Bandwidth Management & Optimization

Opensource Bandwidth Solutions

27 Feb – 03 March '06

Nairobi, Kenya

Facilitator:
Nigel Kukard, Phd CompSc
<nkukard@lbsd.net>

INASP International Network for the Availability of Scientific Publications

TENET Tertiary Education Network

Table of Contents

Introduction

Bandwidth management in todays world is a crucial part of any organization, with the costs of bandwidth being as high as they are in some countries, ISP's and businesses cannot afford to shove a router in on their network, connect it to the net and hope for the best.

Network administrators must draft policies regarding the desired quality of service they wish to provide to their organization, starting with those protocols that require priority traffic right down to those that if they work, you're lucky.

The next step is to choose a product that enforces these policies. There are many commercial products available ranging from sub $1,000 USD right up to so called “high end” solutions which far exceed $10,000 USD. You might ask ... What do these products achieve which one cannot achieve using opensource software? This is my question to you.

After reading this document one should have a much broader understanding of the power which lies in opensource and the goals which one can achieve by combining products together to create an overall solution.

Opensource Bandwidth Management Solutions

There are many solutions available today, these fall into 3 categories...

- HTTP/FTP (Specialized) proxying/caching, such as squid
- DNS caching, such as ISC Bind & djbdns
- SOCKS proxy, Dante
- Kernel based

Squid – HTTP/FTP Proxy

Squid is...

- Full-featured Web proxy cache
- Designed to run on Unix systems
- Free, open-source software

Squid supports...

- Proxying and caching of HTTP, FTP, and other URLs
- Proxying for SSL
- Cache hierarchies
- ICP, HTCP, CARP, Cache Digests
- Transparent caching
- WCCP (Squid v2.3 and above)
- Extensive access controls
- HTTP server acceleration
- SNMP
- Caching of DNS lookups

ISC Bind (named)

Features & drawbacks:

- DNS Security
  - DNSSEC (signed zones)
  - TSIG (signed DNS requests)

- IP version 6
  - Answers DNS queries on IPv6 sockets
  - IPv6 resource records (A6, DNAME, etc.)
  - Experimental IPv6 Resolver Library
- DNS Protocol Enhancements
  - IXFR, DDNS, Notify, EDNS0
  - Improved standards conformance
- Views
  - One server process can provide multiple "views" of the DNS namespace, e.g. an "inside" view to certain clients, and an "outside" view to others.
- Multiprocessor Support
- Improved Portability Architecture
- Supports limiting of memory consumption, but utilizes a minimum of about 32Mb of memory

DJBDNS

Features & drawbacks:

- djbdns is not a single, monolithic program
- Run under separate user accounts
- Supports limiting of memory consumption and uses a minimum of about 4Mb of memory.
- DJB code is program-friendly configuration files
- djbdns is "almost free," in that you can download it, compile it, and redistribute it. What you can not do, however, is distribute a version that is changed in any way. There are those who are not happy with that restriction, and it does tend to frustrate the free software development dynamic. What people *can* do is distribute patches.
- One site reported receiving 500 queries per second per server at peak times for data from a 350-megabyte data.cdb. The tinydns process handled about 7000 queries per second of CPU time. The CPU was a Pentium III-550.

Dante – SOCKS

Highlights

Some key highlights of Dante include:

- Designed with a emphasis on security and scalability.
- Distributed with a liberal license (BSD/CMU-type)
- Multi-layer access controls.
- Allows server applications to be socksified.
- Can socksify most programs at runtime without requiring recompilation.
- Interaction with libwrap (tcp wrappers).
- Can spawn external programs and provide them with endpoint information.
- Bandwidth usage control (via module *Bandwidth*).
- Port/redirection control (via module *Redirect*).
- Supports server-chaining, currently for TCP Connect.
- Control over number of client sessions (via module *Session*).

Dante module - Redirect

The *Redirect module* gives control over both where clients requests and replies will end up, aswell as what addresses and portranges the Dante server will use.

It can be used to redirect clients connections from one address to another, useful for cases where you for instance want clients to be automatically sent to a different address from what they intended.

It can be used to restrict the portranges used by the Dante server, useful for cases where a firewall needs to know what portranges the Dante server will use.

Price: EUR 200

Dante module - Bandwidth control

The *Bandwidth module* gives control over how much bandwidth the Dante server uses on behalf of the different clients.

It can be used to limit bandwidth to non-work related web/ftp sites, or to prevent ftp-related traffic from impacting too much on interactive telnet/ssh traffic.
It can also be used to give more bandwidth to certain clients or for traffic to certain sites.

In addition, when using the Dante bind extension, it can be used to provide bandwidth control to network servers (like e.g. webservers) that do not support bandwidth control internally.

Price: EUR 400

Traffic Shaper (tc)

Linux even goes far beyond what Frame and ATM provide.

Just to prevent confusion, **tc** uses the following rules for bandwidth 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 => byte/s

Queues and Queuing Disciplines explained

With queuing we determine the way in which data is *SENT*. It is important to realize that tc 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 bandwidth. We need to 'own the queue' so to speak, and be the slowest link in the chain. Luckily this is easily possible.

Simple, classless Queuing Disciplines

As said, with queuing disciplines, we change the way data is sent. Classless queuing 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 queuing before we go on 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.

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.

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:

0	1	2	3	4	5	6	7
PRECEDENCE			TOS				MBZ

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

Binary	***Decimal***	***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 -vv 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

TOS	Bits	Means	Linux Priority	Band
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.

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:

Service		***TOS Bits***	***Meaning***
TELNET		1000	minimize delay
FTP	Control	1000	minimize delay
	Data	0100	maximize throughput
TFTP		1000	minimize delay

Service		***TOS Bits***	***Meaning***
SMTP	Command phase	1000	minimize delay
	DATA phase	0100	maximize throughput
DNS	UDP Query	1000	minimize delay
	TCP Query	0000	
	Zone Transfer	0100	maximize throughput
NNTP		0001	minimize monetary cost
ICMP		0000	

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!

Token Bucket Filter (TBF)

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 at 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.

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 the 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).

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.

Stochastic Fairness Queuing (SFQ)

Stochastic Fairness Queuing (SFQ) is a simple implementation of the fair queuing 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 behavior 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!

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!

limit
: The total number of packets that will be queued by this SFQ (after that it starts dropping them).

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.

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 later chapters.

- 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 Queuing.

