

Linux 2.4 Advanced Routing HOWTO

Table of Contents

<u>Linux 2.4 Advanced Routing HOWTO</u>	1
bert hubert <ahu@ds9a.nl> Gregory Maxwell <greg@linuxpower.cx> Martijn van Oosterhout <kleptog@cup	
1.Dedication	1
2.Introduction	1
3.Rules – routing policy database	1
4.GRE and other tunnels	1
5.Multicast routing	2
6.Using Class Based Queueing for bandwidth management	2
7.More queueing disciplines	2
8.Netfilter & iproute – marking packets	2
9.More classifiers	2
10.Kernel network parameters	2
11.Backbone applications of traffic control	2
12.Shaping Cookbook	3
13.Advanced Linux Routing	3
14.Dynamic routing – OSPF and BGP	3
15.Further reading	3
16.Acknowledgements	3
1.Dedication	3
2.Introduction	3
2.1 Disclaimer & License	4
2.2 Prior knowledge	4
2.3 What Linux can do for you	5
2.4 Housekeeping notes	5
2.5 Access, CVS & submitting updates	5
2.6 Layout of this document	6
3.Rules – routing policy database	6
3.1 Simple source routing	7
4.GRE and other tunnels	8
5.Multicast routing	8
6.Using Class Based Queueing for bandwidth management	8
6.1 What is queueing?	9
6.2 First attempt at bandwidth division	10
6.3 What to do with excess bandwidth	13
6.4 Class subdivisions	13
7.More queueing disciplines	13
7.1 pfifo fast	14
7.2 Stochastic Fairness Queueing	14
7.3 Token Bucket Filter	14
7.4 Random Early Detect	15
7.5 Ingress policer qdisc	15
8.Netfilter & iproute – marking packets	15
9.More classifiers	16
9.1 The "fw" classifier	18
9.2 The "u32" classifier	18
9.3 The "route" classifier	19
9.4 The "rsvp" classifier	19

Table of Contents

9.5 The "tcindex" classifier	19
10. Kernel network parameters	20
10.1 Reverse Path Filtering	20
10.2 Obscure settings	21
Generic ipv4	21
Per device settings	24
Neighbor policy	26
Routing settings	27
11. Backbone applications of traffic control	28
11.1 Router queues	28
12. Shaping Cookbook	29
12.1 Running multiple sites with different SLAs	30
12.2 Protecting your host from SYN floods	31
12.3 Ratelimit ICMP to prevent dDoS	32
12.4 Prioritising interactive traffic	33
13. Advanced Linux Routing	34
14. Dynamic routing – OSPF and BGP	34
15. Further reading	35
16. Acknowledgements	36

Linux 2.4 Advanced Routing HOWTO

bert hubert <ahu@ds9a.nl>

Gregory Maxwell <greg@linuxpower.cx>

Martijn van Oosterhout <kleptog@cupid.suninternet.com>
howto@ds9a.nl

v0.0.3 \$Date: 2000/04/01 13:27:51 \$

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

1. Dedicatation

2. Introduction

- [2.1 Disclaimer & License](#)
- [2.2 Prior knowledge](#)
- [2.3 What Linux can do for you](#)
- [2.4 Housekeeping notes](#)
- [2.5 Access, CVS & submitting updates](#)
- [2.6 Layout of this document](#)

3. Rules – routing policy database

- [3.1 Simple source routing](#)

4. GRE and other tunnels

5. Multicast routing

6. Using Class Based Queueing for bandwidth management

- [6.1 What is queueing?](#)
- [6.2 First attempt at bandwidth division](#)
- [6.3 What to do with excess bandwidth](#)
- [6.4 Class subdivisions](#)

7. More queueing disciplines

- [7.1 pfifo_fast](#)
- [7.2 Stochastic Fairness Queueing](#)
- [7.3 Token Bucket Filter](#)
- [7.4 Random Early Detect](#)
- [7.5 Ingress policer qdisc](#)

