# Linux Advanced Routing & Traffic Control HOWTO


## Bert Hubert

Netherlabs BV

<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>

Jasper Spaans

<jasper@spaans.ds9a.nl>



**Revision History**

Revision 1.1 2002−07−22

DocBook Edition



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

# Table of Contents

**Chapter 1. Dedication....1**

**Chapter 2. Introduction....2**
2.1. Disclaimer & License....2
2.2. Prior knowledge....2
2.3. What Linux can do for you....3
2.4. Housekeeping notes....3
2.5. Access, CVS & submitting updates....3
2.6. Mailing list....4
2.7. Layout of this document....4

**Chapter 3. Introduction to iproute2....5**
3.1. Why iproute2?....5
3.2. iproute2 tour....5
3.3. Prerequisites....5
3.4. Exploring your current configuration....6
3.4.1. ip shows us our links....6
3.4.2. ip shows us our IP addresses....6
3.4.3. ip shows us our routes....7
3.5. ARP....8

**Chapter 4. Rules – routing policy database....10**
4.1. Simple source policy routing....10
4.2. Routing for multiple uplinks/providers....11
4.2.1. Split access....12
4.2.2. Load balancing....13

**Chapter 5. GRE and other tunnels....14**
5.1. A few general remarks about tunnels:....14
5.2. IP in IP tunneling....14
5.3. GRE tunneling....15
5.3.1. IPv4 Tunneling....15
5.3.2. IPv6 Tunneling....16
5.4. Userland tunnels....17

**Chapter 6. IPv6 tunneling with Cisco and/or 6bone....18**
6.1. IPv6 Tunneling....18

**Chapter 7. IPsec: secure IP over the Internet....21**

**Chapter 8. Multicast routing....22**

**Chapter 9. Queueing Disciplines for Bandwidth Management....24**
9.1. Queues and Queueing Disciplines explained....24
9.2. Simple, classless Queueing Disciplines....25
9.2.1. pfifo_fast....25
9.2.2. Token Bucket Filter....27
9.2.3. Stochastic Fairness Queueing....29

# Table of Contents

**Chapter 9. Queueing Disciplines for Bandwidth Management**
9.3. Advice for when to use which queue....30
9.4. Terminology....30
9.5. Classful Queueing Disciplines....32
9.5.1. Flow within classful qdiscs & classes....32
9.5.2. The qdisc family: roots, handles, siblings and parents....33
9.5.3. The PRIO qdisc....34
9.5.4. The famous CBQ qdisc....36
9.5.5. Hierarchical Token Bucket....41
9.6. Classifying packets with filters....42
9.6.1. Some simple filtering examples....43
9.6.2. All the filtering commands you will normally need....44
9.7. The Intermediate queueing device (IMQ)....44
9.7.1. Sample configuration....45

**Chapter 10. Load sharing over multiple interfaces....47**
10.1. Caveats....48
10.2. Other possibilities....48

**Chapter 11. Netfilter & iproute – marking packets....49**

**Chapter 12. Advanced filters for (re−)classifying packets....50**
12.1. The u32 classifier....50
12.1.1. U32 selector....51
12.1.2. General selectors....52
12.1.3. Specific selectors....53
12.2. The route classifier....53
12.3. Policing filters....54
12.3.1. Ways to police....54
12.3.2. Overlimit actions....55
12.3.3. Examples....55
12.4. Hashing filters for very fast massive filtering....55

**Chapter 13. Kernel network parameters....58**
13.1. Reverse Path Filtering....58
13.2. Obscure settings....59
13.2.1. Generic ipv4....59
13.2.2. Per device settings....62
13.2.3. Neighbor policy....63
13.2.4. Routing settings....64

**Chapter 14. Advanced & less common queueing disciplines....66**
14.1. bfifo/pfifo....66
14.1.1. Parameters & usage....66
14.2. Clark−Shenker−Zhang algorithm (CSZ)....66
14.3. DSMARK....67
14.3.1. Introduction....67
14.3.2. What is Dsmark related to?....67

# Table of Contents

**Chapter 14. Advanced & less common queueing disciplines**
14.3.3. Differentiated Services guidelines....67
14.3.4. Working with Dsmark....68
14.3.5. How SCH_DSMARK works....68
14.3.6. TC_INDEX Filter....69
14.4. Ingress qdisc....71
14.4.1. Parameters & usage....71
14.5. Random Early Detection (RED)....71
14.6. Generic Random Early Detection....72
14.7. VC/ATM emulation....72
14.8. Weighted Round Robin (WRR)....72

**Chapter 15. Cookbook....74**
15.1. Running multiple sites with different SLAs....74
15.2. Protecting your host from SYN floods....75
15.3. Rate limit ICMP to prevent dDoS....76
15.4. Prioritizing interactive traffic....76
15.5. Transparent web−caching using netfilter, iproute2, ipchains and squid....77
15.5.1. Traffic flow diagram after implementation....80
15.6. Circumventing Path MTU Discovery issues with per route MTU settings....80
15.6.1. Solution....81
15.7. Circumventing Path MTU Discovery issues with MSS Clamping (for ADSL, cable, PPPoE & PPtP users)....82
15.8. The Ultimate Traffic Conditioner: Low Latency, Fast Up & Downloads....82
15.8.1. Why it doesn't work well by default....83
15.8.2. The actual script (CBQ)....84
15.8.3. The actual script (HTB)....86
15.9. Rate limiting a single host or netmask....87

**Chapter 16. Building bridges, and pseudo−bridges with Proxy ARP....89**
16.1. State of bridging and iptables....89
16.2. Bridging and shaping....89
16.3. Pseudo−bridges with Proxy−ARP....89
16.3.1. ARP & Proxy−ARP....90
16.3.2. Implementing it....90

**Chapter 17. Dynamic routing − OSPF and BGP....92**

**Chapter 18. Other possibilities....93**

**Chapter 19. Further reading....95**

**Chapter 20. Acknowledgements....96**

# Chapter 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

---

# Chapter 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 to the HOWTO team. However, please consider posting to the mailing list (see the relevant section) if you have questions which are not directly related to this HOWTO. We are no free helpdesk, but we often will answer questions asked on the list.

Before losing your way in this HOWTO, if all you want to do is simple traffic shaping, skip everything and head to the *Other possibilities* chapter, and read about CBQ.init.

## 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) 2002 by bert hubert, Gregory Maxwell, Martijn van Oosterhout, Remco van Mook, Paul B. Schroeder and others. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

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.

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.

Here are some other references which might help teach 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 teaches 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 is for 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 realize 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 in 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 to the world at large. This is good in a number of 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 docbook, docbook−utils, 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 not asked 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.

# Chapter 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 than the one 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"`.

You can also try here for the latest version.

Some parts of iproute require you to have certain kernel options enabled. It should also be noted that all releases of RedHat up to and including 6.2 come without most of the traffic control features in the default kernel.

RedHat 7.2 has everything in by default.

Also make sure that you have netlink support, should you choose to roll your own kernel. Iproute2 needs it.

---

## 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 to display our interfaces for us.

### 3.4.1. 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 one 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.

### 3.4.2. 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 there 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.

### 3.4.3. 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 that's used to locate his friend on the network, foo is able to communicate with bar until he (his arp cache) forgets where bar is (typically after 15 minutes on Unix).

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/neighbor 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.

---

# Chapter 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 is 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 policy 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 they'll end up using the low−end cable modem.

The 'fast' cable modem is known as 212.64.94.251 and is a 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 be 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
```

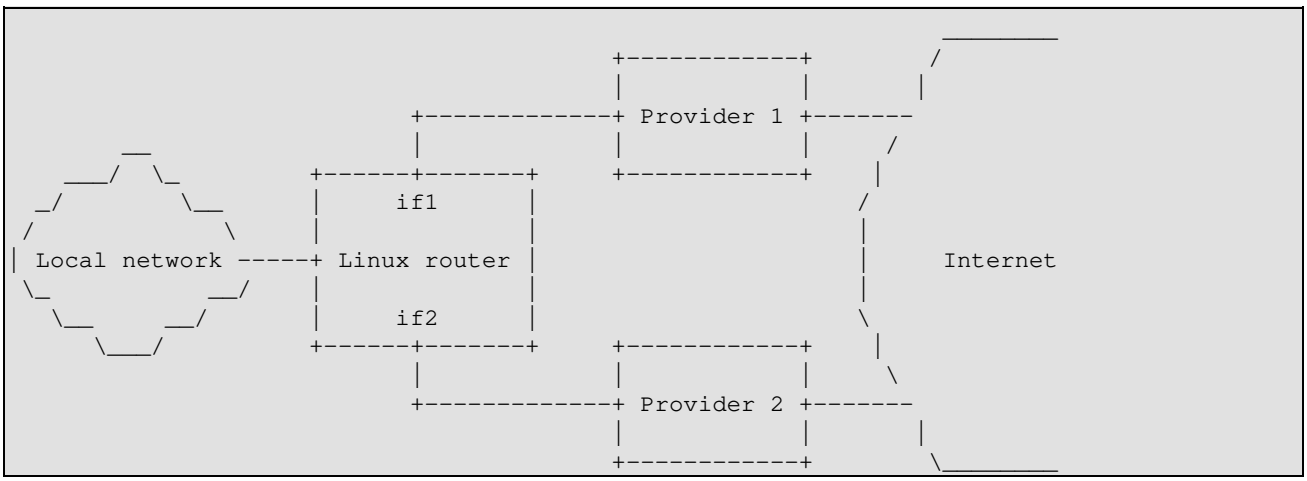
Now all that is left is to generate John's table, and flush the route cache:

```
# 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.

## 4.2. Routing for multiple uplinks/providers

A common configuration is the following, in which there are two providers that connect a local network (or even a single machine) to the big Internet.




There are usually two questions given this setup.

### 4.2.1. Split access

The first is how to route answers to packets coming in over a particular provider, say Provider 1, back out again over that same provider.

Let us first set some symbolical names. Let **$IF1** be the name of the first interface (if1 in the picture above) and **$IF2** the name of the second interface. Then let **$IP1** be the IP address associated with **$IF1** and **$IP2** the IP address associated with **$IF2**. Next, let **$P1** be the IP address of the gateway at Provider 1, and **$P2** the IP address of the gateway at provider 2. Finally, let **$P1_NET** be the IP network **$P1** is in, and **$P2_NET** the IP network **$P2** is in.

One creates two additional routing tables, say **T1** and **T2**. These are added in /etc/iproute2/rt_tables. Then you set up routing in these tables as follows:

```
ip route add $P1_NET dev $IF1 src $IP1 table T1
ip route add default via $P1 table T1
ip route add $P2_NET dev $IF2 src $IP2 table T2
ip route add default via $P2 table T2
```

Nothing spectacular, just build a route to the gateway and build a default route via that gateway, as you would do in the case of a single upstream provider, but put the routes in a separate table per provider. Note that the network route suffices, as it tells you how to find any host in that network, which includes the gateway, as specified above.

Next you set up the main routing table. It is a good idea to route things to the direct neighbour through the interface connected to that neighbour. Note the `src' arguments, they make sure the right outgoing IP address is chosen.

```
ip route add $P1_NET dev $IF1 src $IP1
ip route add $P2_NET dev $IF2 src $IP2
```

Then, your preference for default route:

```
ip route add default via $P1
```

Next, you set up the routing rules. These actually choose what routing table to route with. You want to make sure that you route out a given interface if you already have the corresponding source address:

```
ip rule add from $IP1 table T1
ip rule add from $IP2 table T2
```

This set of commands makes sure all answers to traffic coming in on a particular interface get answered from that interface.

Now, this is just the very basic setup. It will work for all processes running on the router itself, and for the local network, if it is masqueraded. If it is not, then you either have IP space from both providers or you are going to want to masquerade to one of the two providers. In both cases you will want to add rules selecting which provider to route out from based on the IP address of the machine in the local network.

### 4.2.2. Load balancing

The second question is how to balance traffic going out over the two providers. This is actually not hard if you already have set up split access as above.

Instead of choosing one of the two providers as your default route, you now set up the default route to be a multipath route. In the default kernel this will balance routes over the two providers. It is done as follows (once more building on the example in the section on split−access):

```
ip route add default scope global nexthop via $P1 dev $IF1 weight 1 \
nexthop via $P2 dev $IF2 weight 1
```

This will balance the routes over both providers. The **weight** parameters can be tweaked to favor one provider over the other.

Note that balancing will not be perfect, as it is route based, and routes are cached. This means that routes to often−used sites will always be over the same provider.

Furthermore, if you really want to do this, you probably also want to look at Julian Anastasov's patches at http://www.linuxvirtualserver.org/~julian/#routes , Julian's route patch page. They will make things nicer to work with.

---

# Chapter 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.

## 5.3.1. 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 link set netb up
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).

The second line enables the device.

In the third 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 fourth 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 link set neta up
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 tunnel on router A:

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

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

## 5.3.2. IPv6 Tunneling

See Section 6 for a short bit about IPv6 Addresses.

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 is 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.

# Chapter 6. IPv6 tunneling with Cisco and/or 6bone

By Marco Davids <marco@sara.nl>

NOTE to maintainer:

As far as I am concerned, this IPv6−IPv4 tunneling is not per definition GRE tunneling. You could tunnel IPv6 over IPv4 by means of GRE tunnel devices (GRE tunnels ANY to IPv4), but the device used here ("sit") only tunnels IPv6 over IPv4 and is therefore something different.

## 6.1. IPv6 Tunneling

This is another application of the tunneling capabilities of Linux. It is popular among the IPv6 early adopters, or pioneers if you like. The 'hands−on' example described below is certainly not the only way to do IPv6 tunneling. However, it is the method that is often used to tunnel between Linux and a Cisco IPv6 capable router and experience tells us that this is just the thing many people are after. Ten to one this applies to you too ;−)

A short bit about IPv6 addresses:

IPv6 addresses are, compared to IPv4 addresses, really big: 128 bits against 32 bits. And this provides us just with the thing we need: many, many IP−addresses: 340,282,266,920,938,463,463,374,607,431,768,211,465 to be precise. Apart from this, IPv6 (or IPng, for IP Next Generation) is supposed to provide for smaller routing tables on the Internet's backbone routers, simpler configuration of equipment, better security at the IP level and better support for QoS.

An example: 2002:836b:9820:0000:0000:0000:836b:9886

Writing down IPv6 addresses can be quite a burden. Therefore, to make life easier there are some 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.

The address 2002:836b:9820:0000:0000:0000:836b:9886 can be written down as 2002:836b:9820::836b:9886, which is somewhat friendlier.

Another example, the address 3ffe:0000:0000:0000:0000:0020:34A1:F32C can be written down as 3ffe::20:34A1:F32C, which is a lot shorter.

IPv6 is intended to be the successor of the current IPv4. Because it is relatively new technology, there is no worldwide native IPv6 network yet. To be able to move forward swiftly, the 6bone was introduced.

Native IPv6 networks are connected to each other by encapsulating the IPv6 protocol in IPv4 packets and sending them over the existing IPv4 infrastructure from one IPv6 site to another.

That is precisely where the tunnel steps in.

To be able to use IPv6, we should have a kernel that supports it. There are many good documents on how to achieve this. But it all comes down to a few steps:

- Get yourself a recent Linux distribution, with suitable glibc.
- Then get yourself an up−to−date kernel source.

If you are all set, then you can go ahead and compile an IPv6 capable kernel:

- Go to /usr/src/linux and type:
- make menuconfig
- Choose "Networking Options"
- Select "The IPv6 protocol", "IPv6: enable EUI−64 token format", "IPv6: disable provider based addresses"

HINT: Don't go for the 'module' option. Often this won't work well.

In other words, compile IPv6 as 'built−in' in your kernel. You can then save your config like usual and go ahead with compiling the kernel.

HINT: Before doing so, consider editing the Makefile: EXTRAVERSION = −x ; −−> ; EXTRAVERSION = −x−IPv6

There is a lot of good documentation about compiling and installing a kernel, however this document is about something else. If you run into problems at this stage, go and look for documentation about compiling a Linux kernel according to your own specifications.

The file /usr/src/linux/README might be a good start. After you accomplished all this, and rebooted with your brand new kernel, you might want to issue an '/sbin/ifconfig −a' and notice the brand new 'sit0−device'. SIT stands for Simple Internet Transition. You may give yourself a compliment; you are now one major step closer to IP, the Next Generation ;−)

Now on to the next step. You want to connect your host, or maybe even your entire LAN to another IPv6 capable network. This might be the "6bone" that is setup especially for this particular purpose.

Let's assume that you have the following IPv6 network: 3ffe:604:6:8::/64 and you want to connect it to 6bone, or a friend. Please note that the /64 subnet notation works just like with regular IP addresses.