- If you have a big backbone and know what you are doing, consider Random Early Drop.

- 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.

Terminology

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

Queuing Discipline (qdisc)

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

root qdisc

The root qdisc is the qdisc attached to the device.

Classless qdisc

A qdisc with no configurable internal subdivisions.

Classful qdisc

A classful qdisc contains multiple classes. Some 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, each of which is internal to the qdisc. A class, in turn, may have several classes added to it. So a class can have a qdisc as parent or an other class. A leaf class is a class with no child classes. This class has 1 qdisc attached to it. This qdisc is responsible to send the data from that class. When you create a class, a fifo qdisc is attached to it. When you add a child class, this qdisc is removed. For a leaf class, this fifo qdisc can be replaced with an other more suitable qdisc. You can even replace this fifo qdisc with a classful qdisc so you can add extra classes.

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 adapter 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 pass a packet, even though they have one available.

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

```
              Userspace programs
                   ^
                   |
   +---------------+-----------------------------------------+
   |               Y                                         |
   |    -------> IP Stack                                    |
   |   |              |                                      |
   |   |              Y                                      |
   |   |              Y                                      |
   |   ^              |                                      |
   |   |  / ----------> Forwarding ->                        |
   |   ^ /                           |                       |
   |   |/                            Y                       |
   |   |                             |                       |
   |   ^                             Y          /-qdisc1-\   |
   |   |                            Egress     /--qdisc2--\  |
--->->Ingress                       Classifier ---qdisc3---- | ->
   |   Qdisc                                   \__qdisc4__/  |
   |                                            \-qdiscN_/   |
   |                                                         |
   +----------------------------------------------------------+
```

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 'enqueuing'.

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 adapter has both ingress and egress hooks.

Classful Queuing 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 Queuing' and it is so widely mentioned that people identify queuing 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.

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.

The qdisc family: roots, handles, siblings and parents

Each interface has one egress 'root qdisc'. By default, it is the earlier mentioned classless pfifo_fast queueing discipline. Each qdisc and class is 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 qdisc , which polices traffic coming in.

The handles of these qdiscs consist of two parts, a major number and a minor number : <major>:<minor>. It is customary 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. This major number must be unique within a egress or ingress setup. The minor number must be unique within a qdisc and his classes.

How filters are used to classify traffic

Recapping, a typical hierarchy might look like this:

```
                     1:   root qdisc
                      |
                     1:1    child class
                   /  |  \
                  /   |   \
                 /    |    \
                 /    |    \
              1:10  1:11  1:12   child classes
               |      |     |
               |     11:    |    leaf class
               |            |
               10:         12:   qdisc
              /   \       /   \
           10:1  10:2   12:1  12:2   leaf classes
```

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 -> 1:12 -> 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.

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:, each of which queries its siblings, and tries 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.

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. You can also add an other qdisc to the 3 predefined classes, 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 latter holds for almost all cable modems and DSL devices.

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

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.

Sample configuration

We will create this tree:

```
          1:   root qdisc
         / | \
       /   |   \
       /   |   \
     1:1  1:2  1:3    classes
      |    |    |
     10:  20:  30:    qdiscs    qdiscs
     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 nkukard@10.0.0.11:./
nkukard@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.

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 tortuous 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.

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. For Unix kernels, however, it is generally hard to schedule 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.

CBQ classful behavior

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-numbered 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 higher 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!

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 not 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.

Sample configuration

```
               1:            root qdisc
               |
              1:1            child class
             /   \
            /     \
          1:3     1:4        leaf classes
           |       |
          30:     40:        qdiscs
         (sfq)   (sfq)
```

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 1:1 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 leaf 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 qdisc, 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.

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. 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 and-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.

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

TC_PRIO	Num	Corresponds to TOS
BESTEFFOR	0	Maximize Reliability
FILLER	1	Minimize Cost
BULK	2	Maximize Throughput (0x8)
INTERACTIVE_BULK	4	
INTERACTIV	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 at 1:0 now looks like this:

priority	send to
0	1:2
1	1:3
2	1:3
3	1:3

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

Hierarchical Token Bucket (HTB)

The HTB 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.

As your HTB configuration gets more complex, your configuration scales well. With CBQ it is already complex even in simple cases.

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
```

Its recommended that SFQ be added 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.

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 adapter!

Some simple filtering examples

As explained in the Classifier chapter, you can match on literally anything, using a very complicated syntax. To start, I'll show you 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
```

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', destination: 'match ip dport 80 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 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.

On fwmark

You can mark packets with iptables 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.

The Intermediate queuing 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, ...

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 honors the fact that packets dropped by the filter table won't occupy bandwidth.

Advanced filters for (re-)classifying packets

As explained in the section on classful queuing 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.

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 packets 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.

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 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.

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.

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:

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 64 0xff  \
              at 8 flowid 1:4
```

The following matches all TCP packets which have the ACK bit set:

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 match ip protocol  \
              6 0xff match u8 0x10 0xff at nexthdr+13 flowid 1:3
```

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.

The filter below is a modified version of the filter above. The difference is, that it doesn't check the ip header length. Why? Because the filter above does only work on 32 bit systems.

```
# tc filter add dev ppp14 parent 1:0 protocol ip prio 10 u32 \
    match ip protocol 6 0xff match u8 0x10 0xff at nexthdr+13 \
     match u16 0x0000 0xffc0 at 2 flowid 1:3
```

The route classifier

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

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

Here we add a route classifier onto the parent node 1:0 with priority 100. When a packet reaches this node (which, since it is the root, will happen immediately) it will consult the routing table. If the packet matches, it will be send to the given class and have 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 matches the packets going to the network 192.168.10.0. 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.

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 either to entirely cease matching above a certain rate, or not to 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.

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.

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.

With Token Bucket Filter

Uses the following parameters:

- burst/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 hold it in order to delay it.

Overlimit actions

If your filter decides that it is overlimit, it can take 'actions'. Currently, four 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.

Examples

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

Limit incoming icmp traffic to 2kbit, drop packets over the limit:

```
# tc filter add dev $DEV parent ffff: protocol ip prio 20 \
          u32 match ip protocol 1 0xff police rate 2kbit buffer  \
          10k drop flowid :1
```

Limit packets to a certain size (i.e. all packets with a length greater than 84 bytes will get dropped):

```
# tc filter add dev $DEV parent ffff: protocol ip prio 20 u32 match  \
          tos 0 0 police mtu 84 drop flowid :1
```

This method can be used to drop all packets:

```
# tc filter add dev $DEV parent ffff: protocol ip prio 20 u32 match  \
          ip protocol 1 0xff police mtu 1 drop flowid :1
```

It actually drops icmp packets greater-than 1 byte. While packets with a size of 1 byte are possible in theory, you will not find these in a real network.

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

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 will be sent 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!

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.

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.

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.

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.

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
```

BWM Tools

Introduction

Bandwidth Management Tools was designed to provide a full suite of bandwidth management applications, able to shape, log and graph traffic.

