn't be able to talk to each other, that's all. As far as compatibility goes, this code has been around a long time, so it's compatible all the way back to 1.3 kernels. Linux IP−in−IP tunneling doesn't work with other Operating Systems or routers, as far as I know. It's simple, it works. Use it if you have to, otherwise use GRE.

5.3 GRE tunneling

GRE is a tunneling protocol that was originally developed by Cisco, and it can do a few more things than IP−in−IP tunneling. For example, you can also transport multicast traffic and IPv6 through a GRE tunnel.

In Linux, you'll need the ip_gre.o module.

IPv4 Tunneling

Let's do IPv4 tunneling first:

Let's say you have 3 networks: Internal networks A and B, and intermediate network C (or let's say, Internet).

So we have network A:

```
network 10.0.1.0
netmask 255.255.255.0
router  10.0.1.1
```

The router has address 172.16.17.18 on network C. Let's call this network neta (ok, hardly original)

and network B:

```
network 10.0.2.0
netmask 255.255.255.0
router  10.0.2.1
```

The router has address 172.19.20.21 on network C. Let's call this network netb (still not original)

As far as network C is concerned, we assume that it will pass any packet sent from A to B and vice versa. How and why, we do not care.

On the router of network A, you do the following:

```
ip tunnel add netb mode gre remote 172.19.20.21 local 172.16.17.18 ttl 255
```

```
ip addr add 10.0.1.1 dev netb
ip route add 10.0.2.0/24 dev netb
```

Let's discuss this for a bit. In line 1, we added a tunnel device, and called it netb (which is kind of obvious because that's where we want it to go). Furthermore we told it to use the GRE protocol (mode gre), that the remote address is 172.19.20.21 (the router at the other end), that our tunneling packets should originate from 172.16.17.18 (which allows your router to have several IP addresses on network C and let you decide which one to use for tunneling) and that the TTL field of the packet should be set to 255 (ttl 255).

In the second line we gave the newly born interface netb the address 10.0.1.1. This is OK for smaller networks, but when you're starting up a mining expedition (LOTS of tunnels), you might want to consider using another IP range for tunneling interfaces (in this example, you could use 10.0.3.0).

In the third line we set the route for network B. Note the different notation for the netmask. If you're not familiar with this notation, here's how it works: you write out the netmask in binary form, and you count all the ones. If you don't know how to do that, just remember that 255.0.0.0 is /8, 255.255.0.0 is /16 and 255.255.255.0 is /24. Oh, and 255.255.254.0 is /23, in case you were wondering.

But enough about this, let's go on with the router of network B.

```
ip tunnel add neta mode gre remote 172.16.17.18 local 172.19.20.21 ttl 255
ip addr add 10.0.2.1 dev neta
ip route add 10.0.1.0/24 dev neta
```

And when you want to remove the tunnelon router A:

```
ip link set netb down
ip tunnel del netb
```

Of course, you can replace netb with neta for router B.

IPv6 Tunneling

A short bit about IPv6 addresses:

IPv6 addresses are, compared to IPv4 addresses, monstrously big. An example:

`3ffe:2502:200:40:281:48fe:dcfe:d9bc`

So, to make writing them down easier, there are a few rules:

- Don't use leading zeroes. Same as in IPv4.
- Use colons to separate every 16 bits or two bytes.
- When you have lots of consecutive zeroes, you can write this down as ::. You can only do this once in an address and only for quantities of 16 bits, though.

Using these rules, the address 3ffe:0000:0000:0000:0000:0020:34A1:F32C can be written down as

3ffe::20:34A1:F32C, which is a lot shorter.

On with the tunnels.

Let's assume that you have the following IPv6 network, and you want to connect it to 6bone, or a friend.

```
Network 3ffe:406:5:1:5:a:2:1/96
```

Your IPv4 address is 172.16.17.18, and the 6bone router has IPv4 address 172.22.23.24.

```
ip tunnel add sixbone mode sit remote 172.22.23.24 local 172.16.17.18 ttl 255
ip link set sixbone up
ip addr add 3ffe:406:5:1:5:a:2:1/96 dev sixbone
ip route add 3ffe::/15 dev sixbone
```

Let's discuss this. In the first line, we created a tunnel device called sixbone. We gave it mode sit (which is IPv6 in IPv4 tunneling) and told it where to go to (remote) and where to come from (local). TTL is set to maximum, 255. Next, we made the device active (up). After that, we added our own network address, and set a route for 3ffe::/15 (which is currently all of 6bone) through the tunnel.

GRE tunnels are currently the preferred type of tunneling. It's a standard that's also widely adopted outside the Linux community and therefore a Good Thing.

5.4 Userland tunnels

There are literally dozens of implementations of tunneling outside the kernel. Best known are of course PPP and PPTP, but there are lots more (some proprietary, some secure, some that don't even use IP) and that is really beyond the scope of this HOWTO.

6.IPsec: secure IP over the internet

FIXME: Waiting for our feature editor Stefan to finish his stuf

---

7.Multicast routing

FIXME: Editor Vacancy! (somebody is working on it, though)

8.Using Class Based Queueing for bandwidth management

Now, when I discovered this, it *really* blew me away. Linux 2.2 comes with everything to manage bandwidth in ways comparable to high−end dedicated bandwidth management systems.

Linux even goes far beyond what Frame and ATM provide.

The two basic units of Traffic Control are filters and queues. Filters place traffic into queues, and queues gather traffic and decide what to send first, send later, or drop. There are several flavours of filters and queues.

The most common filters are fwmark and u32, the first lets you use the Linux netfilter code to select traffic, and the second allows you to select traffic based on ANY header. The most notable queue is Class Based Queue. CBQ is a super−queue, in that it contains other queues (even other CBQs).

It may not be immediately clear what queueing has to do with bandwidth management, but it really does work.

For our frame of reference, I have modelled this section on an ISP where I learned the ropes, so to speak, Casema Internet in The Netherlands. Casema, which is actually a cable company, has internet needs both for their customers and for their own office. Most corporate computers there have access to the internet. In reality, they have lots of money to spend and do not use Linux for bandwidth management.

We will explore how our ISP could have used Linux to manage their bandwidth.

8.1 What is queueing?

With queueing we determine the order in which data is *sent*. It it important to realise that we can only shape data that we transmit. How does this changing the order determine the speed of transmission? Imagine a cash register which is able to process 3 customers per minute.

People wishing to pay go stand in line at the 'tail end' of the queue. This is 'FIFO queueing' (First In, First Out). Let's suppose however that we let certain people always join in the middle of the queue, instead of at the end. These people spend a lot less time in the queue and are therefore able to shop faster.

With the way the internet works, we have no direct control of what people send us. It's a bit like your (physical!) mailbox at home. There is no way you can influence the world to modify the amount of mail they send you, short of contacting everybody.

However, the internet is mostly based on TCP/IP which has a few features that help us. TCP/IP has no way of knowing the capacity of the network between two hosts, so it just starts sending data faster and faster ('slow start') and when packets start getting lost, because there is no room to send them, it will slow down.

This is the equivalent of not reading half of your mail, and hoping that people will stop sending it to you. With the difference that it works for the Internet :−)

FIXME: explain congestion windows

```
[The Internet] ---<E3, T3, whatever>--- [Linux router] --- [Office+ISP]
                                      eth1          eth0
```

Now, our Linux router has two interfaces which I shall dub eth0 and eth1. eth1 is connected to our router which moves packets from to and from our fibre link.

eth0 is connected to a subnet which contains both the corporate firewall and our network head ends, through which we can connect to our customers.

Because we can only limit what we send, we need two separate but possibly very similar sets of rules. By modifying queueing on eth0, we determine how fast data gets sent to our customers, and therefore how much downstream bandwidth is available for them. Their 'download speed' in short.