8. Netfilter & iproute – marking packets

9. More classifiers

- [9.1 The "fw" classifier](#)
- [9.2 The "u32" classifier](#)
- [9.3 The "route" classifier](#)
- [9.4 The "rsvp" classifier](#)
- [9.5 The "tcindex" classifier](#)

10. Kernel network parameters

- [10.1 Reverse Path Filtering](#)
- [10.2 Obscure settings](#)

11. Backbone applications of traffic control

- [11.1 Router queues](#)

12. Shaping Cookbook

- [12.1 Running multiple sites with different SLAs](#)
- [12.2 Protecting your host from SYN floods](#)
- [12.3 Rate limit ICMP to prevent dDoS](#)
- [12.4 Prioritising interactive traffic](#)

13. Advanced Linux Routing

14. Dynamic routing – OSPF and BGP

15. Further reading

16. 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 Russel
 - 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 Russel of (amongst other things) netfilter fame.

You can always reach us by writing the [HOWTO team](#).

2.1 Disclaimer & License

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

In short, if your STM-64 backbone breaks down and distributes pornography to your most esteemed customers – it's never our fault. Sorry.

Copyright (c) 2000 by bert hubert, Gregory Maxwell and Martijn van Oosterhout

Please freely copy and distribute (sell or give away) this document in any format. It's requested that corrections and/or comments be forwarded to the document maintainer. You may create a derivative work and distribute it provided that you:

1. Send your derivative work (in the most suitable format such as sgm1) 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 other references which might help learn you more:

[Rusty Russels 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 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. [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.

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 a 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-cfref` documentation.

3.1 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.

4. [GRE and other tunnels](#)

FIXME: waiting for our feature tunnel editor to finish his stuff

5. [Multicast routing](#)

FIXME: Editor Vacancy!

6. [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.

6.1 What is queueing?

With queueing we determine the order in which data is *sent*. It is important to realise this, we can only shape data that we transmit. How does 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'. 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 that normally, ACKs are used to determine speed

```
[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.

6.2 First attempt at bandwidth division

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

For example, we may have a 10 megabit internet 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, our 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.

FIXME: Double check with Alexey the the built in cell calculation is sufficient.

FIXME: With a 1500 mtu, the default cell is calculated same as the old example.

FIXME: I checked the sources (userspace and kernel), so we should be safe omitting it.

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
```

Linux 2.4 Advanced Routing HOWTO

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:

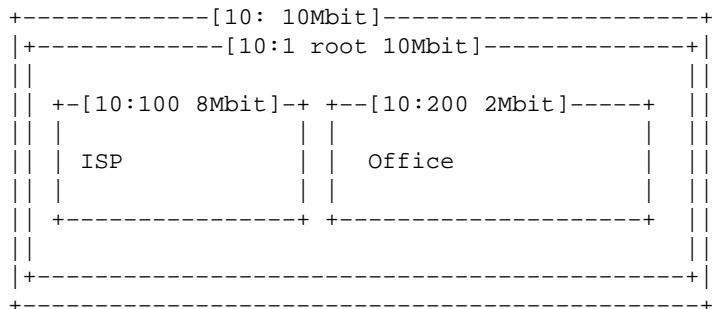
```
# tc class add dev eth0 parent 10:1 classid 10:100 cbq bandwidth 10Mbit rate \
  8Mbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 1000 \
  bounded
```

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:



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

Linux 2.4 Advanced Routing HOWTO

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 it 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 then 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
```

6.3 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.

6.4 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!

7. [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.

7.1 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.

7.2 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.

This is far too much work so SFQ keeps track of only a number of sessions by tracking things based on a hash. Two different sessions might end up in the same hash, which isn't very bad but should not be a permanent situation. Therefore the kernel perturbs the hash with a certain frequency, which can be specified on the `tc` command line.

7.3 Token Bucket Filter

This queue is very straightforward. Imagine a bucket, which holds a number of tokens. Tokens are added with a certain frequency, until the bucket fills up. By then, the bucket contains 'b' tokens.

Whenever packets arrive, they are stored. If there are more tokens than packets, these packets are sent out ('dequeued') immediately in a burst transfer.

If there are more packets than tokens, all packets for which there is a token are sent off, the rest have to wait for new tokens to arrive. So, if the size of a token is, say, 1000 octets, and we add 8 tokens per second, our eventual data rate is 64kilobit per second, excluding a certain 'burstiness' that we allow.

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

7.4 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.

7.5 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.

8. [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 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:

Linux 2.4 Advanced Routing HOWTO

```
# 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/?]
```

9. [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.

9.1 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:

FIXME: Equivalent iptables command?

```
# 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.

9.2 The "u32" classifier

The "u32" classifier is a filter that filters directly based on the contents of the packet. Thus it can filter based on source or destination addresses or ports. It can filter based on the TOS and other truly bizarre fields. It does this by taking a specification of the form [offset/mask/value] and applying that to all the packets. Fortunately you can use symbolic names much as with tcpdump.

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 1 u32 match ip dst HostA flowid 1:1
```

FIXME: What are the other possibilities?

That all there is to it.

9.3 The "route" classifier

FIXME: Doesn't work

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:

```
# ip route add HostA via Gateway flow 1:1
```

[Strangely, though I think I've done everything in the example, this doesn't seem to work for me. I get an error that goes:

Error: either "to" is duplicate, or "flow" is a garbage.

Someone who knows will have to comment on this.]

9.4 The "rsvp" classifier

FIXME: Fill me in

9.5 The "tcindex" classifier

FIXME: Fill me in

10. [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.

10.1 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

10.2 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 whether 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.

/proc/sys/net/ipv4/icmp_destunreach_rate

FIXME: fill this in

/proc/sys/net/ipv4/icmp_echo_ignore_all

FIXME: fill this in

/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

FIXME: fill this in

/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 is the famous cause of the 'Solaris middle star' in traceroutes. Limits number of ICMP Time Exceeded messages sent. FIXME: Units of these rates – either I'm stupid, or this just doesn't work

/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 queued packets will be rebranded 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 for hosts, on by default when configured as a router.

/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 you 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 only a single packet is missing – therefore helping fast recovery.

/proc/sys/net/ipv4/tcp_stdurg

FIXME: fill this in

/proc/sys/net/ipv4/tcp_syn_retries

FIXME: fill this in

/proc/sys/net/ipv4/tcp_synack_retries

FIXME: fill this in

/proc/sys/net/ipv4/tcp_timestamps

FIXME: fill this in

/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

FIXME: fill this in

/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/neighbor/DEV/anycast_delay

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/app_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/base_reachable_time

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/delay_first_probe_time

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/gc_stale_time

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/locktime

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/mcast_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/proxy_delay

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/proxy_qlen

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/retrans_time

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/DEV/ucast_solicit

FIXME: fill this in

/proc/sys/net/ipv4/neighbor/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

11. [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 than your ADSL modem at home.

11.1 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 than 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 than min.

Burst controls how the RED algorithm responds to bursts. Burst must be set large then min/avpkt . Experimentally, I've found $(\text{min}+\text{min}+\text{max})/(3*\text{avpkt})$ 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

12. [Shaping 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.

12.1 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, customer 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: 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?:
# 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?

12.2 Protecting your host from SYN floods

From Alexeys 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 sueful 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
```

12.3 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
```

12.4 Prioritising 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 prioritises
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
```

13. [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.

14. [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 :-)

15. [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/%7ekravietz/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.

16. [Acknowledgements](#)

It is our goal to list everybody who has contributed to this HOWTO, or helped us demistify 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>
 - Alexey Mahotkin <alexm@formulabez.ru>
 - Glen Turner <glen.turner%aarnet.edu.au>
-