BWM Tools intercepts packets using the QUEUE mechanism, when packets arrive they are queued. Another thread picks these packets up and processes them according to the flows defined.

Seeing as BWM Tools uses iptables for matching traffic, the control over traffic is limitless.

BWM Tools is a set of userspace utilities, no kernel patches are required. As long as your iptables supports the -j QUEUE target, traffic shaping will work.

Features

This section lists a few features which make BWM Tools a good solution for small to large enterprises...

Traffic Shaping

- Class based traffic categorization.
- Hierarchical flows allows you to embed flows within flows to form complex traffic shaping setups.
- Parent burst thresholds, this allows child flows to burst until their parent flow has reached a specific utilization threshold.
- Realtime flow monitoring

Firewalling

- Support for all IPTables/Netfilter features on host operating system.

Graphing

- RRD Tool file generation which can be used to create custom graphs.
- Builtin RRD Tool graphing support allowing BWM Tools to generate pretty looking graphs all by itself.

Logging

- Logging of traffic stats to file at pre-defined intervals for use in reporting or graphing.
- Logging of byte, packet, burst & drop counters.
- Traffic flow grouping.

Seeing as BWM Tools is based largely on the functionality of IPTables/Netfilter, we going to touch on this subject below.

IPTables

What is Netfilter/Iptables?

Netfilter is the framework in Linux 2.4+ kernels that allow for firewalling, NAT, and packet mangling. Iptables is the userspace tools that works with the Netfilter framework (technically a lie; Iptables is also a part of the Netfilter framework in the kernel). Think of Netfilter as kernel space, and Iptables as userspace.

Interesting Netfilter features...

- State matching - Connection tracking (can you trust the remote host to determine whether your firewall will accept a packet?).
- Automatic fragmentation reassembly - Connection tracking automatically reassembles fragmented packets for examination.
- Improved matching - Advanced packet matching such as rate limit, string matching (packet data), etc.
- Improved logging - Customized logging levels and entries, also allows user space logging.
- Allows packet mangling - Allows for the mangling of any information inside a packet.
- Userspace queuing - Allows userspace programs access to packets.
- Built-in support for port forwarding - obviates IPMASQADM.
- Progress - Inexorable fact of life.

There are several different things you can do with iptables. You start with three built-in chains INPUT, OUTPUT and FORWARD which you can't delete. Let's look at the operations to manage whole chains:

1. Create a new chain (-N)
2. Delete an empty chain (-X)
3. Change the policy for a built-in chain. (-P)
4. List the rules in a chain (-L)
5. Flush the rules out of a chain (-F)
6. Zero the packet and byte counters on all rules in a chain (-Z)

There are several ways to manipulate rules inside a chain:

1. Append a new rule to a chain (-A)
2. Insert a new rule at some position in a chain (-I)
3. Replace a rule at some position in a chain (-R)
4. Delete a rule at some position in a chain, or the first that matches (-D)

Managing single rules

This is the bread-and-butter of packet filtering; manipulating rules. Most commonly, you will probably use the append (-A) and delete (-D) commands. The others (-I for insert and -R for replace) are simple extensions of these concepts.

Each rule specifies a set of conditions the packet must meet, and what to do if it meets them (a `target'). For example, you might want to drop all ICMP packets coming from the IP address 127.0.0.1. So in this case our conditions are that the protocol must be ICMP and that the source address must be 127.0.0.1. Our target is `DROP'.

127.0.0.1 is the `loopback' interface, which you will have even if you have no real network connection. You can use the `ping' program to generate such packets (it simply sends an ICMP type 8 (echo request) which all cooperative hosts should obligingly respond to with an ICMP type 0 (echo reply) packet). This makes it useful for testing.

```
# ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.2 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.2/0.2 ms

# iptables -A INPUT -s 127.0.0.1 -p icmp -j DROP
# ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
```

You can see here that the first ping succeeds (the `-c 1' tells ping to only send a single packet).

Then we append (-A) to the `INPUT' chain, a rule specifying that for packets from 127.0.0.1 (`-s 127.0.0.1') with protocol ICMP (`-p icmp') we should jump to DROP (`-j DROP').

Then we test our rule, using the second ping. There will be a pause before the program gives up waiting for a response that will never come.

We can delete the rule in one of two ways. Firstly, since we know that it is the only rule in the input chain, we can use a numbered delete, as in:

# iptables -D INPUT 1

To delete rule number 1 in the INPUT chain.

The second way is to mirror the -A command, but replacing the -A with -D. This is useful when you have a complex chain of rules and you don't want to have to count them to figure out that it's rule 37 that you want to get rid of. In this case, we would use:

# iptables -D INPUT -s 127.0.0.1 -p icmp -j DROP

The syntax of -D must have exactly the same options as the -A (or -I or -R) command. If there are multiple identical rules in the same chain, only the first will be deleted.

Filtering Specifications

We have seen the use of `-p' to specify protocol, and `-s' to specify source address, but there are other options we can use to specify packet characteristics. What follows is an exhaustive compendium.

Specifying Source and Destination IP Addresses

Source (`-s', `--source' or `--src') and destination (`-d', `--destination' or `--dst') IP addresses can be specified in four ways. The most common way is to use the full name, such as `localhost' or `www.linuxhq.com'. The second way is to specify the IP address such as `127.0.0.1'.

The third and fourth ways allow specification of a group of IP addresses, such as `199.95.207.0/24' or `199.95.207.0/255.255.255.0'. These both specify any IP address from 199.95.207.0 to 199.95.207.255 inclusive; the digits after the `/' tell which parts of the IP address are significant. `/32' or `/255.255.255.255' is the default (match all of the IP address). To specify any IP address at all `/0' can be used, like so:

[NOTE: `-s 0/0' is redundant here.]
# iptables -A INPUT -s 0/0 -j DROP

This is rarely used, as the effect above is the same as not specifying the `-s' option at all.