On eth1, we determine how fast we send data to The Internet, how fast our users, both corporate and commercial can upload data.

8.2 First attempt at bandwidth division

CBQ enables us to generate several classes, and even classes within classes. The larger devisions might be called 'agencies'. Within these classes may be things like 'bulk' or 'interactive'.

For example, we may have a 10 megabit connection to 'the internet' which is to be shared by our customers, and our corporate needs. We should not allow a few people at the office to steal away large amounts of bandwidth which we should sell to our customers.

On the other hand, or customers should not be able to drown out the traffic from our field offices to the customer database.

Previously, one way to solve this was either to use Frame relay/ATM and create virtual circuits. This works, but frame is not very fine grained, ATM is terribly inefficient at carrying IP traffic, and neither have standardised ways to segregate different types of traffic into different VCs.

However, if you do use ATM, Linux can also happily perform deft acts of fancy traffic classification for you too. Another way is to order separate connections, but this is not very practical and also not very elegant, and still does not solve all your problems.

CBQ to the rescue!

Clearly we have two main classes, 'ISP' and 'Office'. Initially, we really don't care what the divisions do with their bandwidth, so we don't further subdivide their classes.

We decide that the customers should always be guaranteed 8 megabits of downstream traffic, and our office 2 megabits.

Setting up traffic control is done with the iproute2 tool `tc`.

```
# tc qdisc add dev eth0 root handle 10: cbq bandwidth 10Mbit avpkt 1000
```

Ok, lots of numbers here. What has happened? We have configured the 'queueing discipline' of eth0. With 'root' we denote that this is the root discipline. We have given it the handle '10:'. We want to do CBQ, so we mention that on the command line as well. We tell the kernel that it can allocate 10Mbit and that the average packet size is somewhere around 1000 octets.

Now we need to generate our root class, from which all others descend:

```
# tc class add dev eth0 parent 10:0 classid 10:1 cbq bandwidth 10Mbit rate \
  10Mbit allot 1514 weight 1Mbit prio 8 maxburst 20 avpkt 1000
```

Even more numbers to worry about – the Linux CBQ implementation is very generic. With 'parent 10:0' we indicate that this class descends from the root of qdisc handle '10:' we generated earlier. With 'classid 10:1' we name this class.

We really don't tell the kernel a lot more, we just generate a class that completely fills the available device. We also specify that the MTU (plus some overhead) is 1514 octets. We also 'weigh' this class with 1Mbit – a tuning parameter.

We now generate our ISP class:

```
# tc class add dev eth0 parent 10:1 classid 10:100 cbq bandwidth 10Mbit rate \
  8Mbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded
```

We allocate 8Mbit, and indicate that this class must not exceed this by adding the 'bounded' parameter. Otherwise this class would have started borrowing bandwidth from other classes, something we will discuss later on.

To top it off, we generate the root Office class:

```
# tc class add dev eth0 parent 10:1 classid 10:200 cbq bandwidth 10Mbit rate \
  2Mbit allot 1514 weight 200Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded
```

To make this a bit clearer, a diagram which shows our classes:

```
+-------------[10: 10Mbit]-------------------------+
|+-------------[10:1 root 10Mbit]-----------------+|
||                                                ||
|| +-----[10:100 8Mbit]---------+ [10:200 2Mbit]  ||
|| |                            | |            |  ||
|| |          ISP               | |   Office   |  ||
|| |                            | |            |  ||
|| +----------------------------+ +------------+  ||
||                                                ||
|+------------------------------------------------+|
+--------------------------------------------------+
```

Ok, now we have told the kernel what our classes are, but not yet how to manage the queues. We do this presently, in one fell swoop for both classes.

```
# tc qdisc add dev eth0 parent 10:100 sfq quantum 1514b perturb 15
# tc qdisc add dev eth0 parent 10:200 sfq quantum 1514b perturb 15
```

In this case we install the Stochastic Fairness Queueing discipline (sfq), which is not quite fair, but works well up to high bandwidths without burning up CPU cycles. There are other queueing disciplines available which are better, but need more CPU. The Token Bucket Filter is often used.

Now there is only one thing left to do and that is to explain to the kernel which packets belong to which class. Initially we will do this natively with iproute2, but more interesting applications are possible in combination with netfilter.

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 100 u32 match ip dst \
  150.151.23.24 flowid 10:200

# tc filter add dev eth0 parent 10:0 protocol ip prio 25 u32 match ip dst \
  150.151.0.0/16 flowid 10:100
```

Here is is assumed that our office hides behind a firewall with IP address 150.151.23.24 and that all our other IP addresses should be considered to be part of the ISP.

The u32 match is a very simple one – more sophisticated matching rules are possible when using netfilter to mark our packets, which we can than match on in tc.

Now we have fairly divided the downstream bandwidth, we need to do the same for the upstream. For brevity's sake, all in one go:

```
# tc qdisc add dev eth1 root handle 20: cbq bandwidth 10Mbit avpkt 1000

# tc class add dev eth1 parent 20:0 classid 20:1 cbq bandwidth 10Mbit rate \
  10Mbit allot 1514 weight 1Mbit prio 8 maxburst 20 avpkt 1000

# tc class add dev eth1 parent 20:1 classid 20:100 cbq bandwidth 10Mbit rate \
  8Mbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded

# tc class add dev eth1 parent 20:1 classid 20:200 cbq bandwidth 10Mbit rate \
  2Mbit allot 1514 weight 200Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded

# tc qdisc add dev eth1 parent 20:100 sfq quantum 1514b perturb 15
# tc qdisc add dev eth1 parent 20:200 sfq quantum 1514b perturb 15

# tc filter add dev eth1 parent 20:0 protocol ip prio 100 u32 match ip src \
  150.151.23.24 flowid 20:200
```

```
# tc filter add dev eth1 parent 20:0 protocol ip prio 25 u32 match ip src \
  150.151.0.0/16 flowid 20:100
```

8.3 What to do with excess bandwidth

In our hypothetical case, we will find that even when the ISP customers are mostly offline (say, at 8AM), our office still gets only 2Mbit, which is rather wasteful.

By removing the 'bounded' statements, classes will be able to borrow bandwidth from each other.

Some classes may not wish to borrow their bandwidth to other classes. Two rival ISPs on a single link may not want to offer each other freebees. In such a case, you can add the keyword 'isolated' at the end of your 'tc class add' lines.

8.4 Class subdivisions

FIXME: completely untested suppositions! Try this!

We can go further than this. Should the employees at the office decide to all fire up their 'napster' clients, it is still possible that our database runs out of bandwidth. Therefore, we create two subclasses, 'Human' and 'Database'.

Our database always needs 500Kbit, so we have 1.5Mbit left for Human consumption.

We now need to create two new classes, within our Office class:

```
# tc class add dev eth0 parent 10:200 classid 10:250 cbq bandwidth 10Mbit rate \
  500Kbit allot 1514 weight 50Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded

# tc class add dev eth0 parent 10:200 classid 10:251 cbq bandwidth 10Mbit rate \
  1500Kbit allot 1514 weight 150Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded
```

FIXME: Finish this example!

8.5 Loadsharing over multiple interfaces

FIXME: document TEQL

9.More queueing disciplines

The Linux kernel offers us lots of queueing disciplines. By far the most widely used is the pfifo_fast queue − this is the default. This also explains why these advanced features are so robust. They are nothing more than 'just another queue'.

Each of these queues has specific strengths and weaknesses. Not all of them may be as well tested.

9.1 pfifo_fast

This queue is, as the name says, First In, First Out, which means that no packet receives special treatment. At least, not quite. This queue has 3 so called 'bands'. Within each band, FIFO rules apply. However, as long as there are packets waiting in band 0, band 1 won't be processed. Same goes for band 1 and band 2.

9.2 Stochastic Fairness Queueing

SFQ, as said earlier, is not quite deterministic, but works (on average). Its main benefits are that it requires little CPU and memory. 'Real' fair queueing requires that the kernel keep track of all running sessions.

Stochastic Fairness Queueing (SFQ) is a simple implementation of fair queueing algorithms family. It's less accurate than others, but it also requires less calculations while being almost perfectly fair.

The key word in SFQ is conversation (or flow), being a sequence of data packets having enough common parameters to distinguish it from other conversations. The parameters used in case of IP packets are source and destination address, and the protocol number.

SFQ consists of dynamically allocated number of FIFO queues, one queue for one conversation. The discipline runs in round−robin, sending one packet from each FIFO in one turn, and this is why it's called fair. The main advantage of SFQ is that it allows fair sharing the link between several applications and prevent bandwidth take−over by one client. SFQ however cannot determine interactive flows from bulk ones −− one usually needs to do the selection with CBQ before, and then direct the bulk traffic into SFQ.

9.3 Token Bucket Filter

The Token Bucket Filter (TBF) is a simple queue, that only passes packets arriving at rate in bounds of some administratively set limit, with possibility to buffer short bursts.

The TBF implementation consists of a buffer (bucket), constatly filled by some virtual pieces of information called tokens, at specific rate (token rate). The most important parameter of the bucket is its size, that is number of tokens it can store.

Each arriving token lets one incoming data packet of out the queue and is then deleted from the bucket. Associating this algorithm with the two flows −− token and data, gives us three possible scenarios:

- The data arrives into TBF at rate *equal* the rate of incoming tokens. In this case each incoming packet has its matching token and passes the queue without delay.
- The data arrives into TBF at rate *smaller* than the token rate. Only some tokens are deleted at output of each data packet sent out the queue, so the tokens accumulate, up to the bucket size. The saved tokens can be then used to send data over the token rate, if short data burst occurs.
- The data arrives into TBF at rate *bigger* than the token rate. In this case filter overrun occurs –– incoming data can be only sent out without loss until all accumulated tokens are used. After that, overlimit packets are dropped.

The last scenario is very important, because it allows to administratively shape the bandwidth available to data, passing the filter. The accumulation of tokens allows short burst of overlimit data to be still passed without loss, but any lasting overload will cause packets to be constantly dropped.

The Linux kernel seems to go beyond this specification, and also allows us to limit the speed of the burst transmission. However, Alexey warns us:

```
Note that the peak rate TBF is much more tough: with MTU 1500
P_crit = 150Kbytes/sec. So, if you need greater peak rates,
use alpha with HZ=1000 :-)
```

FIXME: is this still true with TSC (pentium+)? Well sort of

FIXME: if not, add section on raising HZ

9.4 Random Early Detect

RED has some extra smartness built in. When a TCP/IP session starts, neither end knows the amount of bandwidth available. So TCP/IP starts to transmit slowly and goes faster and faster, though limited by the latency at which ACKs return.

Once a link is filling up, RED starts dropping packets, which indicate to TCP/IP that the link is congested, and that it should slow down. The smart bit is that RED simulates real congestion, and starts to drop some packets some time before the link is entirely filled up. Once the link is completely saturated, it behaves like a normal policer.

For more information on this, see the Backbone chapter.

9.5 Ingress policer qdisc

The Ingress qdisc comes in handy if you need to ratelimit a host without help from routers or other Linux boxes. You can police incoming bandwidth and drop packets when this bandwidth exceeds your desired rate. This can save your host from a SYN flood, for example, and also works to slow down TCP/IP, which responds to dropped packets by reducing speed.

FIXME: instead of dropping, can we also assign it to a real queue?

FIXME: shaping by dropping packets seems less desirable than using, for example, a token bucket filter. Not sure though, Cisco CAR works this way, and people appear happy with it.

See the reference to IOS Committed Access Rate at the end of this document.

In short: you can use this to limit how fast your computer downloads files, thus leaving more of the available bandwidth for others.

See the section on protecting your host from SYN floods for an example on how this works.

9.6 WRR

This qdisc is not included in the standard kernels but can be downloaded from http://wipl−wrr.dkik.dk/wrr/. Currently the qdisc is only tested with Linux 2.2 kernels but it will probably work with 2.4 kernels too.

The WRR qdisc distributes bandwidth between its classes using the weighted round robin scheme. That is, like the CBQ qdisc it contains classes into which arbitrary qdiscs can be plugged. All classes which have sufficient demand will get bandwidth proportional to the weights associated with the classes. The weights can be set manually using the `tc` program. But they can also be made automatically decreasing for classes transferring much data.

The qdisc has a build−in classifier which assigns packets coming from or sent to different machines to different classes. Either the MAC or IP and either source or destination addresses can be used. The MAC address can only be used when the Linux box is acting as an ethernet bridge, however. The classes are automatically assigned to machines based on the packets seen.

The qdisc can be very useful at sites such as dorms where a lot of unrelated individuals share an Internet connection. A set of scripts setting up a relevant behavior for such a site is a central part of the WRR distribution.

10.Netfilter & iproute − marking packets

So far we've seen how iproute works, and netfilter was mentioned a few times. This would be a good time to browse through Rusty's Remarkably Unreliable Guides. Netfilter itself can be found here.

Netfilter allows us to filter packets, or mangle their headers. One special feature is that we can mark a packet with a number. This is done with the −−set−mark facility.

As an example, this command marks all packets destined for port 25, outgoing mail:

```
# iptables −A PREROUTING −i eth0 −t mangle −p tcp −−dport 25 \
 −j MARK −−set−mark 1
```

Let's say that we have multiple connections, one that is fast (and expensive, per megabyte) and one that is slower, but flat fee. We would most certainly like outgoing mail to go via the cheap route.

We've already marked the packets with a '1', we now instruct the routing policy database to act on this:

```
# echo 201 mail.out >> /etc/iproute2/rt_tables
# ip rule add fwmark 1 table mail.out
# ip rule ls
0:      from all lookup local
32764:  from all fwmark        1 lookup mail.out
32766:  from all lookup main
32767:  from all lookup default
```

Now we generate the mail.out table with a route to the slow but cheap link:

```
# /sbin/ip route add default via 195.96.98.253 dev ppp0 table mail.out
```

And we are done. Should we want to make exceptions, there are lots of ways to achieve this. We can modify the netfilter statement to exclude certain hosts, or we can insert a rule with a lower priority that points to the main table for our excepted hosts.

We can also use this feature to honour TOS bits by marking packets with a different type of service with different numbers, and creating rules to act on that. This way you can even dedicate, say, an ISDN line to interactive sessions.

Needless to say, this also works fine on a host that's doing NAT ('masquerading').

Note: for this to work, you need to have some options enabled in your kernel:

```
IP: advanced router (CONFIG_IP_ADVANCED_ROUTER) [Y/n/?]
 IP: policy routing (CONFIG_IP_MULTIPLE_TABLES) [Y/n/?]
  IP: use netfilter MARK value as routing key (CONFIG_IP_ROUTE_FWMARK) [Y/n/?]
```

See also Transparent web−caching using netfilter, iproute2, ipchains and squid in the Cookbook.

11.More classifiers

Classifiers are the way by which the kernel decides which queue a packet should be placed into. There are various different classifiers, each of which can be used for different purposes.

fw

Bases the decision on how the firewall has marked the packet.

u32

Bases the decision on fields within the packet (i.e. source IP address, etc)

route

Bases the decision on which route the packet will be routed by.

rsvp, rsvp6

Bases the decision on the target (destination,protocol) and optionally the source as well. (I think)

tcindex

FIXME: Fill me in

Note that in general there are many ways in which you can classify packet and that it generally comes down to preference as to which system you wish to use.

Classifiers in general accept a few arguments in common. They are listed here for convenience:

protocol

The protocol this classifier will accept. Generally you will only be accepting only IP traffic. Required.

parent

The handle this classifier is to be attached to. This handle must be an already existing class. Required.

prio

The priority of this classifier. Higher numbers get tested first.

handle

This handle means different things to different filters.
FIXME: Add more

All the following sections will assume you are trying to shape the traffic going to `HostA`. They will assume that the root class has been configured on 1: and that the class you want to send the selected traffic to is 1:1.

11.1 The "fw" classifier

The "fw" classifier relies on the firewall tagging the packets to be shaped. So, first we will setup the firewall to tag them:

```
# iptables -I PREROUTING -t mangle -p tcp -d HostA \
 -j MARK --set-mark 1
```

Now all packets to that machine are tagged with the mark 1. Now we build the packet shaping rules to actually shape the packets. Now we just need to indicate that we want the packets that are tagged with the mark 1 to go to class 1:1. This is accomplished with the command:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 1 fw classid 1:1
```

This should be fairly self–explanatory. Attach to the 1:0 class a filter with priority 1 to filter all packet marked with 1 in the firewall to class 1:1. Note how the handle here is used to indicate what the mark should be.