Your IPv4 address is 145.100.24.181 and the 6bone router has IPv4 address 145.100.1.5

```
# ip tunnel add sixbone mode sit remote 145.100.1.5 [local 145.100.24.181 ttl 255]
# ip link set sixbone up
# ip addr add 3FFE:604:6:7::2/126 dev sixbone
# ip route add 3ffe::0/16 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. If the particular machine you run this on is your IPv6 gateway, then consider adding the following lines:

```
# echo 1 >/proc/sys/net/ipv6/conf/all/forwarding
# /usr/local/sbin/radvd
```

The latter, radvd is –like zebra– a router advertisement daemon, to support IPv6's autoconfiguration features. Search for it with your favourite search–engine if you like. You can check things like this:

```
# /sbin/ip -f inet6 addr
```

If you happen to have radvd running on your IPv6 gateway and boot your IPv6 capable Linux on a machine on your local LAN, you would be able to enjoy the benefits of IPv6 autoconfiguration:

```
# /sbin/ip -f inet6 addr
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue inet6 ::1/128 scope host

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
inet6 3ffe:604:6:8:5054:4cff:fe01:e3d6/64 scope global dynamic
valid_lft forever preferred_lft 604646sec inet6 fe80::5054:4cff:fe01:e3d6/10
scope link
```

You could go ahead and configure your bind for IPv6 addresses. The A type has an equivalent for IPv6: AAAA. The in–addr.arpa's equivalent is: ip6.int. There's a lot of information available on this topic.

There is an increasing number of IPv6–aware applications available, including secure shell, telnet, inetd, Mozilla the browser, Apache the webserver and a lot of others. But this is all outside the scope of this Routing document ;–)

On the Cisco side the configuration would be something like this:

```
!
interface Tunnel1
description IPv6 tunnel
no ip address
no ip directed-broadcast
ipv6 enable
ipv6 address 3FFE:604:6:7::1/126
tunnel source Serial0
tunnel destination 145.100.24.181
tunnel mode ipv6ip
!
ipv6 route 3FFE:604:6:8::/64 Tunnel1
```

But if you don't have a Cisco at your disposal, try one of the many IPv6 tunnel brokers available on the Internet. They are willing to configure their Cisco with an extra tunnel for you. Mostly by means of a friendly web interface. Search for "ipv6 tunnel broker" on your favourite search engine.

---

# Chapter 7. IPsec: secure IP over the Internet

FIXME: editor vacancy. In the meantime, see: The FreeS/WAN project. Another IPSec implementation for Linux is Cerberus, by NIST. However, their web pages have not been updated in over a year, and their version tended to trail well behind the current Linux kernel. USAGI, an alternative IPv6 implementation for Linux, also includes an IPSec implementation, but that might only be for IPv6.

---

# Chapter 8. Multicast routing

FIXME: Editor Vacancy!

The Multicast−HOWTO is ancient (relatively−speaking) and may be inaccurate or misleading in places, for that reason.

Before you can do any multicast routing, you need to configure the Linux kernel to support the type of multicast routing you want to do. This, in turn, requires you to decide what type of multicast routing you expect to be using. There are essentially four "common" types − DVMRP (the Multicast version of the RIP unicast protocol), MOSPF (the same, but for OSPF), PIM−SM ("Protocol Independent Multicasting − Sparse Mode", which assumes that users of any multicast group are spread out, rather than clumped) and PIM−DM (the same, but "Dense Mode", which assumes that there will be significant clumps of users of the same multicast group).

In the Linux kernel, you will notice that these options don't appear. This is because the protocol itself is handled by a routing application, such as Zebra, mrouted, or pimd. However, you still have to have a good idea of which you're going to use, to select the right options in the kernel.

For all multicast routing, you will definitely need to enable "multicasting" and "multicast routing". For DVMRP and MOSPF, this is sufficient. If you are going to use PIM, you must also enable PIMv1 or PIMv2, depending on whether the network you are connecting to uses version 1 or 2 of the PIM protocol.

Once you have all that sorted out, and your new Linux kernel compiled, you will see that the IP protocols listed, at boot time, now include IGMP. This is a protocol for managing multicast groups. At the time of writing, Linux supports IGMP versions 1 and 2 only, although version 3 does exist and has been documented. This doesn't really affect us that much, as IGMPv3 is still new enough that the extra capabilities of IGMPv3 aren't going to be that much use. Because IGMP deals with groups, only the features present in the simplest version of IGMP over the entire group are going to be used. For the most part, that will be IGMPv2, although IGMPv1 is sill going to be encountered.

So far, so good. We've enabled multicasting. Now, we have to tell the Linux kernel to actually do something with it, so we can start routing. This means adding the Multicast virtual network to the router table:

```
ip route add 224.0.0.0/4 dev eth0
```

(Assuming, of course, that you're multicasting over eth0! Substitute the device of your choice, for this.)

Now, tell Linux to forward packets...

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

At this point, you may be wondering if this is ever going to do anything. So, to test our connection, we ping the default group, 224.0.0.1, to see if anyone is alive. All machines on your LAN with multicasting enabled *should* respond, but nothing else. You'll notice that none of the machines that respond have an IP address of 224.0.0.1. What a surprise! :) This is a group address (a "broadcast" to subscribers), and all members of the group will respond with their own address, not the group address.

```
ping −c 2 224.0.0.1
```

At this point, you're ready to do actual multicast routing. Well, assuming that you have two networks to route between.

(To Be Continued!)

---

# Chapter 9. Queueing Disciplines for Bandwidth Management

Now, when I discovered this, it *really* blew me away. Linux 2.2/2.4 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.

Just to prevent confusion, **tc** uses the following rules for bandwith specification:

```
mbps = 1024 kbps = 1024 * 1024 bps => byte/s
mbit = 1024 kbit => kilo bit/s.
mb = 1024 kb = 1024 * 1024 b => byte
mbit = 1024 kbit => kilo bit.
```

Internally, the number is stored in bps and b.

But when **tc** prints the rate, it uses following :

```
1Mbit = 1024 Kbit = 1024 * 1024 bps => bit/s
```

## 9.1. Queues and Queueing Disciplines explained

With queueing we determine the way in which data is *SENT*. It is important to realise that we can only shape data that we transmit.

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. In fact it is a bit smarter than this, but more about that later.

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 :–)

If you have a router and wish to prevent certain hosts within your network from downloading too fast, you need to do your shaping on the *inner* interface of your router, the one that sends data to your own computers.

You also have to be sure you are controlling the bottleneck of the link. If you have a 100Mbit NIC and you have a router that has a 256kbit link, you have to make sure you are not sending more data than your router can handle. Otherwise, it will be the router who is controlling the link and shaping the available bandwith. We need to 'own the queue' so to speak, and be the slowest link in the chain. Luckily this is easily possible.

# 9.2. Simple, classless Queueing Disciplines

As said, with queueing disciplines, we change the way data is sent. Classless queueing disciplines are those that, by and large accept data and only reschedule, delay or drop it.

These can be used to shape traffic for an entire interface, without any subdivisions. It is vital that you understand this part of queueing before we go on the the classful qdisc−containing−qdiscs!

By far the most widely used discipline is the pfifo_fast qdisc − 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.

## 9.2.1. 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.

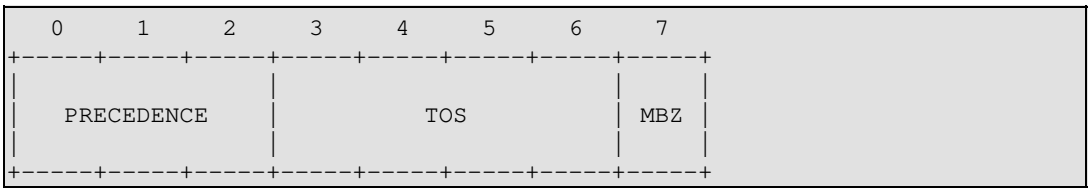
The kernel honors the so called Type of Service flag of packets, and takes care to insert 'minimum delay' packets in band 0.

Do not confuse this classless simple qdisc with the classful PRIO one! Although they behave similarly, pfifo_fast is classless and you cannot add other qdiscs to it with the tc command.

### 9.2.1.1. Parameters & usage

You can't configure the pfifo_fast qdisc as it is the hardwired default. This is how it is configured by default:

*priomap*
Determines how packet priorities, as assigned by the kernel, map to bands. Mapping occurs based on the TOS octet of the packet, which looks like this:




The four TOS bits (the 'TOS field') are defined as:

| Binary | Decimcal | Meaning |
|---|---|---|
| 1000 | 8 | Minimize delay (md) |
| 0100 | 4 | Maximize throughput (mt) |
| 0010 | 2 | Maximize reliability (mr) |
| 0001 | 1 | Minimize monetary cost (mmc) |
| 0000 | 0 | Normal Service |

As there is 1 bit to the right of these four bits, the actual value of the TOS field is double the value of the TOS bits. Tcpdump −v −v shows you the value of the entire TOS field, not just the four bits. It is

the value you see in the first column of this table:

```
TOS     Bits  Means                    Linux Priority    Band
------------------------------------------------------------
0x0     0     Normal Service           0 Best Effort     1
0x2     1     Minimize Monetary Cost   1 Filler          2
0x4     2     Maximize Reliability     0 Best Effort     1
0x6     3     mmc+mr                   0 Best Effort     1
0x8     4     Maximize Throughput      2 Bulk            2
0xa     5     mmc+mt                   2 Bulk            2
0xc     6     mr+mt                    2 Bulk            2
0xe     7     mmc+mr+mt                2 Bulk            2
0x10    8     Minimize Delay           6 Interactive     0
0x12    9     mmc+md                   6 Interactive     0
0x14    10    mr+md                    6 Interactive     0
0x16    11    mmc+mr+md                6 Interactive     0
0x18    12    mt+md                    4 Int. Bulk       1
0x1a    13    mmc+mt+md                4 Int. Bulk       1
0x1c    14    mr+mt+md                 4 Int. Bulk       1
0x1e    15    mmc+mr+mt+md             4 Int. Bulk       1
```

Lots of numbers. The second column contains the value of the relevant four TOS bits, followed by their translated meaning. For example, 15 stands for a packet wanting Minimal Monetary Cost, Maximum Reliability, Maximum Throughput AND Minimum Delay. I would call this a 'Dutch Packet'.

The fourth column lists the way the Linux kernel interprets the TOS bits, by showing to which Priority they are mapped.

The last column shows the result of the default priomap. On the command line, the default priomap looks like this:

```
1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

This means that priority 4, for example, gets mapped to band number 1. The priomap also allows you to list higher priorities (> 7) which do not correspond to TOS mappings, but which are set by other means.

This table from RFC 1349 (read it for more details) tells you how applications might very well set their TOS bits:

```
TELNET                  1000           (minimize delay)
FTP
        Control         1000           (minimize delay)
        Data            0100           (maximize throughput)

TFTP                    1000           (minimize delay)

SMTP
        Command phase   1000           (minimize delay)
        DATA phase      0100           (maximize throughput)

Domain Name Service
        UDP Query       1000           (minimize delay)
        TCP Query       0000
        Zone Transfer   0100           (maximize throughput)
```

```
NNTP                    0001            (minimize monetary cost)

ICMP
        Errors          0000
        Requests        0000 (mostly)
        Responses       <same as request> (mostly)
```

*txqueuelen*

The length of this queue is gleaned from the interface configuration, which you can see and set with ifconfig and ip. To set the queue length to 10, execute: ifconfig eth0 txqueuelen 10

You can't set this parameter with tc!

## 9.2.2. Token Bucket Filter

The Token Bucket Filter (TBF) is a simple qdisc that only passes packets arriving at a rate which is not exceeding some administratively set rate, but with the possibility to allow short bursts in excess of this rate.

TBF is very precise, network– and processor friendly. It should be your first choice if you simply want to slow an interface down!

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

Each arriving token collects one incoming data packet from the data 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 in TBF at a rate that's *equal* to the rate of incoming tokens. In this case each incoming packet has its matching token and passes the queue without delay.
- The data arrives in TBF at a rate that's *smaller* than the token rate. Only a part of the tokens are deleted at output of each data packet that's sent out the queue, so the tokens accumulate, up to the bucket size. The unused tokens can then be used to send data a a speed that's exceeding the standard token rate, in case short data bursts occur.
- The data arrives in TBF at a rate *bigger* than the token rate. This means that the bucket will soon be devoid of tokens, which causes the TBF to throttle itself for a while. This is called an 'overlimit situation'. If packets keep coming in, packets will start to get dropped.

The last scenario is very important, because it allows to administratively shape the bandwidth available to data that's passing the filter.

The accumulation of tokens allows a short burst of overlimit data to be still passed without loss, but any lasting overload will cause packets to be constantly delayed, and then dropped.

Please note that in the actual implementation, tokens correspond to bytes, not packets.

### 9.2.2.1. Parameters & usage

Even though you will probably not need to change them, tbf has some knobs available. First the parameters that are always available:

*limit or latency*

Limit is the number of bytes that can be queued waiting for tokens to become available. You can also specify this the other way around by setting the latency parameter, which specifies the maximum amount of time a packet can sit in the TBF. The latter calculation takes into account the size of the bucket, the rate and possibly the peakrate (if set).

*burst/buffer/maxburst*

Size of the bucket, in bytes. This is the maximum amount of bytes that tokens can be available for instantaneously. In general, larger shaping rates require a larger buffer. For 10mbit/s on Intel, you need at least 10kbyte buffer if you want to reach your configured rate!

If your buffer is too small, packets may be dropped because more tokens arrive per timer tick than fit in your bucket.

*mpu*

A zero−sized packet does not use zero bandwidth. For ethernet, no packet uses less than 64 bytes. The Minimum Packet Unit determines the minimal token usage for a packet.

*rate*

The speedknob. See remarks above about limits!

If the bucket contains tokens and is allowed to empty, by default it does so at infinite speed. If this is unacceptable, use the following parameters:

*peakrate*

If tokens are available, and packets arrive, they are sent out immediately by default, at 'lightspeed' so to speak. That may not be what you want, especially if you have a large bucket.

The peakrate can be used to specify how quickly the bucket is allowed to be depleted. If doing everything by the book, this is achieved by releasing a packet, and then wait just long enough, and release the next. We calculated our waits so we send just at peakrate.

However, due to de default 10ms timer resolution of Unix, with 10.000 bits average packets, we are limited to 1mbit/s of peakrate!

*mtu/minburst*

The 1mbit/s peakrate is not very useful if your regular rate is more than that. A higher peakrate is possible by sending out more packets per timertick, which effectively means that we create a second bucket!

This second bucket defaults to a single packet, which is not a bucket at all.

To calculate the maximum possible peakrate, multiply the configured mtu by 100 (or more correctly, HZ, which is 100 on Intel, 1024 on Alpha).

---

#### 9.2.2.2. Sample configuration

A simple but *very* useful configuration is this:

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

Ok, why is this useful? If you have a networking device with a large queue, like a DSL modem or a cable modem, and you talk to it over a fast device, like over an ethernet interface, you will find that uploading absolutely destroys interactivity.

This is because uploading will fill the queue in the modem, which is probably *huge* because this helps actually achieving good data throughput uploading. But this is not what you want, you want to have the queue not too big so interactivity remains and you can still do other stuff while sending data.

The line above slows down sending to a rate that does not lead to a queue in the modem – the queue will be in Linux, where we can control it to a limited size.

Change 220kbit to your uplink's *actual* speed, minus a few percent. If you have a really fast modem, raise 'burst' a bit.

## 9.2.3. Stochastic Fairness Queueing

Stochastic Fairness Queueing (SFQ) is a simple implementation of the 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), which mostly corresponds to a TCP session or a UDP stream. Traffic is divided into a pretty large number of FIFO queues, one for each conversation. Traffic is then sent in a round robin fashion, giving each session the chance to send data in turn.

This leads to very fair behaviour and disallows any single conversation from drowning out the rest. SFQ is called 'Stochastic' because it doesn't really allocate a queue for each session, it has an algorithm which divides traffic over a limited number of queues using a hashing algorithm.

Because of the hash, multiple sessions might end up in the same bucket, which would halve each session's chance of sending a packet, thus halving the effective speed available. To prevent this situation from becoming noticeable, SFQ changes its hashing algorithm quite often so that any two colliding sessions will only do so for a small number of seconds.

It is important to note that SFQ is only useful in case your actual outgoing interface is really full! If it isn't then there will be no queue on your linux machine and hence no effect. Later on we will describe how to combine SFQ with other qdiscs to get a best–of–both worlds situation.

Specifically, setting SFQ on the ethernet interface heading to your cable modem or DSL router is pointless without further shaping!

### 9.2.3.1. Parameters & usage

The SFQ is pretty much self tuning:

*perturb*
: Reconfigure hashing once this many seconds. If unset, hash will never be reconfigured. Not recommended. 10 seconds is probably a good value.