Specifying Inversion

Many flags, including the `-s' (or `--source') and `-d' (`--destination') flags can have their arguments preceded by `!' (pronounced `not') to match addresses NOT equal to the ones given. For example. `-s ! localhost' matches any packet not coming from localhost.

Specifying Protocol

The protocol can be specified with the `-p' (or `--protocol') flag. Protocol can be a number (if you know the numeric protocol values for IP) or a name for the special cases of `TCP', `UDP' or `ICMP'. Case doesn't matter, so `tcp' works as well as `TCP'.

The protocol name can be prefixed by a `!', to invert it, such as `-p ! TCP' to specify packets which are not TCP.

Specifying an Interface

The `-i' (or `--in-interface') and `-o' (or `--out-interface') options specify the name of an interface to match. An interface is the physical device the packet came in on (`-i') or is going out on (`-o'). You can use the ifconfig command to list the interfaces which are `up' (i.e., working at the moment).

Packets traversing the INPUT chain don't have an output interface, so any rule using `-o' in this chain will never match. Similarly, packets traversing the OUTPUT chain don't have an input interface, so any rule using `-i' in this chain will never match.

Only packets traversing the FORWARD chain have both an input and output interface.

It is perfectly legal to specify an interface that currently does not exist; the rule will not match anything until the interface comes up. This is extremely useful for dial-up PPP links (usually interface ppp0) and the like.

As a special case, an interface name ending with a `+' will match all interfaces (whether they currently exist or not) which begin with that string. For example, to specify a rule which matches all PPP interfaces, the -i ppp+ option would be used.

The interface name can be preceded by a `!' with spaces around it, to match a packet which does not match the specified interface(s), eg -i ! ppp+.

Specifying Fragments

Sometimes a packet is too large to fit down a wire all at once. When this happens, the packet is divided into fragments, and sent as multiple packets. The other end reassembles these fragments to reconstruct the whole packet.

The problem with fragments is that the initial fragment has the complete header fields (IP + TCP, UDP and ICMP) to examine, but subsequent packets only have a subset of the headers (IP without the additional protocol fields). Thus looking inside subsequent fragments for protocol headers (such as is done by the TCP, UDP and ICMP extensions) is not possible.

If you are doing connection tracking or NAT, then all fragments will get merged back together before they reach the packet filtering code, so you need never worry about fragments.

Please also note that in the INPUT chain of the filter table (or any other table hooking into the NF_IP_LOCAL_IN hook) is traversed after defragmentation of the core IP stack.

Otherwise, it is important to understand how fragments get treated by the filtering rules. Any filtering rule that asks for information we don't have will *not* match. This means that the first fragment is treated like any other packet. Second and further fragments won't be. Thus a rule -p TCP --sport www (specifying a source port of `www') will never match a fragment (other than the first fragment). Neither will the opposite rule -p TCP --sport ! www.

However, you can specify a rule specifically for second and further fragments, using the `-f' (or `--fragment') flag. It is also legal to specify that a rule does *not* apply to second and further fragments, by preceding the `-f' with ` ! '.

Usually it is regarded as safe to let second and further fragments through, since filtering will effect the first fragment, and thus prevent reassembly on the target host; however, bugs have been known to allow crashing of machines simply by sending fragments. Your call.

Note for network-heads: malformed packets (TCP, UDP and ICMP packets too short for the firewalling code to read the ports or ICMP code and type) are dropped when such examinations are attempted. So are TCP fragments starting at position 8.

As an example, the following rule will drop any fragments going to 192.168.1.1:

```
# iptables -A OUTPUT -f -d 192.168.1.1 -j DROP
```

Extensions to iptables: Matches

IPTables is extensible, meaning that both the kernel and the iptables tool can be extended to provide new features.

Some of these extensions are standard, and other are more exotic. Extensions can be made by other people and distributed separately for niche users.

Kernel extensions normally live in the kernel module subdirectory, they are demand loaded if your kernel was compiled with CONFIG_KMOD set, so you should not need to manually insert them.

Extensions to the iptables program are shared libraries which usually live in /usr/lib/iptables.

Extensions come in two types: new targets, and new matches (we'll talk about new targets a little later). Some protocols automatically offer new tests:

currently these are TCP, UDP and ICMP as shown below.

For these you will be able to specify the new tests on the command line after the `-p' option, which will load the extension. For explicit new tests, use the `-m' option to load the extension, after which the extended options will be available.

To get help on an extension, use the option to load it (`-p', `-j' or `-m') followed by `-h' or `--help', eg:

# iptables -p tcp –help

Match Extensions

IPTables can use extended packet matching modules. These are loaded in two ways: implicitly, when -p or --protocol is specified, or with the -m or --match options, followed by the matching module name; after these, various extra command line options become available, depending on the specific module. You can specify multiple extended match modules in one line, and you can use the -h or --help options after the module has been specified to receive help specific to that module.

The following are included in the base package, and most of these can be preceded by a **!** to invert the sense of the match.

addrtype

This module matches packets based on their address type. Address types are used within the kernel networking stack and categorize addresses into various groups. The exact definition of that group depends on the specific layer three protocol.

The following address types are possible:

- **UNSPEC**
  an unspecified address (i.e. 0.0.0.0)
- **UNICAST**
  an unicast address
- **LOCAL**
  a local address
- **BROADCAST**

a broadcast address

- **ANYCAST**
  an anycast packet
- **MULTICAST**
  a multicast address
- **BLACKHOLE**
  a blackhole address
- **UNREACHABLE**
  an unreachable address
- **PROHIBIT** a prohibited address

--src-type *type*
Matches if the source address is of given type

--dst-type *type*
Matches if the destination address is of given type

ah

``ah'' : lets you match an AH packet based on its Security Parameter Index (SPI). This module matches the SPIs in AH header of IPSec packets.

--ahspi [!] *spi*[:*spi*]
match spi (range)

For example, we will drop all the AH packets that have a SPI equal to 500:

```
# iptables -A INPUT -p 51 -m ah --ahspi 500 -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       ipv6-auth--  anywhere             anywhere           ah spi:500
```

esp

``esp'' : lets you match an ESP packet based on its SPI. This module matches the SPIs in ESP header of IPSec packets.