That's all there is to it! This is the (IMHO) easy way, the other ways are I think harder to understand. Note that you can apply the full power of the firewalling code with this classifier, including matching MAC addresses, user IDs and anything else the firewall can match.

11.2 The "u32" classifier

The U32 filter is the most advanced filter available in the current implementation. It entirely based on hashing tables, which make it robust when there are many filter rules.

In its simplest form the U32 filter is a list of records, each consisting of two fields: a selector and an action. The selectors, described below, are compared with the currently processed IP packet until the first match and the associated action is performed. The simplest type of action would be directing the packet into defined CBQ class.

The commandline of `tc filter` program, used to configure the filter, consists of three parts: filter specification, a selector and an action. The filter specification can be defined as:

```
tc filter add dev IF [ protocol PROTO ]
                     [ (preference|priority) PRIO ]
                     [ parent CBQ ]
```

The `protocol` field describes protocol that the filter will be applied to. We will only discuss case of `ip` protocol. The `preference` field (`priority` can be used alternatively) sets the priority of currently defined filter. This is important, since you can have several filters (lists of rules) with different priorities. Each list will be passed in the order the rules were added, then list with lower priority (higher preference number) will be processed. The `parent` field defines the CBQ tree top (e.g. 1:0), the filter should be attached to.

The options decribed apply to all filters, not only U32.

U32 selector

The U32 selector contains definition of the pattern, that will be matched to the currently processed packet. Precisely, it defines which bits are to be matched in the packet header and nothing more, but this simple method is very powerful. Let's take a look at the following examples, taken directly from a pretty complex, real−world filter:

```
# filter parent 1: protocol ip pref 10 u32 fh 800::800 order 2048 key ht 800 bkt 0 flowid 1:3 \
  match 00100000/00ff0000 at 0
```

For now, leave the first line alone − all these parameters describe the filter's hash tables. Focus on the selector line, containing `match` keyword. This selector will match to IP headers, whose second byte will be 0x10 (0010). As you can guess, the 00ff number is the match mask, telling the filter exactly which bits to match. Here it's 0xff, so the byte will match if it's exactly 0x10. The `at` keyword means that the match is to be started at specified offset (in bytes) −− in this case it's beginning of the packet. Translating all that to human language, the packet will match if its Type of Service field will have `low delay' bits set. Let's analyze another rule:

```
# filter parent 1: protocol ip pref 10 u32 fh 800::803 order 2051 key ht 800 bkt 0 flowid 1:3 \
  match 00000016/0000ffff at nexthdr+0
```

The `nexthdr` option means next header encapsulated in the IP packet, i.e. header of upper−layer protocol. The match will also start here at the beginning of the next header. The match should occur in the second, 32−bit word of the header. In TCP and UDP protocols this field contains packet's destination port. The number is given in big−endian format, i.e. older bits first, so we simply read 0x0016 as 22 decimal, which stands for SSH service if this was TCP. As you guess, this match is ambigous without a context, and we will discuss this later.

Having understood all the above, we will find the following selector quite easy to read: `match c0a80100/ffffff00 at 16`. What we got here is a three byte match at 17−th byte, counting from the IP header start. This will match for packets with destination address anywhere in 192.168.1/24 network. After analyzing the examples, we can summarize what we have learnt.

General selectors

General selectors define the pattern, mask and offset the pattern will be matched to the packet contents. Using the general selectors you can match virtually any single bit in the IP (or upper layer) header. They are more difficult to write and read, though, than specific selectors that described below. The general selector syntax is:

```
match [ u32 | u16 | u8 ] PATTERN MASK [ at OFFSET | nexthdr+OFFSET]
```

One of the keywords `u32`, `u16` or `u8` specifies length of the pattern in bits. PATTERN and MASK should follow, of length defined by the previous keyword. The OFFSET parameter is the offset, in bytes, to start matching. If `nexthdr+` keyword is given, the offset is relative to start of the upper layer header.

Some examples:

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match u8 64 0xff at 8 \
     flowid 1:4
```

Packet will match to this rule, if its time to live (TTL) is 64. TTL is the field starting just after 8−th byte of the IP header.

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match u8 0x10 0xff at nexthdr+13 \
     protocol tcp \
     flowid 1:3 \
```

This rule will only match TCP packets with ACK bit set. Here we can see an example of using two selectors, the final result will be logical AND of their results. If we take a look at TCP header diagram, we can see that the ACK bit is second older bit (0x10) in the 14−th byte of the TCP header (`at nexthdr+13`). As for the second selector, if we'd like to make our life harder, we could write `match u8 0x06 0xff at 9` instead if using the specific selector `protocol tcp`, because 6 is the number of TCP protocol, present in 10−th byte of the IP header. On the other hand, in this example we couldn't use any specific selector for the first match − simply because there's no specific selector to match TCP ACK bits.

Specific selectors

The following table contains a list of all specific selectors the author of this section has found in the `tc` program source code. They simply make your life easier and increase readability of your filter's configuration.

FIXME: table placeholder − the table is in separate file ,,selector.html''

FIXME: it's also still in Polish :−(

FIXME: must be sgml'ized

Some examples:

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
     match ip tos 0x10 0xff \
     flowid 1:4
```

The above rule will match packets, which have the TOS field set to 0x10. The TOS field starts at second byte of the packet and is one byte big, so we coul write an equivalent general selector: `match u8 0x10 0xff at 1`. This gives us hint to the internals of U32 filter −− the specific rules are always translated to general ones, and in this form they are stored in the kernel memory. This leads to another conclusion −− the `tcp` and `udp` selectors are exactly the same and this is why you can't use single `match tcp dst 53 0xffff` selector to match TCP packets sent to given port −− they will also match UDP packets sent to this port. You must remember to also specify the protocol and end up with the following rule:

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
        match tcp dst 53 0xffff \
        match ip protocol 0x6 0xff \
        flowid 1:2
```

11.3 The "route" classifier

This classifier filters based on the results of the routing tables. When a packet that is traversing through the classes reaches one that is marked with the "route" filter, it splits the packets up based on information in the routing table.

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 route
```

Here we add a route classifier onto the parent node 1:0 with priority 100. When a packet reaches this node (which, since it is the root, will happen immediately) it will consult the routing table and if one matches will send it to the given class and give it a priority of 100. Then, to finally kick it into action, you add the appropriate routing entry:

The trick here is to define 'realm' based on either destination or source. The way to do it is like this:

```
# ip route add Host/Network via Gateway dev Device realm RealmNumber
```

For instance, we can define our destination network 192.168.10.0 with a realm number 10:

```
# ip route add 192.168.10.0/24 via 192.168.10.1 dev eth1 realm 10
```

When adding route filters, we can use realm numbers to represent the networks or hosts and specify how the routes match the filters.

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route to 10 classid 1:10
```

The above rule says packets going to the network 192.168.10.0 match class id 1:10.