*quantum*
: Amount of bytes a stream is allowed to dequeue before the next queue gets a turn. Defaults to 1 maximum sized packet (MTU–sized). Do not set below the MTU!

#### 9.2.3.2. Sample configuration

If you have a device which has identical link speed and actual available rate, like a phone modem, this configuration will help promote fairness:

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls
qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
 Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

The number 800c: is the automatically assigned handle number, limit means that 128 packets can wait in this queue. There are 1024 hashbuckets available for accounting, of which 128 can be active at a time (no more packets fit in the queue!) Once every 10 seconds, the hashes are reconfigured.

## 9.3. Advice for when to use which queue

Summarizing, these are the simple queues that actually manage traffic by reordering, slowing or dropping packets.

The following tips may help in choosing which queue to use. It mentions some qdiscs described in the *Chapter 14* chapter.

- To purely slow down outgoing traffic, use the Token Bucket Filter. Works up to huge bandwidths, if you scale the bucket.
- If your link is truly full and you want to make sure that no single session can dominate your outgoing bandwidth, use Stochastical Fairness Queueing.
- If you have a big backbone and know what you are doing, consider Random Early Drop (see Advanced chapter).
- To 'shape' incoming traffic which you are not forwarding, use the Ingress Policer. Incoming shaping is called 'policing', by the way, not 'shaping'.
- If you *are* forwarding it, use a TBF on the interface you are forwarding the data to. Unless you want to shape traffic that may go out over several interfaces, in which case the only common factor is the incoming interface. In that case use the Ingress Policer.
- If you don't want to shape, but only want to see if your interface is so loaded that it has to queue, use the pfifo queue (not pfifo_fast). It lacks internal bands but does account the size of its backlog.
- Finally – you can also do "social shaping". You may not always be able to use technology to achieve what you want. Users experience technical constraints as hostile. A kind word may also help with getting your bandwidth to be divided right!

## 9.4. Terminology

To properly understand more complicated configurations it is necessary to explain a few concepts first. Because of the complexity and he relative youth of the subject, a lot of different words are used when people in fact mean the same thing.

The following is loosely based on `draft-ietf-diffserv-model-06.txt`, *An Informal Management Model for Diffserv Routers*. It can currently be found at http://www.ietf.org/internet-drafts/draft-ietf-diffserv-model-06.txt.

Read it for the strict definitions of the terms used.

*Queueing Discipline*
: An algorithm that manages the queue of a device, either incoming (ingress) or outgoing (egress).

*Classless qdisc*
: A qdisc with no configurable internal subdivisions.

*Classful qdisc*
: A classful qdisc contains multiple classes. Each of these classes contains a further qdisc, which may again be classful, but need not be. According to the strict definition, pfifo_fast *is* classful, because it contains three bands which are, in fact, classes. However, from the user's configuration perspective, it is classless as the classes can't be touched with the tc tool.

*Classes*
: A classful qdisc may have many classes, which each are internal to the qdisc. Each of these classes may contain a real qdisc.

*Classifier*
: Each classful qdisc needs to determine to which class it needs to send a packet. This is done using the classifier.

*Filter*
: Classification can be performed using filters. A filter contains a number of conditions which if matched, make the filter match.

*Scheduling*
: A qdisc may, with the help of a classifier, decide that some packets need to go out earlier than others. This process is called Scheduling, and is performed for example by the pfifo_fast qdisc mentioned earlier. Scheduling is also called 'reordering', but this is confusing.

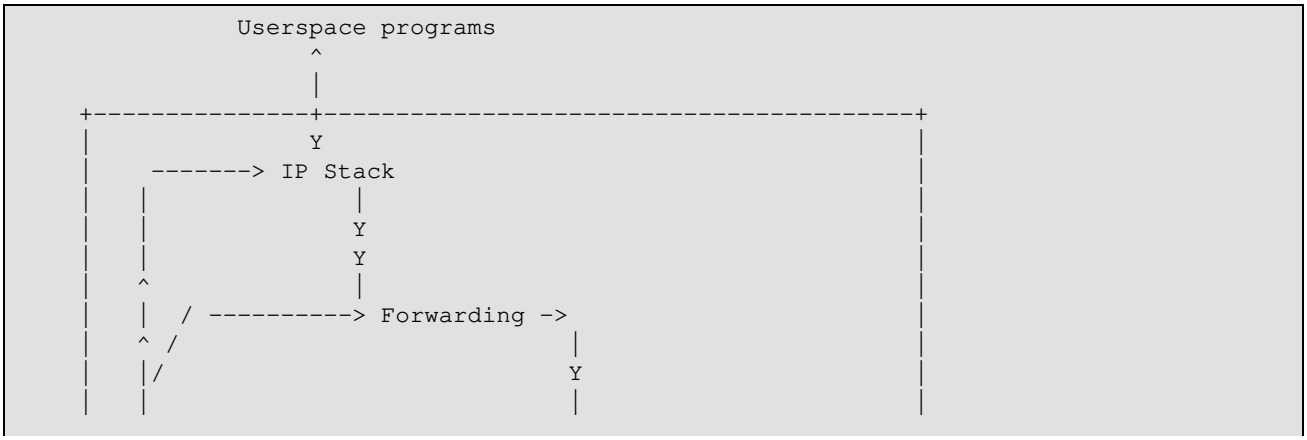
*Shaping*
: The process of delaying packets before they go out to make traffic confirm to a configured maximum rate. Shaping is performed on egress. Colloquially, dropping packets to slow traffic down is also often called Shaping.

*Policing*
: Delaying or dropping packets in order to make traffic stay below a configured bandwidth. In Linux, policing can only drop a packet and not delay it – there is no 'ingress queue'.

*Work−Conserving*
: A work−conserving qdisc always delivers a packet if one is available. In other words, it never delays a packet if the network adaptor is ready to send one (in the case of an egress qdisc).

*non−Work−Conserving*
: Some queues, like for example the Token Bucket Filter, may need to hold on to a packet for a certain time in order to limit the bandwidth. This means that they sometimes refuse to give up a packet, even though they have one available.

Now that we have our terminology straight, let's see where all these things are.



```
    |   ^                              Y          /-qdisc1-\   |
    |   |                             Egress     /--qdisc2--\  |
 --->->Ingress                        Classifier ---qdisc3---- | ->
    |   Qdisc                                    \__qdisc4__/  |
    |                                             \-qdiscN_/   |
    |                                                          |
    +----------------------------------------------------------+
```

Thanks to Jamal Hadi Salim for this ASCII representation.

The big block represents the kernel. The leftmost arrow represents traffic entering your machine from the network. It is then fed to the Ingress Qdisc which may apply Filters to a packet, and decide to drop it. This is called 'Policing'.

This happens at a very early stage, before it has seen a lot of the kernel. It is therefore a very good place to drop traffic very early, without consuming a lot of CPU power.

If the packet is allowed to continue, it may be destined for a local application, in which case it enters the IP stack in order to be processed, and handed over to a userspace program. The packet may also be forwarded without entering an application, in which case it is destined for egress. Userspace programs may also deliver data, which is then examined and forwarded to the Egress Classifier.

There it is investigated and enqueued to any of a number of qdiscs. In the unconfigured default case, there is only one egress qdisc installed, the pfifo_fast, which always receives the packet. This is called 'enqueueing'.

The packet now sits in the qdisc, waiting for the kernel to ask for it for transmission over the network interface. This is called 'dequeueing'.

This picture also holds in case there is only one network adaptor – the arrows entering and leaving the kernel should not be taken too literally. Each network adaptor has both ingress and egress hooks.

## 9.5. Classful Queueing Disciplines

Classful qdiscs are very useful if you have different kinds of traffic which should have differing treatment. One of the classful qdiscs is called 'CBQ' , 'Class Based Queueing' and it is so widely mentioned that people identify queueing with classes solely with CBQ, but this is not the case.

CBQ is merely the oldest kid on the block – and also the most complex one. It may not always do what you want. This may come as something of a shock to many who fell for the 'sendmail effect', which teaches us that any complex technology which doesn't come with documentation must be the best available.

More about CBQ and its alternatives shortly.

### 9.5.1. Flow within classful qdiscs & classes

When traffic enters a classful qdisc, it needs to be sent to any of the classes within – it needs to be 'classified'. To determine what to do with a packet, the so called 'filters' are consulted. It is important to know that the filters are called from within a qdisc, and not the other way around!

The filters attached to that qdisc then return with a decision, and the qdisc uses this to enqueue the packet into one of the classes. Each subclass may try other filters to see if further instructions apply. If not, the class

enqueues the packet to the qdisc it contains.

Besides containing other qdiscs, most classful qdiscs also perform shaping. This is useful to perform both packet scheduling (with SFQ, for example) and rate control. You need this in cases where you have a high speed interface (for example, ethernet) to a slower device (a cable modem).

If you were only to run SFQ, nothing would happen, as packets enter & leave your router without delay: the output interface is far faster than your actual link speed. There is no queue to schedule then.

## 9.5.2. The qdisc family: roots, handles, siblings and parents

Each interface has one egress 'root qdisc', by default the earlier mentioned classless pfifo_fast queueing discipline. Each qdisc can be assigned a handle, which can be used by later configuration statements to refer to that qdisc. Besides an egress qdisc, an interface may also have an ingress, which polices traffic coming in.

The handles of these qdiscs consist of two parts, a major number and a minor number. It is habitual to name the root qdisc '1:', which is equal to '1:0'. The minor number of a qdisc is always 0.

Classes need to have the same major number as their parent.

### 9.5.2.1. How filters are used to classify traffic

Recapping, a typical hierarchy might look like this:

But don't let this tree fool you! You should *not* imagine the kernel to be at the apex of the tree and the network below, that is just not the case. Packets get enqueued and dequeued at the root qdisc, which is the only thing the kernel talks to.

A packet might get classified in a chain like this:

1: –> 1:1 –> 12: –> 12:2

The packet now resides in a queue in a qdisc attached to class 12:2. In this example, a filter was attached to each 'node' in the tree, each choosing a branch to take next. This can make sense. However, this is also possible:

1: –> 12:2

In this case, a filter attached to the root decided to send the packet directly to 12:2.

#### 9.5.2.2. How packets are dequeued to the hardware

When the kernel decides that it needs to extract packets to send to the interface, the root qdisc 1: gets a dequeue request, which is passed to 1:1, which is in turn passed to 10:, 11: and 12:, which each query their siblings, and try to dequeue() from them. In this case, the kernel needs to walk the entire tree, because only 12:2 contains a packet.

In short, nested classes ONLY talk to their parent qdiscs, never to an interface. Only the root qdisc gets dequeued by the kernel!

The upshot of this is that classes never get dequeued faster than their parents allow. And this is exactly what we want: this way we can have SFQ in an inner class, which doesn't do any shaping, only scheduling, and have a shaping outer qdisc, which does the shaping.

---

### 9.5.3. The PRIO qdisc

The PRIO qdisc doesn't actually shape, it only subdivides traffic based on how you configured your filters. You can consider the PRIO qdisc a kind of pfifo_fast on steroids, whereby each band is a separate class instead of a simple FIFO.

When a packet is enqueued to the PRIO qdisc, a class is chosen based on the filter commands you gave. By default, three classes are created. These classes by default contain pure FIFO qdiscs with no internal structure, but you can replace these by any qdisc you have available.

Whenever a packet needs to be dequeued, class :1 is tried first. Higher classes are only used if lower bands all did not give up a packet.

This qdisc is very useful in case you want to prioritize certain kinds of traffic without using only TOS−flags but using all the power of the tc filters. It can also contain more all qdiscs, whereas pfifo_fast is limited to simple fifo qdiscs.

Because it doesn't actually shape, the same warning as for SFQ holds: either use it only if your physical link is really full or wrap it inside a classful qdisc that does shape. The last holds for almost all cable modems and DSL devices.

In formal words, the PRIO qdisc is a Work−Conserving scheduler.

---

#### 9.5.3.1. PRIO parameters & usage

The following parameters are recognized by tc:

*bands*
: Number of bands to create. Each band is in fact a class. If you change this number, you must also change:

*priomap*
: If you do not provide tc filters to classify traffic, the PRIO qdisc looks at the TC_PRIO priority to decide how to enqueue traffic.

  This works just like with the pfifo_fast qdisc mentioned earlier, see there for lots of detail.

The bands are classes, and are called major:1 to major:3 by default, so if your PRIO qdisc is called 12:, tc filter traffic to 12:1 to grant it more priority.

Reiterating, band 0 goes to minor number 1! Band 1 to minor number 2, etc.

---

#### 9.5.3.2. Sample configuration

We will create this tree:

```
     root 1: prio
       /   |   \
     1:1  1:2  1:3
      |    |    |
     10:  20:  30:
     sfq  tbf  sfq
band  0    1    2
```

Bulk traffic will go to 30:, interactive traffic to 20: or 10:.

Command lines:

```
# tc qdisc add dev eth0 root handle 1: prio
## This *instantly* creates classes 1:1, 1:2, 1:3

# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit buffer 1600 limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

Now let's see what we created:

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

As you can see, band 0 has already had some traffic, and one packet was sent while running this command!

We now do some bulk data transfer with a tool that properly sets TOS flags, and take another look:

```
# scp tc ahu@10.0.0.11:./
ahu@10.0.0.11's password:
tc                  100% |*****************************|   353 KB    00:00
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)
```

```
 qdisc sfq 10: quantum 1514b
 Sent 2230 bytes 31 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 389140 bytes 326 pkts (dropped 0, overlimits 0)
```

As you can see, all traffic went to handle 30:, which is the lowest priority band, just as intended. Now to verify that interactive traffic goes to higher bands, we create some interactive traffic:

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 14926 bytes 193 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 401836 bytes 488 pkts (dropped 0, overlimits 0)
```

It worked – all additional traffic has gone to 10:, which is our highest priority qdisc. No traffic was sent to the lowest priority, which previously received our entire scp.

### 9.5.4. The famous CBQ qdisc

As said before, CBQ is the most complex qdisc available, the most hyped, the least understood, and probably the trickiest one to get right. This is not because the authors are evil or incompetent, far from it, it's just that the CBQ algorithm isn't all that precise and doesn't really match the way Linux works.

Besides being classful, CBQ is also a shaper and it is in that aspect that it really doesn't work very well. It should work like this. If you try to shape a 10mbit/s connection to 1mbit/s, the link should be idle 90% of the time. If it isn't, we need to throttle so that it IS idle 90% of the time.

This is pretty hard to measure, so CBQ instead derives the idle time from the number of microseconds that elapse between requests from the hardware layer for more data. Combined, this can be used to approximate how full or empty the link is.

This is rather circumspect and doesn't always arrive at proper results. For example, what if the actual link speed of an interface that is not really able to transmit the full 100mbit/s of data, perhaps because of a badly implemented driver? A PCMCIA network card will also never achieve 100mbit/s because of the way the bus is designed – again, how do we calculate the idle time?

It gets even worse if we consider not–quite–real network devices like PPP over Ethernet or PPTP over TCP/IP. The effective bandwidth in that case is probably determined by the efficiency of pipes to userspace – which is huge.

People who have done measurements discover that CBQ is not always very accurate and sometimes completely misses the mark.

In many circumstances however it works well. With the documentation provided here, you should be able to configure it to work well in most cases.

#### 9.5.4.1. CBQ shaping in detail

As said before, CBQ works by making sure that the link is idle just long enough to bring down the real bandwidth to the configured rate. To do so, it calculates the time that should pass between average packets.

During operations, the effective idletime is measured using an exponential weighted moving average (EWMA), which considers recent packets to be exponentially more important than past ones. The UNIX loadaverage is calculated in the same way.

The calculated idle time is subtracted from the EWMA measured one, the resulting number is called 'avgidle'. A perfectly loaded link has an avgidle of zero: packets arrive exactly once every calculated interval.

An overloaded link has a negative avgidle and if it gets too negative, CBQ shuts down for a while and is then 'overlimit'.

Conversely, an idle link might amass a huge avgidle, which would then allow infinite bandwidths after a few hours of silence. To prevent this, avgidle is capped at maxidle.

If overlimit, in theory, the CBQ could throttle itself for exactly the amount of time that was calculated to pass between packets, and then pass one packet, and throttle again. But see the 'minburst' parameter below.

These are parameters you can specify in order to configure shaping:

*avpkt*
: Average size of a packet, measured in bytes. Needed for calculating maxidle, which is derived from maxburst, which is specified in packets.

*bandwidth*
: The physical bandwidth of your device, needed for idle time calculations.

*cell*
: The time a packet takes to be transmitted over a device may grow in steps, based on the packet size. An 800 and an 806 size packet may take just as long to send, for example – this sets the granularity. Most often set to '8'. Must be an integral power of two.