--espspi [!] *spi*[:*spi*]

The esp match works exactly the same:

```
# iptables -A INPUT -p 50 -m esp --espspi 500 -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       ipv6-crypt--  anywhere             anywhere          esp
spi:500
```

Do not forget to specify the proper protocol through ``-p 50'' or ``-p 51'' (for esp & ah respectively) when you use the ah or esp matches, or else the rule insertion will simply abort for obvious reasons.

childlevel

This is an experimental module. It matches on whether the packet is part of a master connection or one of its children (or grandchildren, etc). For instance, most packets are level 0. FTP data transfer is level 1.

--childlevel [!] *level*

condition

This matches if a specific /proc filename is '0' or '1'.

--condition *[!] filename*

Match on boolean value stored in /proc/net/ipt_condition/filename file

For example, if you want to prohibit access to your web server while doing maintenance, you can use the following:

```
# iptables -A FORWARD -p tcp -d 192.168.1.10 --dport http -m
condition --condition webdown -j REJECT --reject-with tcp-reset
# echo 1 > /proc/net/ipt_condition/webdown
```

The following rule will match only if the ``webdown'' condition is set to “1”.

Notes:

- The condition variables are stored in the `/proc/net/ipt_condition/' directory.
- A condition variable can only be set to ``0'' (FALSE) or ``1'' (TRUE).

- One or many rules can be affected by the state of a single condition variable.
- A condition proc file is automatically created when a new condition is first referenced.
- A condition proc file is automatically deleted when the last reference to it is removed.

connmark

This module matches the netfilter mark field associated with a connection (which can be set using the CONNMARK target below).

--mark *value[/mask]*

Matches packets in connections with the given mark value (if a mask is specified, this is logically ANDed with the mark before the comparison).

connrate

This module matches the current transfer rate in a connection.

--connrate *[!] [from*]:[*to*]

Match against the current connection transfer rate being within 'from' and 'to' bytes per second. When the "!" argument is used before the range, the sense of the match is inverted.

conntrack

This module, when combined with connection tracking, allows access to more connection tracking information than the "state" match. (this module is present only if iptables was compiled under a kernel supporting this feature)

--ctstate *state*

Where state is a comma separated list of the connection states to match. Possible states are...

INVALID

Meaning that the packet is associated with no known connection.

ESTABLISHED

Meaning that the packet is associated with a connection which has seen packets in both directions.

NEW
Meaning that the packet has started a new connection, or otherwise associated with a connection which has not seen packets in both directions.

RELATED
Meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error.

SNAT
A virtual state, matching if the original source address differs from the reply destination.

DNAT
A virtual state, matching if the original destination differs from the reply source.

--ctproto *proto*
Protocol to match (by number or name)

--ctorigsrc *[!] address*[/*mask*]
Match against original source address

--ctorigdst *[!] address*[/*mask*]
Match against original destination address

--ctreplsrc *[!] address*[/*mask*]
Match against reply source address

--ctrepldst *[!] address***[/***mask*]
Match against reply destination address

--ctstatus *[NONE|EXPECTED|SEEN_REPLY|ASSURED*][,...]
Match against internal conntrack states

--ctexpire *time*[*:time*]
Match remaining lifetime in seconds against given value or range of values (inclusive)

For example, if you want to allow all the RELATED connections for TCP protocols only, then you can proceed as follows :

```
# iptables -A FORWARD -m conntrack --ctstate RELATED  \
        --ctproto tcp -j ACCEPT
# iptables –list
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere           ctstate RELATED
```

dscp

This module matches the 6 bit DSCP field within the TOS field in the IP header. DSCP has superseded TOS within the IETF.

--dscp *value*

Match against a numeric (decimal or hex) value [0-32].

--dscp-class *DiffServ Class*

Match the DiffServ class. This value may be any of the BE, EF, AFxx or CSx classes. It will then be converted into it's according numeric value.

dstlimit

This module allows you to limit the packet per second (pps) rate on a per destination IP or per destination port base. As opposed to the `limit' match, every destination ip / destination port has it's own limit.

--dstlimit *avg*

Maximum average match rate (packets per second unless followed by /sec /minute /hour /day postfixes).

--dstlimit-mode *mode*

The limiting hashmode. Is the specified limit per **dstip, dstip-dstport** tuple, **srcip-dstip** tuple, or per **srcipdstip-dstport** tuple.

--dstlimit-name *name*

Name for /proc/net/ipt_dstlimit/* file entry

[***--dstlimit-burst*** burst*]*

Number of packets to match in a burst. Default: 5

[***--dstlimit-htable-size*** size*]*

Number of buckets in the hashtable

[***--dstlimit-htable-max*** max*]*

Maximum number of entries in the hashtable

[***--dstlimit-htable-gcinterval*** interval*]*

Interval between garbage collection runs of the hashtable (in miliseconds). Default is 1000 (1 second).

[***--dstlimit-htable-expire*** time]

After which time are idle entries expired from hashtable (in miliseconds)? Default is 10000 (10 seconds).

ecn

This allows you to match the ECN bits of the IPv4 and TCP header. ECN is the Explicit Congestion Notification mechanism as specified in RFC3168

--ecn-tcp-cwr

This matches if the TCP ECN CWR (Congestion Window Received) bit is set.

--ecn-tcp-ece

This matches if the TCP ECN ECE (ECN Echo) bit is set.

--ecn-ip-ect *num*

This matches a particular IPv4 ECT (ECN-Capable Transport). You have to specify a number between `0' and `3'.

fuzzy

This match implements a TSK FLC (Takagi-Sugeno-Kang Fuzzy Logic Controller). The basic idea is that the match is given two parameters that tell it the desired filtering interval.

When the packet rate is below `lower-limit' the rule will never match. Between `lower-limit' and `upper-limit', matching will occurs according a increasing (mean) rate.

Finally, when the packet rate comes to `upper-limit', (mean) matching rate attains its maximum value, 99%.

Taking into account that the sampling rate is variable and is of approximately 100ms (on a busy machine), the author believes that the module presents good responsiveness, adapting fast to changing traffic patterns.

--lower-limit number
Specifies the lower limit (in packets per second).

--upper-limit *number*
Specifies the upper limit (in packets per second).

For example, if you wish to avoid Denials Of Service, you could use the following rule:

```
# iptables -A INPUT -m fuzzy --lower-limit 100 --upper-limit 1000  \
             -j REJECT
```

Below the 100 pps (packets per second) rate, the filter is inactive.
Between 100 and 1000 pps the mean acceptance rate drops from 100% (when we are at 100 pps) to 1% (when we are at 1000 pps).
Above 1000 pps the acceptance rate keeps constant at 1%.

helper
This module matches packets related to a specific conntrack-helper.