Route filter can also be used to match source routes. For example, there is a subnetwork attached to the Linux router on eth2.

```
# ip route add 192.168.2.0/24 dev eth2 realm 2
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route from 2 classid 1:2
```

Here the filter specifies that packets from the subnetwork 192.168.2.0 (realm 2) will match class id 1:2.

11.4 The "rsvp" classifier

FIXME: Fill me in

11.5 The "tcindex" classifier

FIXME: Fill me in

12.Kernel network parameters

The kernel has lots of parameters which can be tuned for different circumstances. While, as usual, the default parameters serve 99% of installations very well, we don't call this the Advanced HOWTO for the fun of it!

The interesting bits are in /proc/sys/net, take a look there. Not everything will be documented here initially, but we're working on it.

12.1 Reverse Path Filtering

By default, routers route everything, even packets which 'obviously' don't belong on your network. A common example is private IP space escaping onto the internet. If you have an interface with a route of 195.96.96.0/24 to it, you do not expect packets from 212.64.94.1 to arrive there.

Lots of people will want to turn this feature off, so the kernel hackers have made it easy. There are files in /proc where you can tell the kernel to do this for you. The method is called "Reverse Path Filtering". Basically, if the reply to this packet wouldn't go out the interface this packet came in, then this is a bogus packet and should be ignored.

The following fragment will turn this on for all current and future interfaces.

```
# for i in /proc/sys/net/ipv4/conf/*/rp_filter ; do
>  echo 2 > $i
> done
```

Going by the example above, if a packet arrived on the Linux router on eth1 claiming to come from the Office+ISP subnet, it would be dropped. Similarly, if a packet came from the Office subnet, claiming to be from somewhere outside your firewall, it would be dropped also.

The above is full reverse path filtering. The default is to only filter based on IPs that are on directly connected networks. This is because the full filtering breaks in the case of asymmetric routing (where packets come in one way and go out another, like satellite traffic, or if you have dynamic (bgp, ospf, rip) routes in your network. The data comes down through the satellite dish and replies go back through normal land−lines).

If this exception applies to you (and you'll probably know if it does) you can simply turn off the rp_filter on the interface where the satellite data comes in. If you want to see if any packets are being dropped, the log_martians file in the same directory will tell the kernel to log them to your syslog.

```
# echo 1 >/proc/sys/net/ipv4/conf/<interfacename>/log_martians
```

FIXME: is setting the conf/{default,all}/* files enough? − martijn

12.2 Obscure settings

Ok, there are a lot of parameters which can be modified. We try to list them all. Also documented (partly) in Documentation/ip−sysctl.txt.

Some of these settings have different defaults based on wether you answered 'Yes' to 'Configure as router and not host' while compiling your kernel.

Generic ipv4

As a generic note, most rate limiting features don't work on loopback, so don't test them locally. The limits are supplied in 'jiffies', and are enforced using the earlier mentioned token bucket filter.

The kernel has an internal clock which runs at 'HZ' ticks (or 'jiffies') per second. On intel, 'HZ' is mostly 100. So setting a *_rate file to, say 50, would allow for 2 packets per second. The token bucket filter is also configured to allow for a burst of at most 6 packets, if enough tokens have been earned.

/proc/sys/net/ipv4/icmp_destunreach_rate

If the kernel decides that it can't deliver a packet, it will drop it, and send the source of the packet an ICMP notice to this effect.

/proc/sys/net/ipv4/icmp_echo_ignore_all

Don't act on echo packets at all. Please don't set this by default, but if you are used as a relay in a DoS attack, it may be useful.

/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts [Useful]

If you ping the broadcast address of a network, all hosts are supposed to respond. This makes for a dandy denial−of−service tool. Set this to 1 to ignore these broadcast messages.

/proc/sys/net/ipv4/icmp_echoreply_rate

The rate at which echo replies are sent to any one destination.

/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses

FIXME: fill this in

/proc/sys/net/ipv4/icmp_paramprob_rate

FIXME: fill this in

/proc/sys/net/ipv4/icmp_timeexceed_rate

This the famous cause of the 'Solaris middle star' in traceroutes. Limits number of ICMP Time Exceeded messages sent.

/proc/sys/net/ipv4/igmp_max_memberships

FIXME: fill this in

/proc/sys/net/ipv4/inet_peer_gc_maxtime

FIXME: fill this in

/proc/sys/net/ipv4/inet_peer_gc_mintime

FIXME: fill this in

/proc/sys/net/ipv4/inet_peer_maxttl

FIXME: fill this in

/proc/sys/net/ipv4/inet_peer_minttl

FIXME: fill this in

/proc/sys/net/ipv4/inet_peer_threshold

FIXME: fill this in

/proc/sys/net/ipv4/ip_autoconfig

FIXME: fill this in

/proc/sys/net/ipv4/ip_default_ttl

Time To Live of packets. Set to a safe 64. Raise it if you have a huge network. Don't do so for fun − routing loops cause much more damage that way. You might even consider lowering it in some circumstances.

/proc/sys/net/ipv4/ip_dynaddr

You need to set this if you use dial−on−demand with a dynamic interface address. Once your demand interface comes up, any local TCP sockets which haven't seen replies will be rebound to have the right address. This solves the problem that the connection that brings up your interface itself does not work, but the second try does.

/proc/sys/net/ipv4/ip_forward

If the kernel should attempt to forward packets. Off by default.

/proc/sys/net/ipv4/ip_local_port_range

Range of local ports for outgoing connections. Actually quite small by default, 1024 to 4999.

/proc/sys/net/ipv4/ip_no_pmtu_disc

Set this if you want to disable Path MTU discovery − a technique to determine the largest Maximum Transfer Unit possible on your path.

/proc/sys/net/ipv4/ipfrag_high_thresh

FIXME: fill this in

/proc/sys/net/ipv4/ipfrag_low_thresh

FIXME: fill this in

/proc/sys/net/ipv4/ipfrag_time

FIXME: fill this in

/proc/sys/net/ipv4/tcp_abort_on_overflow

FIXME: fill this in

/proc/sys/net/ipv4/tcp_fin_timeout

FIXME: fill this in

/proc/sys/net/ipv4/tcp_keepalive_intvl

FIXME: fill this in

/proc/sys/net/ipv4/tcp_keepalive_probes

FIXME: fill this in

/proc/sys/net/ipv4/tcp_keepalive_time

FIXME: fill this in

/proc/sys/net/ipv4/tcp_max_orphans

FIXME: fill this in

/proc/sys/net/ipv4/tcp_max_syn_backlog

FIXME: fill this in

/proc/sys/net/ipv4/tcp_max_tw_buckets

FIXME: fill this in

/proc/sys/net/ipv4/tcp_orphan_retries

FIXME: fill this in

/proc/sys/net/ipv4/tcp_retrans_collapse

FIXME: fill this in

/proc/sys/net/ipv4/tcp_retries1

FIXME: fill this in

/proc/sys/net/ipv4/tcp_retries2

FIXME: fill this in

/proc/sys/net/ipv4/tcp_rfc1337

FIXME: fill this in

/proc/sys/net/ipv4/tcp_sack

Use Selective ACK which can be used to signify that specific packets are missing − therefore helping fast recovery.

/proc/sys/net/ipv4/tcp_stdurg

FIXME: fill this in

/proc/sys/net/ipv4/tcp_syn_retries

Number of SYN packets the kernel will send before giving up on the new connection.

/proc/sys/net/ipv4/tcp_synack_retries

To open the other side of the connection, the kernel sends a SYN with a piggybacked ACK on it, to acknowledge the earlier received SYN. This is part 2 of the threeway handshake. This setting determines the number of SYN+ACK packets send before the kernel gives up on the connection.