*maxburst*
: This number of packets is used to calculate maxidle so that when avgidle is at maxidle, this number of average packets can be burst before avgidle drops to 0. Set it higher to be more tolerant of bursts. You can't set maxidle directly, only via this parameter.

*minburst*
: As mentioned before, CBQ needs to throttle in case of overlimit. The ideal solution is to do so for exactly the calculated idle time, and pass 1 packet. However, Unix kernels generally have a hard time scheduling events shorter than 10ms, so it is better to throttle for a longer period, and then pass minburst packets in one go, and then sleep minburst times longer.

  The time to wait is called the offtime. Higher values of minburst lead to more accurate shaping in the long term, but to bigger bursts at millisecond timescales.

*minidle*
: If avgidle is below 0, we are overlimits and need to wait until avgidle will be big enough to send one packet. To prevent a sudden burst from shutting down the link for a prolonged period of time, avgidle is reset to minidle if it gets too low.

Minidle is specified in negative microseconds, so 10 means that avgidle is capped at −10us.

*mpu*

Minimum packet size − needed because even a zero size packet is padded to 64 bytes on ethernet, and so takes a certain time to transmit. CBQ needs to know this to accurately calculate the idle time.

*rate*

Desired rate of traffic leaving this qdisc − this is the 'speed knob'!

Internally, CBQ has a lot of fine tuning. For example, classes which are known not to have data enqueued to them aren't queried. Overlimit classes are penalized by lowering their effective priority. All very smart & complicated.

---

#### 9.5.4.2. CBQ classful behaviour

Besides shaping, using the aforementioned idletime approximations, CBQ also acts like the PRIO queue in the sense that classes can have differing priorities and that lower priority numbers will be polled before the higher priority ones.

Each time a packet is requested by the hardware layer to be sent out to the network, a weighted round robin process ('WRR') starts, beginning with the lower priority classes.

These are then grouped and queried if they have data available. If so, it is returned. After a class has been allowed to dequeue a number of bytes, the next class within that priority is tried.

The following parameters control the WRR process:

*allot*

When the outer CBQ is asked for a packet to send out on the interface, it will try all inner qdiscs (in the classes) in turn, in order of the 'priority' parameter. Each time a class gets its turn, it can only send out a limited amount of data. 'Allot' is the base unit of this amount. See the 'weight' parameter for more information.

*prio*

The CBQ can also act like the PRIO device. Inner classes with lower priority are tried first and as long as they have traffic, other classes are not polled for traffic.

*weight*

Weight helps in the Weighted Round Robin process. Each class gets a chance to send in turn. If you have classes with significantly more bandwidth than other classes, it makes sense to allow them to send more data in one round than the others.

A CBQ adds up all weights under a class, and normalizes them, so you can use arbitrary numbers: only the ratios are important. People have been using 'rate/10' as a rule of thumb and it appears to work well. The renormalized weight is multiplied by the 'allot' parameter to determine how much data can be sent in one round.

Please note that all classes within an CBQ hierarchy need to share the same major number!

---

#### 9.5.4.3. CBQ parameters that determine link sharing & borrowing

Besides purely limiting certain kinds of traffic, it is also possible to specify which classes can borrow capacity from other classes or, conversely, lend out bandwidth.

*Isolated/sharing*
: A class that is configured with 'isolated' will not lend out bandwidth to sibling classes. Use this if you have competing or mutually−unfriendly agencies on your link who do want to give each other freebies.

  The control program tc also knows about 'sharing', which is the reverse of 'isolated'.

*bounded/borrow*
: A class can also be 'bounded', which means that it will not try to borrow bandwidth from sibling classes. tc also knows about 'borrow', which is the reverse of 'bounded'.

A typical situation might be where you have two agencies on your link which are both 'isolated' and 'bounded', which means that they are really limited to their assigned rate, and also won't allow each other to borrow.

Within such an agency class, there might be other classes which are allowed to swap bandwidth.

---

#### 9.5.4.4. Sample configuration

This configuration limits webserver traffic to 5mbit and SMTP traffic to 3 mbit. Together, they may not get more than 6mbit. We have a 100mbit NIC and the classes may borrow bandwidth from each other.

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit         \
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit  \
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20      \
  avpkt 1000 bounded
```

This part installs the root and the customary 1:0 class. The 1:1 class is bounded, so the total bandwidth can't exceed 6mbit.

As said before, CBQ requires a *lot* of knobs. All parameters are explained above, however. The corresponding HTB configuration is lots simpler.

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit  \
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit  \
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
```

These are our two classes. Note how we scale the weight with the configured rate. Both classes are not bounded, but they are connected to class 1:1 which is bounded. So the sum of bandwith of the 2 classes will never be more than 6mbit. The classids need to be within the same major number as the parent CBQ, by the way!

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Both classes have a FIFO qdisc by default. But we replaced these with an SFQ queue so each flow of data is treated equally.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
```

```
  sport 25 0xffff flowid 1:4
```

These commands, attached directly to the root, send traffic to the right qdiscs.

Note that we use 'tc class add' to CREATE classes within a qdisc, but that we use 'tc qdisc add' to actually add qdiscs to these classes.

You may wonder what happens to traffic that is not classified by any of the two rules. It appears that in this case, data will then be processed within 1:0, and be unlimited.

If SMTP+web together try to exceed the set limit of 6mbit/s, bandwidth will be divided according to the weight parameter, giving 5/8 of traffic to the webserver and 3/8 to the mail server.

With this configuration you can also say that webserver traffic will always get at minimum 5/8 * 6 mbit = 3.75 mbit.

---

#### 9.5.4.5. Other CBQ parameters: split & defmap

As said before, a classful qdisc needs to call filters to determine which class a packet will be enqueued to.

Besides calling the filter, CBQ offers other options, defmap & split. This is pretty complicated to understand, and it is not vital. But as this is the only known place where defmap & split are properly explained, I'm doing my best.

As you will often want to filter on the Type of Service field only, a special syntax is provided. Whenever the CBQ needs to figure out where a packet needs to be enqueued, it checks if this node is a 'split node'. If so, one of the sub−qdiscs has indicated that it wishes to receive all packets with a certain configured priority, as might be derived from the TOS field, or socket options set by applications.

The packets' priority bits are or−ed with the defmap field to see if a match exists. In other words, this is a short−hand way of creating a very fast filter, which only matches certain priorities. A defmap of ff (hex) will match everything, a map of 0 nothing. A sample configuration may help make things clearer:

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514 \
  cell 8 avpkt 1000 mpu 64

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

Standard CBQ preamble. I never get used to the sheer amount of numbers required!

Defmap refers to TC_PRIO bits, which are defined as follows:

```
TC_PRIO..          Num  Corresponds to TOS
------------------------------------------------
BESTEFFORT         0    Maximize Reliablity
FILLER             1    Minimize Cost
BULK               2    Maximize Throughput (0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    Minimize Delay (0x10)
CONTROL            7
```

The TC_PRIO.. number corresponds to bits, counted from the right. See the pfifo_fast section for more details how TOS bits are converted to priorities.

Now the interactive and the bulk classes:

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit     \
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20        \
  avpkt 1000 split 1:0 defmap c0

# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit     \
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20        \
  avpkt 1000 split 1:0 defmap 3f
```

The 'split qdisc' is 1:0, which is where the choice will be made. C0 is binary for 11000000, 3F for 00111111, so these two together will match everything. The first class matches bits 7 & 6, and thus corresponds to 'interactive' and 'control' traffic. The second class matches the rest.

Node 1:0 now has a table like this:

```
priority        send to
0               1:3
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

For additional fun, you can also pass a 'change mask', which indicates exactly which priorities you wish to change. You only need to use this if you are running 'tc class change'. For example, to add best effort traffic to 1:2, we could run this:

```
# tc class change dev eth1 classid 1:2 cbq defmap 01/01
```

The priority map over at 1:0 now looks like this:

```
priority        send to
0               1:2
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

FIXME: did not test 'tc class change', only looked at the source.

## 9.5.5. Hierarchical Token Bucket

Martin Devera (<devik>) rightly realised that CBQ is complex and does not seem optimized for many typical situations. His Hierarchical approach is well suited for setups where you have a fixed amount of bandwidth which you want to divide for different purposes, giving each purpose a guaranteed bandwidth, with the

possibility of specifying how much bandwidth can be borrowed.

HTB works just like CBQ but does not resort to idle time calculations to shape. Instead, it is a classful Token Bucket Filter – hence the name. It has only a few parameters, which are well documented on his site.

As your HTB configuration gets more complex, your configuration scales well. With CBQ it is already complex even in simple cases! HTB is not yet a part of the standard kernel, but it should soon be!

If you are in a position to patch your kernel, by all means consider HTB.

#### 9.5.5.1. Sample configuration

Functionally almost identical to the CBQ sample configuration above:

```
# tc qdisc add dev eth0 root handle 1: htb default 30

# tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k

# tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:20 htb rate 3mbit ceil 6mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:30 htb rate 1kbit ceil 6mbit burst 15k
```

The author then recommends SFQ for beneath these classes:

```
# tc qdisc add dev eth0 parent 1:10 handle 10: sfq perturb 10
# tc qdisc add dev eth0 parent 1:20 handle 20: sfq perturb 10
# tc qdisc add dev eth0 parent 1:30 handle 30: sfq perturb 10
```

Add the filters which direct traffic to the right classes:

```
# U32="tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32"
# $U32 match ip dport 80 0xffff flowid 1:10
# $U32 match ip sport 25 0xffff flowid 1:20
```

And that's it – no unsightly unexplained numbers, no undocumented parameters.

HTB certainly looks wonderful – if 10: and 20: both have their guaranteed bandwidth, and more is left to divide, they borrow in a 5:3 ratio, just as you would expect.

Unclassified traffic gets routed to 30:, which has little bandwidth of its own but can borrow everything that is left over. Because we chose SFQ internally, we get fairness thrown in for free!

## 9.6. Classifying packets with filters

To determine which class shall process a packet, the so–called 'classifier chain' is called each time a choice needs to be made. This chain consists of all filters attached to the classful qdisc that needs to decide.

To reiterate the tree, which is not a tree:

```
                    root 1:
                      |
                    _1:1_
```

```
                  /  |  \
                 /   |   \
                /    |    \
              10:   11:   12:
             /   \       /   \
          10:1  10:2   12:1  12:2
```

When enqueueing a packet, at each branch the filter chain is consulted for a relevant instruction. A typical setup might be to have a filter in 1:1 that directs a packet to 12: and a filter on 12: that sends the packet to 12:2.

You might also attach this latter rule to 1:1, but you can make efficiency gains by having more specific tests lower in the chain.

You can't filter a packet 'upwards', by the way. Also, with HTB, you should attach all filters to the root!

And again – packets are only enqueued downwards! When they are dequeued, they go up again, where the interface lives. They do NOT fall off the end of the tree to the network adaptor!

## 9.6.1. Some simple filtering examples

As explained in the Classifier chapter, you can match on literally anything, using a very complicated syntax. To start, we will show how to do the obvious things, which luckily are quite easy.

Let's say we have a PRIO qdisc called '10:' which contains three classes, and we want to assign all traffic from and to port 22 to the highest priority band, the filters would be:

```
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip dport 22 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip sport 80 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2 flowid 10:2
```

What does this say? It says: attach to eth0, node 10: a priority 1 u32 filter that matches on IP destination port 22 *exactly* and send it to band 10:1. And it then repeats the same for source port 80. The last command says that anything unmatched so far should go to band 10:2, the next–highest priority.

You need to add 'eth0', or whatever your interface is called, because each interface has a unique namespace of handles.

To select on an IP address, use this:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip dst 4.3.2.1/32 flowid 10:1
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip src 1.2.3.4/32 flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2      \
  flowid 10:2
```

This assigns traffic to 4.3.2.1 and traffic from 1.2.3.4 to the highest priority queue, and the rest to the next–highest one.

You can concatenate matches, to match on traffic from 1.2.3.4 and from port 80, do this:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 match ip src 4.3.2.1/32
  match ip sport 80 0xffff flowid 10:1
```

### 9.6.2. All the filtering commands you will normally need

Most shaping commands presented here start with this preamble:

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 ..
```

These are the so called 'u32' matches, which can match on ANY part of a packet.

*On source/destination address*
: Source mask 'match ip src 1.2.3.0/24', destination mask 'match ip dst 4.3.2.0/24'. To match a single host, use /32, or omit the mask.

*On source/destination port, all IP protocols*
: Source: 'match ip sport 80 0xffff', 'match ip dport 0xffff'

*On ip protocol (tcp, udp, icmp, gre, ipsec)*
: Use the numbers from /etc/protocols, for example, icmp is 1: 'match ip protocol 1 0xff'.

*On fwmark*
: You can mark packets with either ipchains and have that mark survive routing across interfaces. This is really useful to for example only shape traffic on eth1 that came in on eth0. Syntax: # tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 6 fw flowid 1:1 Note that this is not a u32 match!

  You can place a mark like this:

```
# iptables -A PREROUTING -t mangle -i eth0 -j MARK --set-mark 6
```

  The number 6 is arbitrary.

  If you don't want to understand the full tc filter syntax, just use iptables, and only learn to select on fwmark.

*On the TOS field*
: To select interactive, minimum delay traffic:

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

  Use 0x08 0xff for bulk traffic.

For more filtering commands, see the Advanced Filters chapter.

## 9.7. The Intermediate queueing device (IMQ)

The Intermediate queueing device is not a qdisc but its usage is tightly bound to qdiscs. Within linux, qdiscs are attached to network devices and everything that is queued to the device is first queued to the qdisc. From this concept, two limitations arise:

1. Only egress shaping is possible (an ingress qdisc exists, but its possibilities are very limited compared to classful qdiscs).

2. A qdisc can only see traffic of one interface, global limitations can't be placed.

IMQ is there to help solve those two limitations. In short, you can put everything you choose in a qdisc. Specially marked packets get intercepted in netfilter NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks and pass through the qdisc attached to an imq device. An iptables target is used for marking the packets.

This enables you to do ingress shaping as you can just mark packets coming in from somewhere and/or treat interfaces as classes to set global limits. You can also do lots of other stuff like just putting your http traffic in a qdisc, put new connection requests in a qdisc, ...

## 9.7.1. Sample configuration

The first thing that might come to mind is use ingress shaping to give yourself a high guaranteed bandwidth. ;) Configuration is just like with any other interface:

```
tc qdisc add dev imq0 root handle 1: htb default 20

tc class add dev imq0 parent 1: classid 1:1 htb rate 2mbit burst 15k

tc class add dev imq0 parent 1:1 classid 1:10 htb rate 1mbit
tc class add dev imq0 parent 1:1 classid 1:20 htb rate 1mbit

tc qdisc add dev imq0 parent 1:10 handle 10: pfifo
tc qdisc add dev imq0 parent 1:20 handle 20: sfq

tc filter add dev imq0 parent 10:0 protocol ip prio 1 u32 match \
               ip dst 10.0.0.230/32 flowid 1:10
```

In this example u32 is used for classification. Other classifiers should work as expected. Next traffic has to be selected and marked to be enqueued to imq0.