--helper *string*
Matches packets related to the specified conntrack-helper.

string can be "ftp" for packets related to a ftp-session on default port. For other ports append -portnr to the value, ie. "ftp-2121".

Same rules apply for other conntrack-helpers.

icmp
This extension is loaded if `--protocol icmp' is specified. It provides the following option:

--icmp-type [!] *typename*
This allows specification of the ICMP type, which can be a numeric ICMP type, or one of the ICMP type names below.

Valid ICMP Types:

any
echo-reply (pong)
destination-unreachable
   network-unreachable
   host-unreachable
   protocol-unreachable
   port-unreachable
   fragmentation-needed
   source-route-failed
   network-unknown
   host-unknown
   network-prohibited
   host-prohibited
   TOS-network-unreachable
   TOS-host-unreachable
   communication-prohibited
   host-precedence-violation
   precedence-cutoff
source-quench
redirect
   network-redirect
   host-redirect
   TOS-network-redirect
   TOS-host-redirect
echo-request (ping)
router-advertisement
router-solicitation
time-exceeded (ttl-exceeded)
   ttl-zero-during-transit
   ttl-zero-during-reassembly
parameter-problem
   ip-header-bad
   required-option-missing
timestamp-request
timestamp-reply
address-mask-request
address-mask-reply

iprange

This matches on a given arbitrary range of IPv4 addresses

[!]***--src-range*** ip-ip

Match source IP in the specified range.

[!]***--dst-range*** ip-ip
Match destination IP in the specified range.

length

This module matches the length of a packet against a specific value or range of values.

--length *length*[:*length*]

For example, let's drop all the pings with a packet size greater than 85 bytes :

```
# iptables -A INPUT -p icmp --icmp-type echo-request -m length  \
               --length 86:0xffff -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       icmp --  anywhere             anywhere           icmp echo-request
length 86:65535
```

Values of the range not present will be implied. The implied value for minimum is 0, and for maximum is 65535.

limit

This module matches at a limited rate using a token bucket filter. A rule using this extension will match until this limit is reached (unless the `!' flag is used). It can be used in combination with the LOG target to give limited logging, for example.

--limit *rate*
Maximum average matching rate: specified as a number, with an optional `/second', `/minute', `/hour', or `/day' suffix; the default is 3/hour.

--limit-burst *number*
Maximum initial number of packets to match: this number gets recharged by one every time the limit specified above is not reached, up to this number; the default is 5.

mac

--mac-source [!] *address*

Match source MAC address. It must be of the form XX:XX:XX:XX:XX:XX. Note that this only makes sense for packets coming from an Ethernet device and entering the PREROUTING, FORWARD or INPUT chains.

mark

This module matches the netfilter mark field associated with a packet (which can be set using the MARK target below).

--mark *value*[/*mask*]

Matches packets with the given unsigned mark value (if a mask is specified, this is logically ANDed with the mask before the comparison).

mport

This module matches a set of source or destination ports. Up to 15 ports can be specified. It can only be used in conjunction with -p tcp or -p udp.

--source-ports *port*[,*port*[,*port*...]]

Match if the source port is one of the given ports. The flag --sports is a convenient alias for this option.

--destination-ports *port*[,*port*[,*port*...]]

Match if the destination port is one of the given ports. The flag --dports is a convenient alias for this option.

--ports *port*[,*port*[,*port*...]]

Match if the both the source and destination ports are equal to each other and to one of the given ports.

For example, if you want to block ftp, ssh, telnet and http in one line, you can:

```
# iptables -A INPUT -p tcp -m mport --ports 20:23,80 -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       tcp  --  anywhere             anywhere           mport ports ftp-
data:telnet,http
```

multiport

This module matches a set of source or destination ports. Up to 15 ports can be specified. It can only be used in conjunction with -p tcp or -p udp.

--source-ports *port*[*,port*[*,port...*]]
: Match if the source port is one of the given ports. The flag --sports is a convenient alias for this option.

--destination-ports *port*[*,port*[*,port...*]]
: Match if the destination port is one of the given ports. The flag --dports is a convenient alias for this option.

--ports *port*[*,port*[*,port...*]]
: Match if the both the source and destination ports are equal to each other and to one of the given ports.

nth

This module matches every `n'th packet

--every *value*
: Match every `value' packet

[***--counter*** num*]*
: Use internal counter number `num'. Default is `0'.

[***--start*** num*]*
: Initialize the counter at the number `num' insetad of `0'. Most between `0' and `value'-1.

[***--packet*** num*]*
: Match on `num' packet. Most be between `0' and `value'-1.

For example, if you want to drop every 2 ping packets, you can do as follows :

```
# iptables -A INPUT -p icmp --icmp-type echo-request -m nth  \
        --every 2 -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       icmp --  anywhere             anywhere           icmp echo-request
every 2th
```

For example, if you want to balance the load to the 3 addresses 10.0.0.5, 10.0.0.6 and 10.0.0.7, then you can do as follows :

```
# iptables -t nat -A POSTROUTING -o eth0 -m nth --counter 7  \
        --every 3 --packet 0 -j SNAT --to-source 10.0.0.5
# iptables -t nat -A POSTROUTING -o eth0 -m nth --counter 7  \
        --every 3 --packet 1 -j SNAT --to-source 10.0.0.6
# iptables -t nat -A POSTROUTING -o eth0 -m nth --counter 7  \
        --every 3 --packet 2 -j SNAT --to-source 10.0.0.7

# iptables -t nat --list
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       all  --  anywhere             anywhere           every 3th packet #0
to:10.0.0.5
SNAT       all  --  anywhere             anywhere           every 3th packet #1
to:10.0.0.6
SNAT       all  --  anywhere             anywhere           every 3th packet #2
to:10.0.0.7
```

owner

This module attempts to match various characteristics of the packet creator, for locally-generated packets. It is only valid in the OUTPUT chain, and even this some packets (such as ICMP ping responses) may have no owner, and hence never match.

--uid-owner *userid*
: Matches if the packet was created by a process with the given effective user id.

--gid-owner *groupid*
: Matches if the packet was created by a process with the given effective group id.

--pid-owner *processid*
: Matches if the packet was created by a process with the given process id.

--sid-owner *sessionid*
: Matches if the packet was created by a process in the given session group.