/proc/sys/net/ipv4/tcp_timestamps

Timestamps are used, amongst other things, to protect against wrapping sequence numbers. A 1 gigabit link might conceivably re−encounter a previous sequence number with an out−of−line value, because if was of a previous generation. The timestamp will let it recognise this 'ancient packet'.

/proc/sys/net/ipv4/tcp_tw_recycle

FIXME: fill this in

/proc/sys/net/ipv4/tcp_window_scaling

TCP/IP normally allows windows up to 65535 bytes big. For really fast networks, this may not be enough. The window scaling options allows for almost gigabyte windows, which is good for high bandwidth*delay products.

Per device settings

DEV can either stand for a real interface, or for 'all' or 'default'. Default also changes settings for interfaces yet to be created.

/proc/sys/net/ipv4/conf/DEV/accept_redirects

If a router decides that you are using it for a wrong purpose (ie, it needs to resend your packet on the same interface), it will send us a ICMP Redirect. This is a slight security risk however, so you may want to turn it off, or use secure redirects.

/proc/sys/net/ipv4/conf/DEV/accept_source_route

Not used very much anymore. You used to be able to give a packet a list of IP addresses it should visit on its way. Linux can be made to honor this IP option.

/proc/sys/net/ipv4/conf/DEV/bootp_relay

FIXME: fill this in

/proc/sys/net/ipv4/conf/DEV/forwarding

FIXME:

/proc/sys/net/ipv4/conf/DEV/log_martians

See the section on reverse path filters.

/proc/sys/net/ipv4/conf/DEV/mc_forwarding

If we do multicast forwarding on this interface

/proc/sys/net/ipv4/conf/DEV/proxy_arp

If you set this to 1, all other interfaces will respond to arp queries destined for addresses on this interface. Can be very useful when building 'ip pseudo bridges'. Do take care that your netmasks are very correct before enabling this!

/proc/sys/net/ipv4/conf/DEV/rp_filter

See the section on reverse path filters.

/proc/sys/net/ipv4/conf/DEV/secure_redirects

FIXME: fill this in

/proc/sys/net/ipv4/conf/DEV/send_redirects

If we send the above mentioned redirects.

/proc/sys/net/ipv4/conf/DEV/shared_media

FIXME: fill this in

/proc/sys/net/ipv4/conf/DEV/tag

FIXME: fill this in

Neighbor pollicy

Dev can either stand for a real interface, or for 'all' or 'default'. Default also changes settings for interfaces yet to be created.

/proc/sys/net/ipv4/neigh/DEV/anycast_delay

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/app_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/base_reachable_time

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/delay_first_probe_time

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/gc_stale_time

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/locktime

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/mcast_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/proxy_delay

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/proxy_qlen

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/retrans_time

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/ucast_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neigh/DEV/unres_qlen

FIXME: fill this in

Routing settings

/proc/sys/net/ipv4/route/error_burst

FIXME: fill this in

/proc/sys/net/ipv4/route/error_cost

FIXME: fill this in

/proc/sys/net/ipv4/route/flush

FIXME: fill this in

/proc/sys/net/ipv4/route/gc_elasticity

FIXME: fill this in

/proc/sys/net/ipv4/route/gc_interval

FIXME: fill this in

/proc/sys/net/ipv4/route/gc_min_interval

FIXME: fill this in

/proc/sys/net/ipv4/route/gc_thresh

FIXME: fill this in

/proc/sys/net/ipv4/route/gc_timeout

FIXME: fill this in

/proc/sys/net/ipv4/route/max_delay

FIXME: fill this in

/proc/sys/net/ipv4/route/max_size

FIXME: fill this in

/proc/sys/net/ipv4/route/min_adv_mss

FIXME: fill this in

/proc/sys/net/ipv4/route/min_delay

FIXME: fill this in

/proc/sys/net/ipv4/route/min_pmtu

FIXME: fill this in

/proc/sys/net/ipv4/route/mtu_expires

FIXME: fill this in

/proc/sys/net/ipv4/route/redirect_load

FIXME: fill this in

/proc/sys/net/ipv4/route/redirect_number

FIXME: fill this in

/proc/sys/net/ipv4/route/redirect_silence

FIXME: fill this in

13.Backbone applications of traffic control

This chapter is meant as an introduction to backbone routing, which often involves >100 megabit bandwidths, which requires a different approach then your ADSL modem at home.

13.1 Router queues

The normal behaviour of router queues on the Internet is called tail−drop. Tail−drop works by queueing up to a certain amount, then dropping all traffic that 'spills over'. This is very unfair, and also leads to retransmit synchronisation. When retransmit synchronisation occurs, the sudden burst of drops from a router that has reached its fill will cause a delayed burst of retransmits, which will over fill the congested router again.

In order to cope with transient congestion on links, backbone routers will often implement large queues. Unfortunately, while these queues are good for throughput, they can substantially increase latency and cause TCP connections to behave very bursty during congestion.

These issues with tail−drop are becoming increasingly troublesome on the Internet because the use of network unfriendly applications is increasing. The Linux kernel offers us RED, short for Random Early Detect.

RED isn't a cure−all for this, applications which inappropriately fail to implement exponential backoff still get an unfair share of the bandwidth, however, with RED they do not cause as much harm to the throughput and latency of other connections.

RED statistically drops packets from flows before it reaches its hard limit. This causes a congested backbone link to slow more gracefully, and prevents retransmit synchronisation. This also helps TCP find its 'fair' speed faster by allowing some packets to get dropped sooner keeping queue sizes low and latency under control. The probability of a packet being dropped from a particular connection is proportional to its bandwidth usage rather then the number of packets it transmits.

RED is a good queue for backbones, where you can't afford the complexity of per−session state tracking needed by fairness queueing.

In order to use RED, you must decide on three parameters: Min, Max, and burst. Min sets the minimum queue size in bytes before dropping will begin, Max is a soft maximum that the algorithm will attempt to stay under, and burst sets the maximum number of packets that can 'burst through'.

You should set the min by calculating that highest acceptable base queueing latency you wish, and multiply it by your bandwidth. For instance, on my 64kbit/s ISDN link, I might want a base queueing latency of 200ms so I set min to 1600 bytes. Setting min too small will degrade throughput and too large will degrade latency. Setting a small min is not a replacement for reducing the MTU on a slow link to improve interactive response.

You should make max at least twice min to prevent synchronisation. On slow links with small min's it might be wise to make max perhaps four or more times large then min.

Burst controls how the RED algorithm responds to bursts. Burst must be set larger then min/avpkt. Experimentally, I've found (min+min+max)/(3*avpkt) to work okay.

Additionally, you need to set limit and avpkt. Limit is a safety value, after there are limit bytes in the queue, RED 'turns into' tail−drop. I typical set limit to eight times max. Avpkt should be your average packet size. 1000 works okay on high speed Internet links with a 1500byte MTU.

Read the paper on RED queueing by Sally Floyd and Van Jacobson for technical information.

FIXME: more needed. This means *you* greg :−) − ahu

14.Cookbook

This section contains 'cookbook' entries which may help you solve problems. A cookbook is no replacement for understanding however, so try and comprehend what is going on.

14.1 Running multiple sites with different SLAs

You can do this in several ways. Apache has some support for this with a module, but we'll show how Linux can do this for you, and do so for other services as well. These commands are stolen from a presentation by Jamal Hadi that's referenced below.

Let's say we have two customers, with http, ftp and streaming audio, and we want to sell them a limited amount of bandwidth. We do so on the server itself.

Customer A should have at most 2 megabits, cusomer B has paid for 5 megabits. We separate our customers by creating virtual IP addresses on our server.

```
# ip address add 188.177.166.1 dev eth0
# ip address add 188.177.166.2 dev eth0
```

It is up to you to attach the different servers to the right IP address. All popular daemons have support for this.