```
iptables -t mangle -A PREROUTING -i eth0 -j IMQ --todev 0

ip link set imq0 up
```

The IMQ iptables targets is valid in the PREROUTING and POSTROUTING chains of the mangle table. It's syntax is

```
IMQ [ --todev n ]       n : number of imq device
```

An ip6tables target is also provided.

Please note traffic is not enqueued when the target is hit but afterwards. The exact location where traffic enters the imq device depends on the direction of the traffic (in/out). These are the predefined netfilter hooks used by iptables:

```
enum nf_ip_hook_priorities {
        NF_IP_PRI_FIRST = INT_MIN,
        NF_IP_PRI_CONNTRACK = -200,
        NF_IP_PRI_MANGLE = -150,
        NF_IP_PRI_NAT_DST = -100,
        NF_IP_PRI_FILTER = 0,
        NF_IP_PRI_NAT_SRC = 100,
        NF_IP_PRI_LAST = INT_MAX,
```

```
};
```

For ingress traffic, imq registers itself with NF_IP_PRI_MANGLE + 1 priority which means packets enter the imq device directly after the mangle PREROUTING chain has been passed.

For egress imq uses NF_IP_PRI_LAST which honours the fact that packets dropped by the filter table won't occupy bandwidth.

The patches and some more information can be found at the imq site.

---

# Chapter 10. Load sharing over multiple interfaces

There are several ways of doing this. One of the easiest and straightforward ways is 'TEQL' – "True" (or "trivial") link equalizer. Like most things having to do with queueing, load sharing goes both ways. Both ends of a link may need to participate for full effect.

Imagine this situation:

```
                +-------+   eth1   +-------+
                |       |==========|       |
'network 1' ----|   A   |          |   B   |---- 'network 2'
                |       |==========|       |
                +-------+   eth2   +-------+
```

A and B are routers, and for the moment we'll assume both run Linux. If traffic is going from network 1 to network 2, router A needs to distribute the packets over both links to B. Router B needs to be configured to accept this. Same goes the other way around, when packets go from network 2 to network 1, router B needs to send the packets over both eth1 and eth2.

The distributing part is done by a 'TEQL' device, like this (it couldn't be easier):

```
# tc qdisc add dev eth1 root teql0
# tc qdisc add dev eth2 root teql0
# ip link set dev teql0 up
```

Don't forget the 'ip link set up' command!

This needs to be done on both hosts. The device teql0 is basically a roundrobbin distributor over eth1 and eth2, for sending packets. No data ever comes in over an teql device, that just appears on the 'raw' eth1 and eth2.

But now we just have devices, we also need proper routing. One way to do this is to assign a /31 network to both links, and a /31 to the teql0 device as well:

FIXME: does this need something like 'nobroadcast'? A /31 is too small to house a network address and a broadcast address – if this doesn't work as planned, try a /30, and adjust the ip addresses accordingly. You might even try to make eth1 and eth2 do without an IP address!

On router A:

```
# ip addr add dev eth1 10.0.0.0/31
# ip addr add dev eth2 10.0.0.2/31
# ip addr add dev teql0 10.0.0.4/31
```

On router B:

```
# ip addr add dev eth1 10.0.0.1/31
# ip addr add dev eth2 10.0.0.3/31
# ip addr add dev teql0 10.0.0.5/31
```

Router A should now be able to ping 10.0.0.1, 10.0.0.3 and 10.0.0.5 over the 2 real links and the 1 equalized device. Router B should be able to ping 10.0.0.0, 10.0.0.2 and 10.0.0.4 over the links.

If this works, Router A should make 10.0.0.5 its route for reaching network 2, and Router B should make 10.0.0.4 its route for reaching network 1. For the special case where network 1 is your network at home, and network 2 is the Internet, Router A should make 10.0.0.5 its default gateway.

## 10.1. Caveats

Nothing is as easy as it seems. eth1 and eth2 on both router A and B need to have return path filtering turned off, because they will otherwise drop packets destined for ip addresses other than their own:

```
# echo 0 > /proc/net/ipv4/conf/eth1/rp_filter
# echo 0 > /proc/net/ipv4/conf/eth2/rp_filter
```

Then there is the nasty problem of packet reordering. Let's say 6 packets need to be sent from A to B – eth1 might get 1, 3 and 5. eth2 would then do 2, 4 and 6. In an ideal world, router B would receive this in order, 1, 2, 3, 4, 5, 6. But the possibility is very real that the kernel gets it like this: 2, 1, 4, 3, 6, 5. The problem is that this confuses TCP/IP. While not a problem for links carrying many different TCP/IP sessions, you won't be able to to a bundle multiple links and get to ftp a single file lots faster, except when your receiving or sending OS is Linux, which is not easily shaken by some simple reordering.

However, for lots of applications, link load balancing is a great idea.

## 10.2. Other possibilities

William Stearns has used an advanced tunneling setup to achieve good use of multiple, unrelated, internet connections together. It can be found on his tunneling page.

The HOWTO may feature more about this in the future.

# Chapter 11. 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').

IMPORTANT: We received a report that MASQ and SNAT at least collide with marking packets. Rusty Russell explains it in this posting. Turn off the reverse path filter to make it work properly.

Note: to mark packets, 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 the Section 15.5 in the *Cookbook*.

# Chapter 12. Advanced filters for (re–)classifying packets

As explained in the section on classful queueing disciplines, filters are needed to classify packets into any of the sub–queues. These filters are called from within the classful qdisc.

Here is an incomplete list of classifiers available:

*fw*
: Bases the decision on how the firewall has marked the packet. This can be the easy way out if you don't want to learn tc filter syntax. See the Queueing chapter for details.

*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*
: Routes packets based on RSVP . Only useful on networks you control – the Internet does not respect RSVP.

*tcindex*
: Used in the DSMARK qdisc, see the relevant section.

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. Lower numbers get tested first.

*handle*
: This handle means different things to different filters.

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.

## 12.1. 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 occurs, and then the associated action is performed. The simplest type of action would be directing the packet into defined CBQ class.

The command line 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 described above apply to all filters, not only U32.

### 12.1.1. 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:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00100000 00ff0000 at 0 flowid 1:10
```

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:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00000016 0000ffff at nexthdr+0 flowid 1:10
```

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 ambiguous 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 learned.

### 12.1.2. 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
```

FIXME: it has been pointed out that this syntax does not work currently.

Use this to match ACKs on packets smaller than 64 bytes:

```
## match acks the hard way,
## IP protocol 6,
## IP header length 0x5(32 bit words),
## IP Total length 0x34 (ACK + 12 bytes of TCP options)
## TCP ack set (bit 5, offset 33)
# tc filter add dev ppp14 parent 1:0 protocol ip prio 10 u32 \
            match ip protocol 6 0xff \
            match u8 0x05 0x0f at 0 \
            match u16 0x0000 0xffc0 at 2 \
            match u8 0x10 0xff at 33 \
            flowid 1:3
```

This rule will only match TCP packets with ACK bit set, and no further payload. 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 of 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.

### 12.1.3. 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
```

FIXME: tcp dst match does not work as described below:

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 could 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
```

## 12.2. 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.

## 12.3. Policing filters

To make even more complicated setups possible, you can have filters that only match up to a certain bandwidth. You can declare a filter to entirely cease matching above a certain rate, or only to not match only the bandwidth exceeding a certain rate.

So if you decided to police at 4mbit/s, but 5mbit/s of traffic is present, you can stop matching either the entire 5mbit/s, or only not match 1mbit/s, and do send 4mbit/s to the configured class.

If bandwidth exceeds the configured rate, you can drop a packet, reclassify it, or see if another filter will match it.

### 12.3.1. Ways to police

There are basically two ways to police. If you compiled the kernel with 'Estimators', the kernel can measure for each filter how much traffic it is passing, more or less. These estimators are very easy on the CPU, as they simply count 25 times per second how many data has been passed, and calculate the bitrate from that.

The other way works again via a Token Bucket Filter, this time living within your filter. The TBF only matches traffic UP TO your configured bandwidth, if more is offered, only the excess is subject to the configured overlimit action.

#### 12.3.1.1. With the kernel estimator

This is very simple and has only one parameter: avrate. Either the flow remains below avrate, and the filter classifies the traffic to the classid configured, or your rate exceeds it in which case the specified action is taken, which is 'reclassify' by default.

The kernel uses an Exponential Weighted Moving Average for your bandwidth which makes it less sensitive to short bursts.

#### 12.3.1.2. With Token Bucket Filter

Uses the following parameters:

- buffer/maxburst
- mtu/minburst
- mpu
- rate

Which behave mostly identical to those described in the Token Bucket Filter section. Please note however that if you set the mtu of a TBF policer too low, *no* packets will pass, whereas the egress TBF qdisc will just pass them slower.

Another difference is that a policer can only let a packet pass, or drop it. It cannot delay hold on to it in order to delay it.

### 12.3.2. Overlimit actions

If your filter decides that it is overlimit, it can take 'actions'. Currently, three actions are available:

*continue*
: Causes this filter not to match, but perhaps other filters will.

*drop*
: This is a very fierce option which simply discards traffic exceeding a certain rate. It is often used in the ingress policer and has limited uses. For example, you may have a name server that falls over if offered more than 5mbit/s of packets, in which case an ingress filter could be used to make sure no more is ever offered.

*Pass/OK*
: Pass on traffic ok. Might be used to disable a complicated filter, but leave it in place.

*reclassify*
: Most often comes down to reclassification to Best Effort. This is the default action.

### 12.3.3. Examples

The only real example known is mentioned in the 'Protecting your host from SYN floods' section.

FIXME: if you have used this, please share your experience with us

## 12.4. Hashing filters for very fast massive filtering

If you have a need for thousands of rules, for example if you have a lot of clients or computers, all with different QoS specifications, you may find that the kernel spends a lot of time matching all those rules.

By default, all filters reside in one big chain which is matched in descending order of priority. If you have 1000 rules, 1000 checks may be needed to determine what to do with a packet.

Matching would go much quicker if you would have 256 chains with each four rules – if you could divide packets over those 256 chains, so that the right rule will be there.

Hashing makes this possible. Let's say you have 1024 cable modem customers in your network, with IP addresses ranging from 1.2.0.0 to 1.2.3.255, and each has to go in another bin, for example 'lite', 'regular' and 'premium'. You would then have 1024 rules like this:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.1 classid 1:1
...
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.254 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.255 classid 1:2
```

To speed this up, we can use the last part of the IP address as a 'hash key'. We then get 256 tables, the first of which looks like this:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.1.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.2.0 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.0 classid 1:2
```

The next one starts like this:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.1 classid 1:1
...
```

This way, only four checks are needed at most, two on average.

Configuration is pretty complicated, but very worth it by the time you have this many rules. First we make a filter root, then we create a table with 256 entries:

```
# tc filter add dev eth1 parent 1:0 prio 5 protocol ip u32
# tc filter add dev eth1 parent 1:0 prio 5 handle 2: protocol ip u32 divisor 256
```

Now we add some rules to entries in the created table:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
       match ip src 1.2.0.123 flowid 1:1
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
       match ip src 1.2.1.123 flowid 1:2
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
       match ip src 1.2.3.123 flowid 1:3
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
       match ip src 1.2.4.123 flowid 1:2
```

This is entry 123, which contains matches for 1.2.0.123, 1.2.1.123, 1.2.2.123, 1.2.3.123, and sends them to 1:1, 1:2, 1:3 and 1:2 respectively. Note that we need to specify our hash bucket in hex, 0x7b is 123.