--cmd-owner *name*

Matches if the packet was created by a process with the given command name. (this option is present only if iptables was compiled under a kernel supporting this feature)

NOTE: pid, sid and command matching are broken on SMP

physdev

This module matches on the bridge port input and output devices enslaved to a bridge device. This module is a part of the infrastructure that enables a transparent bridging IP firewall and is only useful for kernel versions above version 2.5.44.

--physdev-in name

Name of a bridge port via which a packet is received (only for packets entering the INPUT, FORWARD and PREROUTING chains). If the interface name ends in a "+", then any interface which begins with this name will match. If the packet didn't arrive through a bridge device, this packet won't match this option, unless '!' is used.

--physdev-out name

Name of a bridge port via which a packet is going to be sent (for packets entering the FORWARD, OUTPUT and POSTROUTING chains). If the interface name ends in a "+", then any interface which begins with this name will match. Note that in the nat and mangle OUTPUT chains one cannot match on the bridge output port, however one can in the filter OUTPUT chain. If the packet won't leave by a bridge device or it is yet unknown what the output device will be, then the packet won't match this option, unless

--physdev-is-in

Matches if the packet has entered through a bridge interface.

--physdev-is-out

Matches if the packet will leave through a bridge interface.

--physdev-is-bridged

Matches if the packet is being bridged and therefore is not being routed. This is only useful in the FORWARD and POSTROUTING chains.

pkttype

This module matches the link-layer packet type.

--pkt-type *[unicast*|*broadcast*|*multicast*]

If for example you want to silently drop all the broadcasted packets:

```
# iptables -A INPUT -m pkttype --pkt-type broadcast -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere           PKTTYPE = broadcast
```

random

This module randomly matches a certain percentage of all packets.

--average *percent*
Matches the given percentage. If omitted, a probability of 50% is set.

For example, if you want to drop 50% of the pings randomly, you can do as follows:

```
# iptables -A INPUT -p icmp --icmp-type echo-request -m random  \
        --average 50 -j DROP
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
DROP       icmp --  anywhere     anywhere       icmp echo-request  random
50%
```

realm

This matches the routing realm. Routing realms are used in complex routing setups involving dynamic routing protocols like BGP.

--realm *[!]*value[/mask]
Matches a given realm number (and optionally mask).

For example, to log all the outgoing packet with a realm of 10, you can do the following :

```
# iptables -A OUTPUT -m realm --realm 10 -j LOG
# iptables --list
```

```
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
LOG        all  --  anywhere             anywhere           REALM match 0xa LOG
level warning
```

recent

This match allows you to dynamically create a list of IP addresses and then match against that list in a few different ways.

--name name

Specify the list to use for the commands. If no name is given then 'DEFAULT' will be used.

[!] **--set**

This will add the source address of the packet to the list. If the source address is already in the list, this will update the existing entry. This will always return success or failure if \`!' is passed in.

[!] **--rcheck**

This will check if the source address of the packet is currently in the list and return true if it is, and false otherwise. Opposite is returned if \`!' is passed in.

[!] **--update**

This will check if the source address of the packet is currently in the list. If it is then that entry will be updated and the rule will return true. If the source address is not in the list then the rule will return false. Opposite is returned if \`!' is passed in.

[!] **--remove**

This will check if the source address of the packet is currently in the list and if so that address will be removed from the list and the rule will return true. If the address is not found, false is returned. Opposite is returned if \`!' is passed in.

[!] **--seconds seconds**

This option must be used in conjunction with one of \`rcheck' or \`update'. When used, this will narrow the match to only happen when the address is in the list and was seen within the last given number of seconds. Opposite is returned if \`!' is passed in.

[!] **--hitcount hits**

This option must be used in conjunction with one of \`rcheck' or `update'. When used, this will narrow the match to only happen when the address is in the list and packets had been received greater than or equal to the given value. This option may be used along with `seconds' to create an even narrower match requiring a certain number of hits within a specific time frame. Opposite returned if `!' passed in.

--rttl

This option must be used in conjunction with one of `rcheck' or `update'. When used, this will narrow the match to only happen when the address is in the list and the TTL of the current packet matches that of the packet which hit the --set rule. This may be useful if you have problems with people faking their source address in order to DoS you via this module by disallowing others access to your site by sending bogus packets to you.

For example, you can create a `badguy' list out of people attempting to connect to port 139 on your firewall and then DROP all future packets from them without considering them.

```
# iptables -A FORWARD -m recent --name badguy --rcheck  \
              --seconds 60 -j DROP
# iptables -A FORWARD -p tcp -i eth0 --dport 139 -m recent  \
              --name badguy --set -j DROP
# iptables –list
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere          recent: CHECK
seconds:60
DROP       tcp  --  anywhere             anywhere          tcp dpt:netbios-ssn
recent: SET
```

set

This modules macthes IP sets which can be defined by ipset.

--set setname flag[,flag...]

Where flags are src and/or dst and there can be no more than six of them. Hence the command

```
# iptables -A FORWARD -m set --set test src,dst
```

This will match packets, for which (depending on the type of the set) the source address or port number of the packet can be found in the specified set. If there is a binding belonging to the mached set element or there is a default binding for the given set, then the rule will match the packet only if additionally (depending on the type of the set) the destination address or port number of the packet can be found in the set according to the binding.

state

This module, when combined with connection tracking, allows access to the connection tracking state for this packet.

--state *state*

Where state is a comma separated list of the connection states to match. Possible states are a subset of the conntrack match above... INVALID, ESTABLISHED, NEW, RELATED.

string

This match allows you to match a string anywhere in the packet.

--string [!] string

Match a string in a packet

For example, to match packets containing the string ``cmd.exe'' anywhere in the packet and queue them to a userland IDS, you could use:

```
# iptables -A INPUT -m string --string 'cmd.exe' -j QUEUE
# iptables –list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
QUEUE      all  --  anywhere             anywhere           STRING match
cmd.exe
```

Please do use this match with caution. A lot of people want to use this match to stop worms, along with the DROP target. This is a major mistake. It would be defeated by any IDS evasion method.