We first attach a CBQ qdisc to eth0:

```
# tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit cell 8 avpkt 1000 \
  mpu 64
```

We then create classes for our customers:

```
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit rate \
```

```
  2MBit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1 maxburst 21
# tc class add dev eth0 parent 1:0 classid 1:2 cbq bandwidth 10Mbit rate \
  5Mbit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1 maxburst 21
```

Then we add filters for our two classes:

```
##FIXME: Why this line, what does it do?, what is a divisor?:
##FIXME: A divisor has something to do with a hash table, and the number of
##       buckets - ahu
# tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 1: u32 divisor 1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src 188.177.166.1
  flowid 1:1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src 188.177.166.2
  flowid 1:2
```

And we're done.

FIXME: why no token bucket filter? is there a default pfifo_fast fallback somewhere?

14.2 Protecting your host from SYN floods

From Alexey's iproute documentation, adapted to netfilter and with more plausible paths. If you use this, take care to adjust the numbers to reasonable values for your system.

If you want to protect an entire network, skip this script, which is best suited for a single host.

```
#! /bin/sh -x
#
# sample script on using the ingress capabilities
# this script shows how one can rate limit incoming SYNs
# Useful for TCP-SYN attack protection. You can use
# IPchains to have more powerful additions to the SYN (eg
# in addition the subnet)
#
#path to various utilities;
#change to reflect yours.
#
TC=/sbin/tc
IP=/sbin/ip
IPTABLES=/sbin/iptables
INDEV=eth2
#
# tag all incoming SYN packets through $INDEV as mark value 1
############################################################
$iptables -A PREROUTING -i $INDEV -t mangle -p tcp --syn \
  -j MARK --set-mark 1
############################################################
#
# install the ingress qdisc on the ingress interface
############################################################
$TC qdisc add dev $INDEV handle ffff: ingress
```

```
############################################################

#
#
# SYN packets are 40 bytes (320 bits) so three SYNs equals
# 960 bits (approximately 1kbit); so we rate limit below
# the incoming SYNs to 3/sec (not very useful really; but
#serves to show the point - JHS
############################################################
$TC filter add dev $INDEV parent ffff: protocol ip prio 50 handle 1 fw \
police rate 1kbit burst 40 mtu 9k drop flowid :1
############################################################


#
echo "---- qdisc parameters Ingress  ----------"
$TC qdisc ls dev $INDEV
echo "---- Class parameters Ingress  ----------"
$TC class ls dev $INDEV
echo "---- filter parameters Ingress ----------"
$TC filter ls dev $INDEV parent ffff:

#deleting the ingress qdisc
#$TC qdisc del $INDEV ingress
```

14.3 Ratelimit ICMP to prevent dDoS

Recently, distributed denial of service attacks have become a major nuisance on the internet. By properly filtering and ratelimiting your network, you can both prevent becoming a casualty or the cause of these attacks.

You should filter your networks so that you do not allow non−local IP source addressed packets to leave your network. This stops people from anonymously sending junk to the internet.

Rate limiting goes much as shown earlier. To refresh your memory, our ASCIIgram again:

```
[The Internet] ---<E3, T3, whatever>--- [Linux router] --- [Office+ISP]
                                      eth1          eth0
```

We first set up the prerequisite parts:

```
# tc qdisc add dev eth0 root handle 10: cbq bandwidth 10Mbit avpkt 1000
# tc class add dev eth0 parent 10:0 classid 10:1 cbq bandwidth 10Mbit rate \
  10Mbit allot 1514 prio 5 maxburst 20 avpkt 1000
```

If you have 100Mbit, or more, interfaces, adjust these numbers. Now you need to determine how much ICMP traffic you want to allow. You can perform measurements with tcpdump, by having it write to a file for a while, and seeing how much ICMP passes your network. Do not forget to raise the snapshot length!

If measurement is impractical, you might want to choose 5% of your available bandwidth. Let's set up our class:

```
# tc class add dev eth0 parent 10:1 classid 10:100 cbq bandwidth 10Mbit rate \
  100Kbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 250 \
  bounded
```

This limits at 100Kbit. Now we need a filter to assign ICMP traffic to this class:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 100 u32 match ip
  protocol 1 0xFF flowid 10:100
```

14.4 Prioritizing interactive traffic

If lots of data is coming down your link, or going up for that matter, and you are trying to do some maintenance via telnet or ssh, this may not go too well. Other packets are blocking your keystrokes. Wouldn't it be great if there were a way for your interactive packets to sneak past the bulk traffic? Linux can do this for you!

As before, we need to handle traffic going both ways. Evidently, this works best if there are Linux boxes on both ends of your link, although other UNIX's are able to do this. Consult your local Solaris/BSD guru for this.

The standard pfifo_fast scheduler has 3 different 'bands'. Traffic in band 0 is transmitted first, after which traffic in band 1 and 2 gets considered. It is vital that our interactive traffic be in band 0!

We blatantly adapt from the (soon to be obsolete) ipchains HOWTO:

There are four seldom−used bits in the IP header, called the Type of Service (TOS) bits. They effect the way packets are treated; the four bits are "Minimum Delay", "Maximum Throughput", "Maximum Reliability" and "Minimum Cost". Only one of these bits is allowed to be set. Rob van Nieuwkerk, the author of the ipchains TOS−mangling code, puts it as follows:

```
Especially the "Minimum Delay" is important for me. I switch
it on for "interactive" packets in my upstream (Linux)
router. I'm behind a 33k6 modem link. Linux prioritizes
packets in 3 queues. This way I get acceptable interactive
performance while doing bulk downloads at the same time.
```

The most common use is to set telnet & ftp control connections to "Minimum Delay" and FTP data to "Maximum Throughput". This would be done as follows, on your upstream router:

```
# iptables -A PREROUTING -t mangle -p tcp --sport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

Now, this only works for data going from your telnet foreign host to your local computer. The other way around appears to be done for you, ie, telnet, ssh & friends all set the TOS field on outgoing packets automatically.

Should you have a client that does not do this, you can always do it with netfilter. On your local box:

```
# iptables -A OUTPUT -t mangle -p tcp --dport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

14.5 Transparent web−caching using netfilter, iproute2, ipchains and squid

This section was sent in by reader Ram Narula from Internet for Education (Thailand).

The regular technique in accomplishing this in Linux is probably with use of ipchains AFTER making sure that the "outgoing" port 80(web) traffic gets routed through the server running squid.

There are 3 common methods to make sure "outgoing" port 80 traffic gets routed to the server running squid and 4th one is being introduced here.

Making the gateway router do it.

> If you can tell your gateway router to match packets that has outgoing destination port of 80 to be sent to the IP address of squid server.
> BUT
> This would put additional load on the router and some commercial routers might not even support this.

Using a Layer 4 switch.

> Layer 4 switches can handle this without any problem.
> BUT
> The cost for this equipment is usually very high. Typical layer 4 switch would normally cost more than a typical router+good linux server.

Using cache server as network's gateway.

You can force ALL traffic through cache server.
BUT
This is quite risky because Squid does utilize lots of cpu power which might result in slower over−all network performance or the server itself might crash and no one on the network will be able to access the internet if that occurs.

Linux+NetFilter router.

By using NetFilter another technique can be implemented which is using NetFilter for "mark"ing the packets with destination port 80 and using iproute2 to route the "mark"ed packets to the Squid server.