Next create a 'hashing filter' that directs traffic to the right entry in the hashing table:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 800:: \
        match ip src 1.2.0.0/16 \
        hashkey mask 0x000000ff at 12 \
        link 2:
```

Ok, some numbers need explaining. The default hash table is called 800:: and all filtering starts there. Then we select the source address, which lives as position 12, 13, 14 and 15 in the IP header, and indicate that we are only interested in the last part. This we send to hash table 2:, which we created earlier.

It is quite complicated, but it does work in practice and performance will be staggering. Note that this example could be improved to the ideal case where each chain contains 1 filter!

---

# Chapter 13. 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.

In the meantime you may want to have a look at the Linux−Kernel sources; read the file Documentation/filesystems/proc.txt. Most of the features are explained there.

(FIXME)

## 13.1. 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

## 13.2. 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.

### 13.2.1. 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.

Several entries in the following list have been copied from /usr/src/linux/Documentation/networking/ip−sysctl.txt, written by Alexey Kuznetsov <kuznet@ms2.inr.ac.ru> and Andi Kleen <ak@muc.de>

*/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*
: Set this to ignore ICMP errors caused by hosts in the network reacting badly to frames sent to what they perceive to be the broadcast address.

*/proc/sys/net/ipv4/icmp_paramprob_rate*
: A relatively unknown ICMP message, which is sent in response to incorrect packets with broken IP or TCP headers. With this file you can control the rate at which it is sent.

*/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*
: Maximum number of listening igmp (multicast) sockets on the host. FIXME: Is this true?

*/proc/sys/net/ipv4/inet_peer_gc_maxtime*
: FIXME: Add a little explanation about the inet peer storage? Minimum interval between garbage collection passes. This interval is in effect under low (or absent) memory pressure on the pool. Measured in jiffies.

*/proc/sys/net/ipv4/inet_peer_gc_mintime*
: Minimum interval between garbage collection passes. This interval is in effect under high memory pressure on the pool. Measured in jiffies.

*/proc/sys/net/ipv4/inet_peer_maxttl*
: Maximum time−to−live of entries. Unused entries will expire after this period of time if there is no memory pressure on the pool (i.e. when the number of entries in the pool is very small). Measured in jiffies.

*/proc/sys/net/ipv4/inet_peer_minttl*
: Minimum time−to−live of entries. Should be enough to cover fragment time−to−live on the reassembling side. This minimum time−to−live is guaranteed if the pool size is less than inet_peer_threshold. Measured in jiffies.

*/proc/sys/net/ipv4/inet_peer_threshold*
: The approximate size of the INET peer storage. Starting from this threshold entries will be thrown aggressively. This threshold also determines entries' time−to−live and time intervals between garbage collection passes. More entries, less time−to−live, less GC interval.

*/proc/sys/net/ipv4/ip_autoconfig*
: This file contains the number one if the host received its IP configuration by RARP, BOOTP, DHCP or a similar mechanism. Otherwise it is zero.

*/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. See also the section on Path MTU discovery in the *Cookbook* chapter.

*/proc/sys/net/ipv4/ipfrag_high_thresh*
: Maximum memory used to reassemble IP fragments. When ipfrag_high_thresh bytes of memory is allocated for this purpose, the fragment handler will toss packets until ipfrag_low_thresh is reached.

*/proc/sys/net/ipv4/ip_nonlocal_bind*
: Set this if you want your applications to be able to bind to an address which doesn't belong to a device on your system. This can be useful when your machine is on a non−permanent (or even dynamic) link, so your services are able to start up and bind to a specific address when your link is down.

*/proc/sys/net/ipv4/ipfrag_low_thresh*
: Minimum memory used to reassemble IP fragments.

*/proc/sys/net/ipv4/ipfrag_time*
: Time in seconds to keep an IP fragment in memory.

*/proc/sys/net/ipv4/tcp_abort_on_overflow*
: A boolean flag controlling the behaviour under lots of incoming connections. When enabled, this causes the kernel to actively send RST packets when a service is overloaded.

*/proc/sys/net/ipv4/tcp_fin_timeout*
: Time to hold socket in state FIN−WAIT−2, if it was closed by our side. Peer can be broken and never close its side, or even died unexpectedly. Default value is 60sec. Usual value used in 2.2 was 180 seconds, you may restore it, but remember that if your machine is even underloaded WEB server, you risk to overflow memory with kilotons of dead sockets, FIN−WAIT−2 sockets are less dangerous than

FIN−WAIT−1, because they eat maximum 1.5K of memory, but they tend to live longer. Cf. tcp_max_orphans.

*/proc/sys/net/ipv4/tcp_keepalive_time*

How often TCP sends out keepalive messages when keepalive is enabled. Default: 2hours.

*/proc/sys/net/ipv4/tcp_keepalive_intvl*

How frequent probes are retransmitted, when a probe isn't acknowledged. Default: 75 seconds.

*/proc/sys/net/ipv4/tcp_keepalive_probes*

How many keepalive probes TCP will send, until it decides that the connection is broken. Default value: 9. Multiplied with tcp_keepalive_intvl, this gives the time a link can be non−responsive after a keepalive has been sent.

*/proc/sys/net/ipv4/tcp_max_orphans*

Maximal number of TCP sockets not attached to any user file handle, held by system. If this number is exceeded orphaned connections are reset immediately and warning is printed. This limit exists only to prevent simple DoS attacks, you _must_ not rely on this or lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value, and tune network services to linger and kill such states more aggressively. Let me remind you again: each orphan eats up to 64K of unswappable memory.

*/proc/sys/net/ipv4/tcp_orphan_retries*

How may times to retry before killing TCP connection, closed by our side. Default value 7 corresponds to 50sec−16min depending on RTO. If your machine is a loaded WEB server, you should think about lowering this value, such sockets may consume significant resources. Cf. tcp_max_orphans.

*/proc/sys/net/ipv4/tcp_max_syn_backlog*

Maximal number of remembered connection requests, which still did not receive an acknowledgment from connecting client. Default value is 1024 for systems with more than 128Mb of memory, and 128 for low memory machines. If server suffers of overload, try to increase this number. Warning! If you make it greater than 1024, it would be better to change TCP_SYNQ_HSIZE in include/net/tcp.h to keep TCP_SYNQ_HSIZE*16<=tcp_max_syn_backlog and to recompile kernel.

*/proc/sys/net/ipv4/tcp_max_tw_buckets*

Maximal number of timewait sockets held by system simultaneously. If this number is exceeded time−wait socket is immediately destroyed and warning is printed. This limit exists only to prevent simple DoS attacks, you _must_ not lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value.

*/proc/sys/net/ipv4/tcp_retrans_collapse*

Bug−to−bug compatibility with some broken printers. On retransmit try to send bigger packets to work around bugs in certain TCP stacks.

*/proc/sys/net/ipv4/tcp_retries1*

How many times to retry before deciding that something is wrong and it is necessary to report this suspicion to network layer. Minimal RFC value is 3, it is default, which corresponds to 3sec−8min depending on RTO.

*/proc/sys/net/ipv4/tcp_retries2*

How may times to retry before killing alive TCP connection. RFC 1122 says that the limit should be longer than 100 sec. It is too small number. Default value 15 corresponds to 13−30min depending on RTO.

*/proc/sys/net/ipv4/tcp_rfc1337*

This boolean enables a fix for 'time−wait assassination hazards in tcp', described in RFC 1337. If enabled, this causes the kernel to drop RST packets for sockets in the time−wait state. Default: 0

*/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*

Use the Host requirements interpretation of the TCP urg pointer field. Most hosts use the older BSD interpretation, so if you turn this on Linux might not communicate correctly with them. Default: FALSE

*/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 sent 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 it was of a previous generation. The timestamp will let it recognize this 'ancient packet'.

*/proc/sys/net/ipv4/tcp_tw_recycle*

Enable fast recycling TIME−WAIT sockets. Default value is 1. It should not be changed without advice/request of technical experts.

*/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.

## 13.2.2. 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*

Accept packets with source address 0.b.c.d with destinations not to this host as local ones. It is supposed that a BOOTP relay daemon will catch and forward such packets.

The default is 0, since this feature is not implemented yet (kernel version 2.2.12).

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

Enable or disable IP forwarding on this interface.

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

See the section on *Reverse Path Filtering*.

*/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, this interface will respond to ARP requests for addresses the kernel has routes to. Can be very useful when building 'ip pseudo bridges'. Do take care that your netmasks are very correct before enabling this! Also be aware that the rp_filter, mentioned elsewhere, also operates on ARP queries!

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

See the section on *Reverse Path Filtering*.

*/proc/sys/net/ipv4/conf/DEV/secure_redirects*
: Accept ICMP redirect messages only for gateways, listed in default gateway list. Enabled by default.

*/proc/sys/net/ipv4/conf/DEV/send_redirects*
: If we send the above mentioned redirects.

*/proc/sys/net/ipv4/conf/DEV/shared_media*
: If it is not set the kernel does not assume that different subnets on this device can communicate directly. Default setting is 'yes'.

*/proc/sys/net/ipv4/conf/DEV/tag*
: FIXME: fill this in

## 13.2.3. Neighbor policy

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*
: Maximum for random delay of answers to neighbor solicitation messages in jiffies (1/100 sec). Not yet implemented (Linux does not have anycast support yet).

*/proc/sys/net/ipv4/neigh/DEV/app_solicit*
: Determines the number of requests to send to the user level ARP daemon. Use 0 to turn off.

*/proc/sys/net/ipv4/neigh/DEV/base_reachable_time*
: A base value used for computing the random reachable time value as specified in RFC2461.

*/proc/sys/net/ipv4/neigh/DEV/delay_first_probe_time*
: Delay for the first time probe if the neighbor is reachable. (see gc_stale_time)

*/proc/sys/net/ipv4/neigh/DEV/gc_stale_time*
: Determines how often to check for stale ARP entries. After an ARP entry is stale it will be resolved again (which is useful when an IP address migrates to another machine). When ucast_solicit is greater than 0 it first tries to send an ARP packet directly to the known host When that fails and mcast_solicit is greater than 0, an ARP request is broadcast.

*/proc/sys/net/ipv4/neigh/DEV/locktime*
: An ARP/neighbor entry is only replaced with a new one if the old is at least locktime old. This prevents ARP cache thrashing.

*/proc/sys/net/ipv4/neigh/DEV/mcast_solicit*
: Maximum number of retries for multicast solicitation.

*/proc/sys/net/ipv4/neigh/DEV/proxy_delay*
: Maximum time (real time is random [0..proxytime]) before answering to an ARP request for which we have an proxy ARP entry. In some cases, this is used to prevent network flooding.

*/proc/sys/net/ipv4/neigh/DEV/proxy_qlen*
: Maximum queue length of the delayed proxy arp timer. (see proxy_delay).

*/proc/sys/net/ipv4/neigh/DEV/retrans_time*
: The time, expressed in jiffies (1/100 sec), between retransmitted Neighbor Solicitation messages. Used for address resolution and to determine if a neighbor is unreachable.

*/proc/sys/net/ipv4/neigh/DEV/ucast_solicit*
: Maximum number of retries for unicast solicitation.

*/proc/sys/net/ipv4/neigh/DEV/unres_qlen*
: Maximum queue length for a pending arp request − the number of packets which are accepted from other layers while the ARP address is still resolved.

*Internet QoS: Architectures and Mechanisms for Quality of Service, Zheng Wang, ISBN 1−55860−608−4*
: Hardcover textbook covering topics related to Quality of Service. Good for understanding basic

concepts.

## 13.2.4. Routing settings

*/proc/sys/net/ipv4/route/error_burst*
: These parameters are used to limit the warning messages written to the kernel log from the routing code. The higher the error_cost factor is, the fewer messages will be written. Error_burst controls when messages will be dropped. The default settings limit warning messages to one every five seconds.

*/proc/sys/net/ipv4/route/error_cost*
: These parameters are used to limit the warning messages written to the kernel log from the routing code. The higher the error_cost factor is, the fewer messages will be written. Error_burst controls when messages will be dropped. The default settings limit warning messages to one every five seconds.

*/proc/sys/net/ipv4/route/flush*
: Writing to this file results in a flush of the routing cache.

*/proc/sys/net/ipv4/route/gc_elasticity*
: Values to control the frequency and behavior of the garbage collection algorithm for the routing cache. This can be important for when doing fail over. At least gc_timeout seconds will elapse before Linux will skip to another route because the previous one has died. By default set to 300, you may want to lower it if you want to have a speedy fail over.

  Also see this post by Ard van Breemen.

*/proc/sys/net/ipv4/route/gc_interval*
: See /proc/sys/net/ipv4/route/gc_elasticity.

*/proc/sys/net/ipv4/route/gc_min_interval*
: See /proc/sys/net/ipv4/route/gc_elasticity.

*/proc/sys/net/ipv4/route/gc_thresh*
: See /proc/sys/net/ipv4/route/gc_elasticity.

*/proc/sys/net/ipv4/route/gc_timeout*
: See /proc/sys/net/ipv4/route/gc_elasticity.

*/proc/sys/net/ipv4/route/max_delay*
: Delays for flushing the routing cache.

*/proc/sys/net/ipv4/route/max_size*
: Maximum size of the routing cache. Old entries will be purged once the cache reached has this size.

*/proc/sys/net/ipv4/route/min_adv_mss*
: FIXME: fill this in

*/proc/sys/net/ipv4/route/min_delay*
: Delays for flushing the routing cache.

*/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*
: Factors which determine if more ICMP redirects should be sent to a specific host. No redirects will be sent once the load limit or the maximum number of redirects has been reached.

*/proc/sys/net/ipv4/route/redirect_number*
: See /proc/sys/net/ipv4/route/redirect_load.

*/proc/sys/net/ipv4/route/redirect_silence*
: Timeout for redirects. After this period redirects will be sent again, even if this has been stopped,

because the load or number limit has been reached.

# Chapter 14. Advanced & less common queueing disciplines

Should you find that you have needs not addressed by the queues mentioned earlier, the kernel contains some other more specialized queues mentioned here.

## 14.1. `bfifo/pfifo`

These classless queues are even simpler than pfifo_fast in that they lack the internal bands – all traffic is really equal. They have one important benefit though, they have some statistics. So even if you don't need shaping or prioritizing, you can use this qdisc to determine the backlog on your interface.

pfifo has a length measured in packets, bfifo in bytes.

### 14.1.1. Parameters & usage

*limit*

Specifies the length of the queue. Measured in bytes for bfifo, in packets for pfifo. Defaults to the interface txqueuelen (see pfifo_fast chapter) packets long or txqueuelen*mtu bytes for bfifo.

## 14.2. Clark–Shenker–Zhang algorithm (CSZ)

This is so theoretical that not even Alexey (the main CBQ author) claims to understand it. From his source:

> David D. Clark, Scott Shenker and Lixia Zhang *Supporting Real−Time Applications in an Integrated Services Packet Network: Architecture and Mechanism*.
>
> As I understand it, the main idea is to create WFQ flows for each guaranteed service and to allocate the rest of bandwith to dummy flow−0. Flow−0 comprises the predictive services and the best effort traffic; it is handled by a priority scheduler with the highest priority band allocated for predictive services, and the rest −−− to the best effort packets.
>
> Note that in CSZ flows are NOT limited to their bandwidth. It is supposed that the flow passed admission control at the edge of the QoS network and it doesn't need further shaping. Any attempt to improve the flow or to shape it to a token bucket at intermediate hops will introduce undesired delays and raise jitter.
>
> At the moment CSZ is the only scheduler that provides true guaranteed service. Another schemes (including CBQ) do not provide guaranteed delay and randomize jitter."
>
> Does not currently seem like a good candidate to use, unless you've read and understand the article mentioned.

# 14.3. DSMARK

Esteve Camps

<marvin@grn.es>

This text is an extract from my thesis on *QoS Support in Linux*, September 2000.

Source documents:

- Draft−almesberger−wajhak−diffserv−linux−01.txt.
- Examples in iproute2 distribution.
- White Paper−QoS protocols and architectures and IP QoS Frequently Asked Questions both by *Quality of Service Forum*.

This chapter was written by Esteve Camps <esteve@hades.udg.es>.

## 14.3.1. Introduction

First of all, first of all, it would be a great idea for you to read RFCs written about this (RFC2474, RFC2475, RFC2597 and RFC2598) at IETF DiffServ working Group web site and Werner Almesberger web site (he wrote the code to support Differentiated Services on Linux).

## 14.3.2. What is Dsmark related to?

Dsmark is a queueing discipline that offers the capabilities needed in Differentiated Services (also called DiffServ or, simply, DS). DiffServ is one of two actual QoS architectures (the other one is called Integrated Services) that is based on a value carried by packets in the DS field of the IP header.

One of the first solutions in IP designed to offer some QoS level was the Type of Service field (TOS byte) in IP header. By changing that value, we could choose a high/low level of throughput, delay or reliability. But this didn't provide sufficient flexibility to the needs of new services (such as real−time applications, interactive applications and others). After this, new architectures appeared. One of these was DiffServ which kept TOS bits and renamed DS field.

## 14.3.3. Differentiated Services guidelines

Differentiated Services is group−oriented. I mean, we don't know anything about flows (this will be the Integrated Services purpose); we know about flow aggregations and we will apply different behaviours depending on which aggregation a packet belongs to.

When a packet arrives to an edge node (entry node to a DiffServ domain) entering to a DiffServ Domain we'll have to policy, shape and/or mark those packets (marking refers to assigning a value to the DS field. It's just like the cows :−) ). This will be the mark/value that the internal/core nodes on our DiffServ Domain will look at to determine which behaviour or QoS level apply.

As you can deduce, Differentiated Services involves a domain on which all DS rules will have to be applied. In fact you can think I will classify all the packets entering my domain. Once they enter my domain they will

be subjected to the rules that my classification dictates and every traversed node will apply that QoS level.

In fact, you can apply your own policies into your local domains, but some *Service Level Agreements* should be considered when connecting to other DS domains.

At this point, you maybe have a lot of questions. DiffServ is more than I've explained. In fact, you can understand that I can not resume more than 3 RFCs in just 50 lines :−).

## 14.3.4. Working with Dsmark

As the DiffServ bibliography specifies, we differentiate boundary nodes and interior nodes. These are two important points in the traffic path. Both types perform a classification when the packets arrive. Its result may be used in different places along the DS process before the packet is released to the network. It's just because of this that the diffserv code supplies an structure called sk_buff, including a new field called skb−>tc_index where we'll store the result of initial classification that may be used in several points in DS treatment.

The skb−>tc_index value will be initially set by the DSMARK qdisc, retrieving it from the DS field in IP header of every received packet. Besides, cls_tcindex classifier will read all or part of skb−>tcindex value and use it to select classes.

But, first of all, take a look at DSMARK qdisc command and its parameters:

```
... dsmark indices INDICES [ default_index DEFAULT_INDEX ] [ set_tc_index ]
```

What do these parameters mean?

- *indices*: size of table of (mask,value) pairs. Maximum value is 2^n, where n>=0.
- *Default_index*: the default table entry index if classifier finds no match.
- *Set_tc_index*: instructs dsmark discipline to retrieve the DS field and store it onto skb−>tc_index.

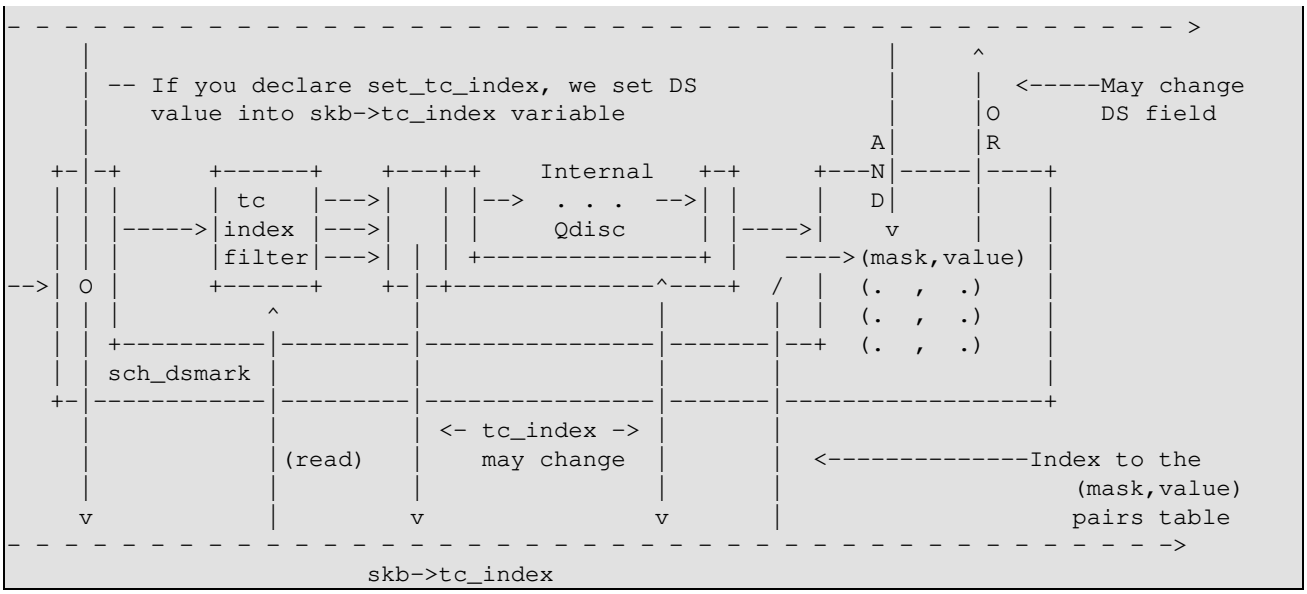
Let's see the DSMARK process.

## 14.3.5. How SCH_DSMARK works.

This qdisc will apply the next steps:

- If we have declared set_tc_index option in qdisc command, DS field is retrieved and stored onto skb−>tc_index variable.
- Classifier is invoked. The classifier will be executed and it will return a class ID that will be stored in skb−>tc_index variable.If no filter matches are found, we consider the default_index option to be the classId to store. If neither set_tc_index nor default_index has been declared results may be unpredictable.
- After been sent to internal qdiscs where you can reuse the result of the filter, the classid returned by the internal qdisc is stored into skb−>tc_index. We will use this value in the future to index a mask−value table. The final result to assign to the packet will be that resulting from next operation:
  ```
  New_Ds_field = ( Old_DS_field & mask ) | value
  ```
- Thus, new value will result from "anding" ds_field and mask values and next, this result "ORed" with value parameter. See next diagram to understand all this process:

```
                        skb->ihp->tos
```




How to do marking? Just change the mask and value of the class you want to remark. See next line of code:

```
tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
```

This changes the (mask,value) pair in hash table, to remark packets belonging to class 1:1.You have to "change" this values because of default values that (mask,value) gets initially (see table below).

Now, we'll explain how TC_INDEX filter works and how fits into this. Besides, TCINDEX filter can be used in other configurations rather than those including DS services.

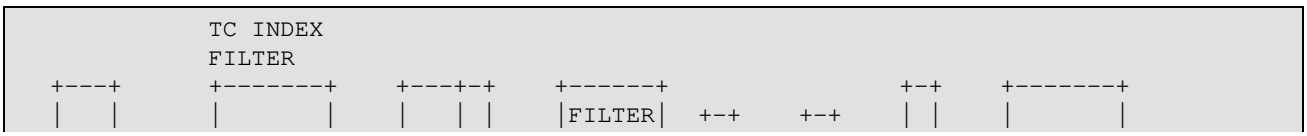
## 14.3.6. TC_INDEX Filter

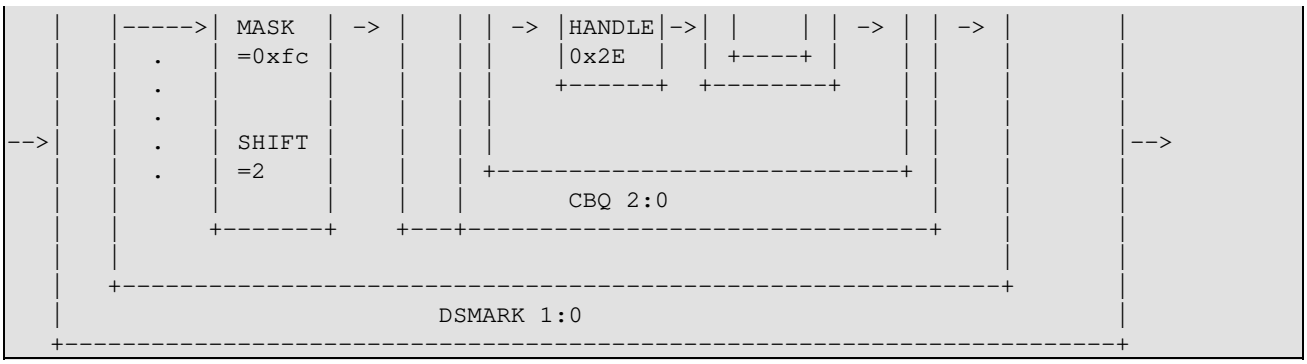
This is the basic command to declare a TC_INDEX filter:

```
... tcindex [ hash SIZE ] [ mask MASK ] [ shift SHIFT ]
            [ pass_on | fall_through ]
            [ classid CLASSID ] [ police POLICE_SPEC ]
```

Next, we show the example used to explain TC_INDEX operation mode. Pay attention to bolded words: tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 *set_tc_index* tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex *mask 0xfc shift 2* tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq bandwidth 10Mbit cell 8 avpkt 1000 mpu 64 # EF traffic class tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 10Mbit rate 1500Kbit avpkt 1000 prio 1 bounded isolated allot 1514 weight 1 maxburst 10 # Packet fifo qdisc for EF traffic tc qdisc add dev eth0 parent 2:1 pfifo limit 5 tc filter add dev eth0 parent 2:0 protocol ip prio 1 *handle 0x2e* tcindex *classid 2:1 pass_on* (This code is not complete. It's just an extract from EFCBQ example included in iproute2 distribution).

First of all, suppose we receive a packet marked as EF . If you read RFC2598, you'll see that DSCP recommended value for EF traffic is 101110. This means that DS field will be 10111000 (remember that less significant bits in TOS byte are not used in DS) or 0xb8 in hexadecimal codification.






The packet arrives, then, set with 0xb8 value at DS field. As we explained before, dsmark qdisc identified by 1:0 id in the example, retrieves DS field and store it in skb–>tc_index variable. Next step in the example will correspond to the filter associated to this qdisc (second line in the example). This will perform next operations:

```
Value1 = skb->tc_index & MASK
Key = Value1 >> SHIFT
```

In the example, MASK=0xFC i SHIFT=2.

```
Value1 = 10111000 & 11111100 = 10111000
Key = 10111000 >> 2 = 00101110 -> 0x2E in hexadecimal
```

The returned value will correspond to a qdisc internal filter handle (in the example, identifier 2:0). If a filter with this id exists, policing and metering conditions will be verified (in case that filter includes this) and the classid will be returned (in our example, classid 2:1) and stored in skb–>tc_index variable.

But if any filter with that identifier is found, the result will depend on fall_through flag declaration. If so, value key is returned as classid. If not, an error is returned and process continues with the rest filters. Be careful if you use fall_through flag; this can be done if a simple relation exists between values of skb–>tc_index variable and class id's.

The latest parameters to comment on are hash and pass_on. The first one relates to hash table size. Pass_on will be used to indicate that if no classid equal to the result of this filter is found, try next filter. The default action is fall_through (look at next table).

Finally, let's see which possible values can be set to all this TCINDEX parameters:

| TC Name | Value | Default |
|---|---|---|
| Hash | 1...0x10000 | Implementation dependent |
| Mask | 0...0xffff | 0xffff |
| Shift | 0...15 | 0 |
| Fall through / Pass_on | Flag | Fall_through |
| Classid | Major:minor | None |
| Police | ..... | None |

This kind of filter is very powerful. It's necessary to explore all possibilities. Besides, this filter is not only used in DiffServ configurations. You can use it as any other kind of filter.

I recommend you to look at all DiffServ examples included in iproute2 distribution. I promise I will try to

complement this text as soon as I can. Besides, all I have explained is the result of a lot of tests. I would thank you tell me if I'm wrong in any point.

# 14.4. Ingress qdisc

All qdiscs discussed so far are egress qdiscs. Each interface however can also have an ingress qdisc which is not used to send packets out to the network adaptor. Instead, it allows you to apply tc filters to packets coming in over the interface, regardless of whether they have a local destination or are to be forwarded.

As the tc filters contain a full Token Bucket Filter implementation, and are also able to match on the kernel flow estimator, there is a lot of functionality available. This effectively allows you to police incoming traffic, before it even enters the IP stack.

## 14.4.1. Parameters & usage

The ingress qdisc itself does not require any parameters. It differs from other qdiscs in that it does not occupy the root of a device. Attach it like this:

```
# tc qdisc add dev eth0 ingress
```

This allows you to have other, sending, qdiscs on your device besides the ingress qdisc.

For a contrived example how the ingress qdisc could be used, see the Cookbook.

# 14.5. Random Early Detection (RED)

This section 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.

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 synchronization. When retransmit synchronization 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 burstily 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, also called Random Early Drop, as that is how it works.

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 synchronization. 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 synchronization. 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 ok.

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 OK on high speed Internet links with a 1500byte MTU.

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

## 14.6. Generic Random Early Detection

Not a lot is known about GRED. It looks like GRED with several internal queues, whereby the internal queue is chosen based on the Diffserv tcindex field. According to a slide found here, it contains the capabilities of Cisco's 'Distributed Weighted RED', as well as Dave Clark's RIO.

Each virtual queue can have its own Drop Parameters specified.

FIXME: get Jamal or Werner to tell us more

## 14.7. VC/ATM emulation

This is quite a major effort by Werner Almesberger to allow you to build Virtual Circuits over TCP/IP sockets. A Virtual Circuit is a concept from ATM network theory.

For more information, see the ATM on Linux homepage.

## 14.8. Weighted Round Robin (WRR)

This qdisc is not included in the standard kernels but can be downloaded from ý. Currently the qdisc is only tested with Linux 2.2 kernels but it will probably work with 2.4/2.5 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 built−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.

---

# Chapter 15. 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.

## 15.1. 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: 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?

## 15.2. 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.

It appears that you need the very latest version of the iproute2 tools to get this to work with 2.4.0.

```
#! /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
```

## 15.3. Rate limit ICMP to prevent dDoS

Recently, distributed denial of service attacks have become a major nuisance on the Internet. By properly filtering and rate limiting 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
```

## 15.4. 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 an application 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
```

## 15.5. 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-|
|------|
```

### 15.5.1. 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
```

## 15.6. Circumventing Path MTU Discovery issues with per route MTU settings

For sending bulk data, the Internet generally works better when using larger packets. Each packet implies a routing decision, when sending a 1 megabyte file, this can either mean around 700 packets when using packets that are as large as possible, or 4000 if using the smallest default.

However, not all parts of the Internet support full 1460 bytes of payload per packet. It is therefore necessary to try and find the largest packet that will 'fit', in order to optimize a connection.

This process is called 'Path MTU Discovery', where MTU stands for 'Maximum Transfer Unit.'

When a router encounters a packet that's too big too send in one piece, AND it has been flagged with the "Don't Fragment" bit, it returns an ICMP message stating that it was forced to drop a packet because of this. The sending host acts on this hint by sending smaller packets, and by iterating it can find the optimum packet size for a connection over a certain path.

This used to work well until the Internet was discovered by hooligans who do their best to disrupt communications. This in turn lead administrators to either block or shape ICMP traffic in a misguided attempt to improve security or robustness of their Internet service.

What has happened now is that Path MTU Discovery is working less and less well and fails for certain routes, which leads to strange TCP/IP sessions which die after a while.

Although I have no proof for this, two sites who I used to have this problem with both run Alteon Acedirectors before the affected systems – perhaps somebody more knowledgeable can provide clues as to why this happens.

### 15.6.1. Solution

When you encounter sites that suffer from this problem, you can disable Path MTU discovery by setting it manually. Koos van den Hout, slightly edited, writes:

> The following problem: I set the mtu/mru of my leased line running ppp to 296 because it's only 33k6 and I cannot influence the queueing on the other side. At 296, the response to a key press is within a reasonable time frame.
>
> And, on my side I have a masqrouter running (of course) Linux.
>
> Recently I split 'server' and 'router' so most applications are run on a different machine than the routing happens on.
>
> I then had trouble logging into irc. Big panic! Some digging did find out that I got connected to irc, even showed up as 'connected' on irc but I did not receive the motd from irc. I checked what could be wrong and noted that I already had some previous trouble reaching certain websites related to the MTU, since I had no trouble reaching them when the MTU was 1500, the problem just showed when the MTU was set to 296. Since irc servers block about every kind of traffic not needed for their immediate operation, they also block icmp.
>
> I managed to convince the operators of a webserver that this was the cause of a problem, but the irc server operators were not going to fix this.
>
> So, I had to make sure outgoing masqueraded traffic started with the lower mtu of the outside link. But I want local ethernet traffic to have the normal mtu (for things like nfs traffic).
>
> Solution:

```
ip route add default via 10.0.0.1 mtu 296
```

(10.0.0.1 being the default gateway, the inside address of the masquerading router)

In general, it is possible to override PMTU Discovery by setting specific routes. For example, if only a certain subnet is giving problems, this should help:

```
ip route add 195.96.96.0/24 via 10.0.0.1 mtu 1000
```

## 15.7. Circumventing Path MTU Discovery issues with MSS Clamping (for ADSL, cable, PPPoE & PPtP users)

As explained above, Path MTU Discovery doesn't work as well as it should anymore. If you know for a fact that a hop somewhere in your network has a limited (<1500) MTU, you cannot rely on PMTU Discovery finding this out.

Besides MTU, there is yet another way to set the maximum packet size, the so called Maximum Segment Size. This is a field in the TCP Options part of a SYN packet.

Recent Linux kernels, and a few PPPoE drivers (notably, the excellent Roaring Penguin one), feature the possibility to 'clamp the MSS'.

The good thing about this is that by setting the MSS value, you are telling the remote side unequivocally 'do not ever try to send me packets bigger than this value'. No ICMP traffic is needed to get this to work.

The bad thing is that it's an obvious hack – it breaks 'end to end' by modifying packets. Having said that, we use this trick in many places and it works like a charm.

In order for this to work you need at least iptables−1.2.1a and Linux 2.4.3 or higher. The basic command line is:

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS  --clamp-mss-to-pmtu
```

This calculates the proper MSS for your link. If you are feeling brave, or think that you know best, you can also do something like this:

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --set-mss 128
```

This sets the MSS of passing SYN packets to 128. Use this if you have VoIP with tiny packets, and huge http packets which are causing chopping in your voice calls.

## 15.8. The Ultimate Traffic Conditioner: Low Latency, Fast Up & Downloads

Note: This script has recently been upgraded and previously only worked for Linux clients in your network! So you might want to update if you have Windows machines or Macs in your network and noticed that they were not able to download faster while others were uploading.

I attempted to create the holy grail:

*Maintain low latency for interactive traffic at all times*
: This means that downloading or uploading files should not disturb SSH or even telnet. These are the most important things, even 200ms latency is sluggish to work over.

*Allow 'surfing' at reasonable speeds while up or downloading*
: Even though http is 'bulk' traffic, other traffic should not drown it out too much.

*Make sure uploads don't harm downloads, and the other way around*
: This is a much observed phenomenon where upstream traffic simply destroys download speed.

It turns out that all this is possible, at the cost of a tiny bit of bandwidth. The reason that uploads, downloads and ssh hurt each other is the presence of large queues in many domestic access devices like cable or DSL modems.

The next section explains in depth what causes the delays, and how we can fix them. You can safely skip it and head straight for the script if you don't care how the magic is performed.

---

## 15.8.1. Why it doesn't work well by default

ISPs know that they are benchmarked solely on how fast people can download. Besides available bandwidth, download speed is influenced heavily by packet loss, which seriously hampers TCP/IP performance. Large queues can help prevent packet loss, and speed up downloads. So ISPs configure large queues.

These large queues however damage interactivity. A keystroke must first travel the upstream queue, which may be seconds (!) long and go to your remote host. It is then displayed, which leads to a packet coming back, which must then traverse the downstream queue, located at your ISP, before it appears on your screen.

This HOWTO teaches you how to mangle and process the queue in many ways, but sadly, not all queues are accessible to us. The queue over at the ISP is completely off–limits, whereas the upstream queue probably lives inside your cable modem or DSL device. You may or may not be able to configure it. Most probably not.

So, what next? As we can't control either of those queues, they must be eliminated, and moved to your Linux router. Luckily this is possible.

*Limit upload speed*
: By limiting our upload speed to slightly less than the truly available rate, no queues are built up in our modem. The queue is now moved to Linux.

*Limit download speed*
: This is slightly trickier as we can't really influence how fast the internet ships us data. We can however drop packets that are coming in too fast, which causes TCP/IP to slow down to just the rate we want. Because we don't want to drop traffic unnecessarily, we configure a 'burst' size we allow at higher speed.

Now, once we have done this, we have eliminated the downstream queue totally (except for short bursts), and gain the ability to manage the upstream queue with all the power Linux offers.

What remains to be done is to make sure interactive traffic jumps to the front of the upstream queue. To make sure that uploads don't hurt downloads, we also move ACK packets to the front of the queue. This is what normally causes the huge slowdown observed when generating bulk traffic both ways. The ACKnowledgements for downstream traffic must compete with upstream traffic, and get delayed in the

process.

If we do all this we get the following measurements using an excellent ADSL connection from xs4all in the Netherlands:

```
Baseline latency:
round-trip min/avg/max = 14.4/17.1/21.7 ms

Without traffic conditioner, while downloading:
round-trip min/avg/max = 560.9/573.6/586.4 ms

Without traffic conditioner, while uploading:
round-trip min/avg/max = 2041.4/2332.1/2427.6 ms

With conditioner, during 220kbit/s upload:
round-trip min/avg/max = 15.7/51.8/79.9 ms

With conditioner, during 850kbit/s download:
round-trip min/avg/max = 20.4/46.9/74.0 ms

When uploading, downloads proceed at ~80% of the available speed. Uploads
at around 90%. Latency then jumps to 850 ms, still figuring out why.
```

What you can expect from this script depends a lot on your actual uplink speed. When uploading at full speed, there will always be a single packet ahead of your keystroke. That is the lower limit to the latency you can achieve – divide your MTU by your upstream speed to calculate. Typical values will be somewhat higher than that. Lower your MTU for better effects!

Next, two versions of this script, one with Devik's excellent HTB, the other with CBQ which is in each Linux kernel, unlike HTB. Both are tested and work well.

## 15.8.2. The actual script (CBQ)

Works on all kernels. Within the CBQ qdisc we place two Stochastic Fairness Queues that make sure that multiple bulk streams don't drown each other out.

Downstream traffic is policed using a tc filter containing a Token Bucket Filter.

You might improve on this script by adding 'bounded' to the line that starts with 'tc class add .. classid 1:20'. If you lowered your MTU, also lower the allot & avpkt numbers!

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null
```

```
###### uplink

# install root CBQ

tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit

# shape everything at $UPLINK speed - this prevents huge queues in your
# DSL modem which destroy latency:
# main class

tc class add dev $DEV parent 1: classid 1:1 cbq rate ${UPLINK}kbit \
allot 1500 prio 5 bounded isolated

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 cbq rate ${UPLINK}kbit \
   allot 1600 prio 1 avpkt 1000

# bulk and default class 1:20 - gets slightly less traffic,
#  and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 cbq rate $[9*$UPLINK/10]kbit \
   allot 1600 prio 2 avpkt 1000

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# start filters
# TOS Minimum Delay (ssh, NOT scp) in 1:10:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10

# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 11 u32 \
        match ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets in
# the interactive class:

tc filter add dev $DEV parent 1: protocol ip prio 12 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20

tc filter add dev $DEV parent 1: protocol ip prio 13 u32 \
   match ip dst 0.0.0.0/0 flowid 1:20

########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress
```

```
# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:

tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

If you want this script to be run by ppp on connect, copy it to /etc/ppp/ip−up.d.

If the last two lines give an error, update your tc tool to a newer version!

## 15.8.3. The actual script (HTB)

The following script achieves all goals using the wonderful HTB queue, see the relevant chapter. Well worth patching your kernel for!

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### uplink

# install root HTB, point default traffic to 1:20:

tc qdisc add dev $DEV root handle 1: htb default 20

# shape everything at $UPLINK speed - this prevents huge queues in your
# DSL modem which destroy latency:

tc class add dev $DEV parent 1: classid 1:1 htb rate ${UPLINK}kbit burst 6k

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 htb rate ${UPLINK}kbit \
   burst 6k prio 1

# bulk & default class 1:20 - gets slightly less traffic,
# and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 htb rate $[9*$UPLINK/10]kbit \
   burst 6k prio 2

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# TOS Minimum Delay (ssh, NOT scp) in 1:10:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10
```

```
# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
        match ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets in
# the interactive class:

tc filter add dev $DEV parent 1: protocol ip prio 10 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20


########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress

# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:

tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

If you want this script to be run by ppp on connect, copy it to /etc/ppp/ip−up.d.

If the last two lines give an error, update your tc tool to a newer version!

## 15.9. Rate limiting a single host or netmask

Although this is described in stupendous details elsewhere and in our manpages, this question gets asked a lot and happily there is a simple answer that does not need full comprehension of traffic control.

This three line script does the trick:

```
tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit

tc class add dev $DEV parent 1: classid 1:1 cbq rate 512kbit \
allot 1500 prio 5 bounded isolated

tc filter add dev $DEV parent 1: protocol ip prio 16 u32 \
match ip dst 195.96.96.97 flowid 1:1
```

The first line installs a class based queue on your interface, and tells the kernel that for calculations, it can be assumed to be a 10mbit interface. If you get this wrong, no real harm is done. But getting it right will make everything more precise.

The second line creates a 512kbit class with some reasonable defaults. For details, see the cbq manpages and Chapter 9.

The last line tells which traffic should go to the shaped class. Traffic not matched by this rule is NOT shaped. To make more complicated matches (subnets, source ports, destination ports), see Section 9.6.2.

If you changed anything and want to reload the script, execute 'tc qdisc del dev $DEV root' to clean up your existing configuration.

The script can further be improved by adding a last optional line 'tc qdisc add dev $DEV parent 1:1 sfq perturb 10'. See Section 9.2.3 for details on what this does.

---

# Chapter 16. Building bridges, and pseudo−bridges with Proxy ARP

Bridges are devices which can be installed in a network without any reconfiguration. A network switch is basically a many−port bridge. A bridge is often a 2−port switch. Linux does however support multiple interfaces in a bridge, making it a true switch.

Bridges are often deployed when confronted with a broken network that needs to be fixed without any alterations. Because the bridge is a layer−2 device, one layer below IP, routers and servers are not aware of its existence. This means that you can transparently block or modify certain packets, or do shaping.

Another good thing is that a bridge can often be replaced by a cross cable or a hub, should it break down.

The bad news is that a bridge can cause great confusion unless it is very well documented. It does not appear in traceroutes, but somehow packets disappear or get changed from point A to point B ('this network is HAUNTED!'). You should also wonder if an organization that 'does not want to change anything' is doing the right thing.

The Linux 2.4/2.5 bridge is documented on this page.

## 16.1. State of bridging and iptables

As of Linux 2.4.14, bridging and iptables do not 'see' each other without help. If you bridge packets from eth0 to eth1, they do not 'pass' by iptables. This means that you cannot do filtering, or NAT or mangling or whatever.

There are several projects going on to fix this, the truly right one is by the author of the Linux 2.4 bridging code, Lennert Buytenhek. He recently informed us that as of bridge−nf 0.0.2 (see the url above), the code is stable and usable in production environments. He is now asking the kernel people if and how the patch can be merged, stay tuned!

## 16.2. Bridging and shaping

This does work as advertised. Be sure to figure out which side each interface is on, otherwise you might be shaping outbound traffic in your internal interface, which won't work. Use tcpdump if needed.

## 16.3. Pseudo−bridges with Proxy−ARP

If you just want to implement a Pseudo−bridge, skip down a few sections to 'Implementing it', but it is wise to read a bit about how it works in practice.

A Pseudo−bridge works a bit differently. By default, a bridge passes packets unaltered from one interface to the other. It only looks at the hardware address of packets to determine what goes where. This in turn means that you can bridge traffic that Linux does not understand, as long as it has an hardware address it does.

A 'Pseudo−bridge' works differently and looks more like a hidden router than a bridge, but like a bridge, it has little impact on network design.

An advantage of the fact that it is not a bridge lies in the fact that packets really pass through the kernel, and can be filtered, changed, redirected or rerouted.

A real bridge can also be made to perform these feats, but it needs special code, like the Ethernet Frame Diverter, or the above mentioned patch.

Another advantage of a pseudo−bridge is that it does not pass packets it does not understand − thus cleaning your network of a lot of cruft. In cases where you need this cruft (like SAP packets, or Netbeui), use a real bridge.

## 16.3.1. ARP & Proxy−ARP

When a host wants to talk to another host on the same physical network segment, it sends out an Address Resolution Protocol packet, which, somewhat simplified, reads like this 'who has 10.0.0.1, tell 10.0.0.7'. In response to this, 10.0.0.1 replies with a short 'here' packet.

10.0.0.7 then sends packets to the hardware address mentioned in the 'here' packet. It caches this hardware address for a relatively long time, and after the cache expires, it re−asks the question.

When building a Pseudo−bridge, we instruct the bridge to reply to these ARP packets, which causes the hosts in the network to send its packets to the bridge. The bridge then processes these packets, and sends them to the relevant interface.

So, in short, whenever a host on one side of the bridge asks for the hardware address of a host on the other, the bridge replies with a packet that says 'hand it to me'.

This way, all data traffic gets transmitted to the right place, and always passes through the bridge.

## 16.3.2. Implementing it

In the bad old days, it used to be possible to instruct the Linux Kernel to perform 'proxy−ARP' for just any subnet. So, to configure a pseudo−bridge, you would have to specify both the proper routes to both sides of the bridge AND create matching proxy−ARP rules. This is bad in that it requires a lot of typing, but also because it easily allows you to make mistakes which make your bridge respond to ARP queries for networks it does not know how to route.

With Linux 2.4/2.5 (and possibly 2.2), this possibility has been withdrawn and has been replaced by a flag in the /proc directory, called 'proxy_arp'. The procedure for building a pseudo−bridge is then:

1. Assign an IP address to both interfaces, the 'left' and the 'right' one
2. Create routes so your machine knows which hosts reside on the left, and which on the right
3. Turn on proxy−ARP on both interfaces, echo 1 > /proc/sys/net/ipv4/conf/ethL/proxy_arp, echo 1 > /proc/sys/net/ipv4/conf/ethR/proxy_arp, where L and R stand for the numbers of your interfaces on the left and on the right side

Also, do not forget to turn on the ip_forwarding flag! When converting from a true bridge, you may find that this flag was turned off as it is not needed when bridging.

Another thing you might note when converting is that you need to clear the arp cache of computers in the network − the arp cache might contain old pre−bridge hardware addresses which are no longer correct.

On a Cisco, this is done using the command 'clear arp−cache', under Linux, use 'arp −d ip.address'. You can also wait for the cache to expire manually, which can take rather long.

You can speed this up using the wonderful 'arping' tool, which on many distributions is part of the 'iputils' package. Using 'arping' you can send out unsolicited ARP messages so as to update remote arp caches.

This is a very powerful technique that is also used by 'black hats' to subvert your routing!

On Linux 2.4, you may need to execute 'echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind' before being able to send out unsolicited ARP messages!

You may also discover that your network was misconfigured if you are/were of the habit of specifying routes without netmasks. To explain, some versions of route may have guessed your netmask right in the past, or guessed wrong without you noticing. When doing surgical routing like described above, it is *vital* that you check your netmasks!

---

# Chapter 17. 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 standardized 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 :−)

---

# Chapter 18. Other possibilities

This chapter is a list of projects having to do with advanced Linux routing & traffic shaping. Some of these links may deserve chapters of their own, some are documented very well of themselves, and don't need more HOWTO.

*802.1Q VLAN Implementation for Linux (site)*
: VLANs are a very cool way to segregate your networks in a more virtual than physical way. Good information on VLANs can be found here. With this implementation, you can have your Linux box talk VLANs with machines like Cisco Catalyst, 3Com: {Corebuilder, Netbuilder II, SuperStack II switch 630}, Extreme Ntwks Summit 48, Foundry: {ServerIronXL, FastIron}.

  A great HOWTO about VLANs can be found here.

  Update: has been included in the kernel as of 2.4.14 (perhaps 13).

*Alternate 802.1Q VLAN Implementation for Linux (site)*
: Alternative VLAN implementation for linux. This project was started out of disagreement with the 'established' VLAN project's architecture and coding style, resulting in a cleaner overall design.

*Linux Virtual Server (site)*
: These people are brilliant. The Linux Virtual Server is a highly scalable and highly available server built on a cluster of real servers, with the load balancer running on the Linux operating system. The architecture of the cluster is transparent to end users. End users only see a single virtual server.

  In short whatever you need to load balance, at whatever level of traffic, LVS will have a way of doing it. Some of their techniques are positively evil! For example, they let several machines have the same IP address on a segment, but turn off ARP on them. Only the LVS machine does ARP – it then decides which of the backend hosts should handle an incoming packet, and sends it directly to the right MAC address of the backend server. Outgoing traffic will flow directly to the router, and not via the LVS machine, which does therefor not need to see your 5Gbit/s of content flowing to the world, and cannot be a bottleneck.

  The LVS is implemented as a kernel patch in Linux 2.0 and 2.2, but as a Netfilter module in 2.4/2.5, so it does not need kernel patches! Their 2.4 support is still in early development, so beat on it and give feedback or send patches.

*CBQ.init (site)*
: Configuring CBQ can be a bit daunting, especially if all you want to do is shape some computers behind a router. CBQ.init can help you configure Linux with a simplified syntax.

  For example, if you want all computers in your 192.168.1.0/24 subnet (on 10mbit eth1) to be limited to 28kbit/s download speed, put this in the CBQ.init configuration file:

```
DEVICE=eth1,10Mbit,1Mbit
RATE=28Kbit
WEIGHT=2Kbit
PRIO=5
RULE=192.168.1.0/24
```

  By all means use this program if the 'how and why' don't interest you. We're using CBQ.init in production and it works very well. It can even do some more advanced things, like time dependent shaping. The documentation is embedded in the script, which explains why you can't find a README.

*Chronox easy shaping scripts (site)*

Stephan Mueller (smueller@chronox.de) wrote two useful scripts, 'limit.conn' and 'shaper'. The first one allows you to easily throttle a single download session, like this:

```
# limit.conn -s SERVERIP -p SERVERPORT -l LIMIT
```

It works on Linux 2.2 and 2.4/2.5.

The second script is more complicated, and can be used to make lots of different queues based on iptables rules, which are used to mark packets which are then shaped.

*Virtual Router Redundancy Protocol implementation (site)*

This is purely for redundancy. Two machines with their own IP address and MAC Address together create a third IP Address and MAC Address, which is virtual. Originally intended purely for routers, which need constant MAC addresses, it also works for other servers.

The beauty of this approach is the incredibly easy configuration. No kernel compiling or patching required, all userspace.

Just run this on all machines participating in a service:

```
# vrrpd -i eth0 -v 50 10.0.0.22
```

And you are in business! 10.0.0.22 is now carried by one of your servers, probably the first one to run the vrrp daemon. Now disconnect that computer from the network and very rapidly one of the other computers will assume the 10.0.0.22 address, as well as the MAC address.

I tried this over here and had it up and running in 1 minute. For some strange reason it decided to drop my default gateway, but the -n flag prevented that.

This is a 'live' fail over:

```
64 bytes from 10.0.0.22: icmp_seq=3 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=4 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=5 ttl=255 time=16.8 ms
64 bytes from 10.0.0.22: icmp_seq=6 ttl=255 time=1.8 ms
64 bytes from 10.0.0.22: icmp_seq=7 ttl=255 time=1.7 ms
```

Not *one* ping packet was lost! Just after packet 4, I disconnected my P200 from the network, and my 486 took over, which you can see from the higher latency.

---

# Chapter 19. 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 Salim, 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.

*Differentiated Services on Linux*
: This document by Werner Almesberger, Jamal Hadi Salim and Alexey Kuznetsov describes DiffServ facilities in the Linux kernel, amongst which are TBF, GRED, the DSMARK qdisc and the tcindex classifier.

*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.

*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 :−)

*Docum experimental site(site)*
: Stef Coene is busy convincing his boss to sell Linux support, and so he is experimenting a lot, especially with managing bandwidth. His site has a lot of practical information, examples, tests and also points out some CBQ/tc bugs.

*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.

---

# Chapter 20. 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 recognize the people who are helping.

- Junk Alins

  <juanjo@mat.upc.es>
- Joe Van Andel
- Michael T. Babcock

  <mbabcock@fibrespeed.net>
- Christopher Barton

  <cpbarton%uiuc.edu>
- Ard van Breemen

  <ard%kwaak.net>
- Ron Brinker

  <service%emcis.com>
- ?ukasz Bromirski

  <L.Bromirski@prosys.com.pl>
- Lennert Buytenhek

  <buytenh@gnu.org>
- Esteve Camps

  <esteve@hades.udg.es>
- Stef Coene

  <stef.coene@docum.org>
- Don Cohen

  <don-lartc%isis.cs3-inc.com>
- Jonathan Corbet

  <lwn%lwn.net>
- Gerry N5JXS Creager

  <gerry%cs.tamu.edu>
- Marco Davids

  <marco@sara.nl>
- Jonathan Day

  <jd9812@my-deja.com>
- Martin aka devik Devera

<devik@cdi.cz>
- Stephan "Kobold" Gehring

<Stephan.Gehring@bechtle.de>
- Jacek Glinkowski

<jglinkow%hns.com>
- Andrea Glorioso

<sama%perchetopi.org>
- Nadeem Hasan

<nhasan@usa.net>
- Erik Hensema

<erik%hensema.xs4all.nl>
- Vik Heyndrickx

<vik.heyndrickx@edchq.com>
- Spauldo Da Hippie

<spauldo%usa.net>
- Koos van den Hout

<koos@kzdoos.xs4all.nl>
- Stefan Huelbrock <shuelbrock%datasystems.de>
- Alexander W. Janssen <yalla%ynfonatic.de>
- Gareth John <gdjohn%zepler.org>
- Martin Josefsson <gandalf%wlug.westbo.se>
- Andi Kleen <ak%suse.de>
- Andreas J. Koenig <andreas.koenig%anima.de>
- Pawel Krawczyk <kravietz%alfa.ceti.pl>
- Amit Kucheria <amitk@ittc.ku.edu>
- Edmund Lau <edlau%ucf.ics.uci.edu>
- Philippe Latu <philippe.latu%linux–france.org>
- Arthur van Leeuwen <arthurvl%sci.kun.nl>
- Jason Lunz <j@cc.gatech.edu>
- Stuart Lynne <sl@fireplug.net>
- Alexey Mahotkin <alexm@formulabez.ru>
- Predrag Malicevic <pmalic@ieee.org>
- Patrick McHardy <kaber@trash.net>
- Andreas Mohr <andi%lisas.de>
- Andrew Morton <akpm@zip.com.au>
- Wim van der Most
- Stephan Mueller <smueller@chronox.de>
- Togan Muftuoglu <toganm%yahoo.com>
- Chris Murray <cmurray@stargate.ca>
- Patrick Nagelschmidt <dto%gmx.net>
- Ram Narula <ram@princess1.net>
- Jorge Novo <jnovo@educanet.net>
- Patrik <ph@kurd.nu>

- P?l Osgy?ny <oplab%westel900.net>
- Lutz Preßler <Lutz.Pressler%SerNet.DE>
- Jason Pyeron <jason%pyeron.com>
- Rusty Russell <rusty%rustcorp.com.au>
- Mihai RUSU <dizzy%roedu.net>
- Jamal Hadi Salim <hadi%cyberus.ca>
- David Sauer <davids%penguin.cz>
- Sheharyar Suleman Shaikh <sss23@drexel.edu>
- Stewart Shields <MourningBlade%bigfoot.com>
- Nick Silberstein <nhsilber%yahoo.com>
- Konrads Smelkov <konrads@interbaltika.com>
- William Stearns

  <wstearns@pobox.com>
- Andreas Steinmetz <ast%domdv.de>
- Jason Tackaberry <tack@linux.com>
- Charles Tassell <ctassell%isn.net>
- Glen Turner <glen.turner%aarnet.edu.au>
- Tea Sponsor: Eric Veldhuyzen <eric%terra.nu>
- Song Wang <wsong@ece.uci.edu>
- Lazar Yanackiev

  <Lyanackiev%gmx.net>