```
|----------------|
| Implementation |
|----------------|

 Addresses used
 10.0.0.1 naret (NetFilter server)
 10.0.0.2 silom (Squid server)
 10.0.0.3 donmuang (Router connected to the internet)
 10.0.0.4 kaosarn (other server on network)
 10.0.0.5 RAS
 10.0.0.0/24 main network
 10.0.0.0/19 total network

|---------------|
|Network diagram|
|---------------|

Internet
|
donmuang
|
------------hub/switch----------
|        |             |       |
naret   silom        kaosarn  RAS etc.
```

First, make all traffic pass through naret by making sure it is the default gateway except for silom. Silom's default gateway has to be donmuang (10.0.0.3) or this would create web traffic loop.

(all servers on my network had 10.0.0.1 as the default gateway which was the former IP address of donmuang router so what I did was changed the IP address of donmuang to 10.0.0.3 and gave naret ip address of 10.0.0.1)

```
Silom
-----
-setup squid and ipchains
```

Setup Squid server on silom, make sure it does support transparent caching/proxying, the default port is usually 3128, so all traffic for port 80 has to be redirected to port 3128 locally. This can be done by using ipchains with the following:

```
silom# ipchains -N allow1
silom# ipchains -A allow1 -p TCP -s 10.0.0.0/19 -d 0/0 80 -j REDIRECT 3128
silom# ipchains -I input -j allow1
```

Or, in netfilter lingo:

```
silom# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 3128
```

(note: you might have other entries as well)

For more information on setting Squid server please refer to Squid faq page on http://squid.nlanr.net).

Make sure ip forwarding is enabled on this server and the default gateway for this server is donmuang router (NOT naret).

```
Naret
-----
-setup iptables and iproute2
-disable icmp REDIRECT messages (if needed)
```

1. "Mark" packets of destination port 80 with value 2

   ```
   naret# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 80 \
    -j MARK --set-mark 2
   ```
2. Setup iproute2 so it will route packets with "mark" 2 to silom

   ```
   naret# echo 202 www.out >> /etc/iproute2/rt_tables
   naret# ip rule add fwmark 2 table www.out
   naret# ip route add default via 10.0.0.2 dev eth0 table www.out
   naret# ip route flush cache
   ```

If donmuang and naret is on the same subnet then naret should not send out icmp REDIRECT messages. In this case it is, so icmp REDIRECTs has to be disabled by:

```
naret# echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/default/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects
```

The setup is complete, check the configuration

```
On naret:

naret# iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
MARK       tcp  --  anywhere             anywhere           tcp dpt:www MARK set 0x2

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

naret# ip rule ls
0:      from all lookup local
32765:  from all fwmark        2 lookup www.out
32766:  from all lookup main
32767:  from all lookup default

naret# ip route list table www.out
default via 203.114.224.8 dev eth0

naret# ip route
10.0.0.1 dev eth0  scope link
10.0.0.0/24 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 10.0.0.3 dev eth0

(make sure silom belongs to one of the above lines, in this case
it's the line with 10.0.0.0/24)

|------|
|-DONE-|
|------|
```

Traffic flow diagram after implementation

```
|-----------------------------------------|
|Traffic flow diagram after implementation|
|-----------------------------------------|

INTERNET
/\
||
```

```
\/
-----------------donmuang router---------------------
/\                                      /\         ||
||                                      ||         ||
||                                      \/         ||
naret                                  silom       ||
*destination port 80 traffic=========>(cache)      ||
/\                                      ||         ||
||                                      \/         \/
\\===================================kaosarn, RAS, etc.
```

Note that the network is asymmetric as there is one extra hop on general outgoing path.

```
Here is run down for packet traversing the network from kaosarn
to and from the internet.

For web/http traffic:
kaosarn http request->naret->silom->donmuang->internet
http replies from internet->donmuang->silom->kaosarn

For non-web/http requests(eg. telnet):
kaosarn outgoing data->naret->donmuang->internet
incoming data from internet->donmuang->kaosarn
```

15.Advanced Linux Routing

This section is for all you people who either want to understand why the whole system works or have a configuration that's so bizarre that you need the low down to make it work.

This section is completely optional. It's quite possible that this section will be quite complex and really not intended for normal users. You have been warned.

FIXME: Decide what really need to go in here.

15.1 How does packet queueing really work?

This is the low−down on how the packet queueing system really works.

Lists the steps the kernel takes to classify a packet, etc...

FIXME: Write this.

15.2 Advanced uses of the packet queueing system

Go through Alexey's extremely tricky example involving the unused bits in the TOS field.

FIXME: Write this.

15.3 Other packet shaping systems

I'd like to include a brief description of other packet shaping systems in other operating systems and how they compare to the Linux one. Since Linux is one of the few OSes that has a completely original (non−BSD derived) TCP/IP stack, I think it would be useful to see how other people do it.

Unfortunately I have no experiene with other systems so cannot write this.

FIXME: Anyone? − Martijn

16.Dynamic routing − OSPF and BGP

Once your network starts to get really big, or you start to consider 'the internet' as your network, you need tools which dynamically route your data. Sites are often connected to each other with multiple links, and more are popping up all the time.

The Internet has mostly standardised on OSPF and BGP4 (rfc1771). Linux supports both, by way of `gated` and `zebra`

While currently not within the scope of this document, we would like to point you to the definitive works:

Overview:

Cisco Systems Designing large−scale IP internetworks

For OSPF:

Moy, John T. "OSPF. The anatomy of an Internet routing protocol" Addison Wesley. Reading, MA. 1998.

Halabi has also written a good guide to OSPF routing design, but this appears to have been dropped from the Cisco web site.

For BGP:

Halabi, Bassam "Internet routing architectures" Cisco Press (New Riders Publishing). Indianapolis, IN. 1997.

also

Cisco Systems

Using the Border Gateway Protocol for interdomain routing

Although the examples are Cisco−specific, they are remarkably similar to the configuration language in Zebra :−)

17.Further reading

http://snafu.freedom.org/linux2.2/iproute−notes.html

Contains lots of technical information, comments from the kernel

http://www.davin.ottawa.on.ca/ols/

Slides by Jamal Hadi, one of the authors of Linux traffic control

http://defiant.coinet.com/iproute2/ip−cref/

HTML version of Alexeys LaTeX documentation − explains part of iproute2 in great detail

http://www.aciri.org/floyd/cbq.html

Sally Floyd has a good page on CBQ, including her original papers. None of it is Linux specific, but it does a fair job discussing the theory and uses of CBQ. Very technical stuff, but good reading for those so inclined.

http://ceti.pl/~kravietz/cbq/NET4_tc.html

Yet another HOWTO, this time in Polish! You can copy/paste command lines however, they work just the same in every language. The author is cooperating with us and may soon author sections of this HOWTO.

Differentiated Services on Linux

Discussion on how to use Linux in a diffserv compliant environment. Pretty far removed from your everyday routing needs, but very interesting none the less. We may include a section on this at a later date.

IOS Committed Access Rate

>From the helpful folks of Cisco who have the laudable habit of putting their documentation online. Cisco syntax is different but the concepts are the same, except that we can do more and do it without routers the price of cars :−)

TCP/IP Illustrated, volume 1, W. Richard Stevens, ISBN 0−201−63346−9

Required reading if you truly want to understand TCP/IP. Entertaining as well.

18.Acknowledgements

It is our goal to list everybody who has contributed to this HOWTO, or helped us demystify how things work. While there are currently no plans for a Netfilter type scoreboard, we do like to recognise the people who are helping.

- Jamal Hadi <hadi%cyberus.ca>
- Nadeem Hasan <nhasan@usa.net>
- Philippe Latu <philippe.latu%linux−france.org>
- Jason Lunz <j@cc.gatech.edu>
- Alexey Mahotkin <alexm@formulabez.ru>
- Pawel Krawczyk <kravietz%alfa.ceti.pl>
- Wim van der Most
- Ram Narula <ram@princess1.net>
- Rusty Rusell (with apologies for always misspelling your name)
- Charles Tassell <ctassell%isn.net>
- Glen Turner <glen.turner%aarnet.edu.au>
- Song Wang <wsong@ece.uci.edu>