In a similar fashion, a lot of people have been using this match as a mean to stop particular functions in HTTP like POST or GET by dropping any HTTP packet containing the string POST. Please understand that this job is better done by a filtering proxy. Additionally, any HTML content with the word POST would get dropped with the former method. This match has been designed to be able to queue to userland interesting packets for better analysis, that's all. Dropping packet based on this would be defeated by any IDS evasion method.

tcp

These extensions are loaded if `--protocol tcp' is specified. It provides the following options:

--source-port [!] *port*[:*port*]

Source port or port range specification. This can either be a service name or a port number. An inclusive range can also be specified, using the format *port*:*port*. If the first port is omitted, "0" is assumed; if the last is omitted, "65535" is assumed. If the second port greater then the first they will be swapped. The flag --sport is a convenient alias for this option.

--destination-port [!] *port*[:*port*]

Destination port or port range specification. The flag --dport is a convenient alias for this option.

--tcp-flags [!] *mask comp*

Match when the TCP flags are as specified. The first argument is the flags which we should examine, written as a comma-separated list, and the second argument is a comma-separated list of flags which must be set. Flags are: SYN ACK FIN RST URG PSH ALL NONE.

Hence the command:

```
#  iptables -A FORWARD -p tcp --tcp-flags  SYN,ACK,FIN,RST SYN
```

will only match packets with the SYN flag set, and the ACK, FIN and RST flags unset.

[!] **--syn**

Only match TCP packets with the SYN bit set and the ACK and RST bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will prevent incoming TCP connections, but outgoing TCP connections will be unaffected. It is equivalent to --tcp-flags SYN,RST,ACK SYN. If the "!" flag precedes the "--syn", the sense of the option is inverted.

It is sometimes useful to allow TCP connections in one direction, but not the other. For example, you might want to allow connections to an external WWW server, but not connections from that server.

The naive approach would be to block TCP packets coming from the server. Unfortunately, TCP connections require packets going in both directions to work at all.

The solution is to block only the packets used to request a connection. These packets are called SYN packets (ok, technically they're packets with the SYN flag set, and the RST and ACK flags cleared, but we call them SYN packets for short). By disallowing only these packets, we can stop attempted connections in their tracks.

This flag can be inverted by preceding it with a `!', which means every packet other than the connection initiation.

--tcp-option [!] *number*
Match if TCP option set.

--mss *value*[:*value*]
Match TCP SYN or SYN/ACK packets with the specified MSS value (or range), which control the maximum packet size for that connection.

tcpmss
This matches the TCP MSS (maximum segment size) field of the TCP header. You can only use this on TCP SYN or SYN/ACK packets, since the MSS is only negotiated during the TCP handshake at connection startup time.

[!] **--mss** *value[:value]*
Match a given TCP MSS value or range.

time
This matches if the packet arrival time/date is within a given range. All options are facultative.

--timestart *value*
Match only if it is after `value' (Inclusive, format: HH:MM ; default 00:00).

--timestop *value*
Match only if it is before `value' (Inclusive, format: HH:MM ; default 23:59).

--days *listofdays*
Match only if today is one of the given days. (format: Mon,Tue,Wed,Thu,Fri,Sat,Sun ; default everyday)

--datestart *date*
Match only if it is after `date' (Inclusive, format: YYYY[:MM[:DD[:hh[:mm[:ss]]]]] ; h,m,s start from 0 ; default to 1970)

--datestop *date*
Match only if it is before `date' (Inclusive, format: YYYY[:MM[:DD[:hh[:mm[:ss]]]]] ; h,m,s start from 0 ; default to 2037)

For example, to accept packets that have an arrival time from 8:00H to 18:00H from Monday to Friday you can do as follows:

```
# iptables -A INPUT -m time --timestart 8:00 --timestop 18:00  \
              --days Mon,Tue,Wed,Thu,Fri -j ACCEPT
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source          destination
ACCEPT     all  --  anywhere        anywhere       TIME from 8:0 to 18:0 on
Mon,Tue,Wed,Thu,Fri
```

tos

This module matches the 8 bits of Type of Service field in the IP header (ie. including the precedence bits).

--tos *tos*
The argument is either a standard name or a numeric value to match.

Minimize-Delay 16 (0x10)
Maximize-Throughput 8 (0x08)
Maximize-Reliability 4 (0x04)
Minimize-Cost 2 (0x02)
Normal-Service 0 (0x00)

ttl

This module matches the time to live field in the IP header.

--ttl-eq *ttl*
Matches the given TTL value.

--ttl-gt *ttl*
Matches if TTL is greater than the given TTL value.

--ttl-lt *ttl*
Matches if TTL is less than the given TTL value.

For example if you want to log any packet that have a TTL less than 5, you can do as follows:

```
# iptables -A INPUT -m ttl --ttl-lt 5 -j LOG
# iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
LOG        all  --  anywhere             anywhere           TTL match TTL < 5 LOG
level warning
```

udp

These extensions are loaded if `--protocol udp' is specified. It provides the following options:

--source-port [!] *port*[:*port*]
Source port or port range specification. See the description of the --source-port option of the TCP extension for details.

--destination-port [!] *port*[:*port*]
Destination port or port range specification. See the description of the --destination-port option of the TCP extension for details.

unclean

This module takes no options, but attempts to match packets which seem malformed or unusual. This is regarded as experimental.

Target Specifications

Now we know what examinations we can do on a packet, we need a way of saying what to do to the packets which match our tests. This is called a rule's target.

There are two very simple built-in targets: DROP and ACCEPT. We've already met them. If a rule matches a packet and its target is one of these two, no further rules are consulted: the packet's fate has been decided.

There are two types of targets other than the built-in ones: extensions and user-defined chains.

User-defined chains

One powerful feature which iptables inherits from ipchains is the ability for the user to create new chains, in addition to the three built-in ones (INPUT, FORWARD and OUTPUT). By convention, user-defined chains are lower-case to distinguish them.

When a packet matches a rule whose target is a user-defined chain, the packet begins traversing the rules in that user-defined chain. If that chain doesn't decide the fate of the packet, then once traversal on that chain has finished, traversal resumes on the next rule in the current chain.

Time for more ASCII art. Consider two (silly) chains: INPUT (the built-in chain) and test (a user-defined chain).

```
 `INPUT'                         `test'
----------------------------    ----------------------------
| Rule1: -p ICMP -j DROP   |    | Rule1: -s 192.168.1.1    |
|--------------------------|    |--------------------------|
| Rule2: -p TCP -j test    |    | Rule2: -d 192.168.1.1    |
|--------------------------|    ----------------------------
| Rule3: -p UDP -j DROP    |
----------------------------
```

Consider a TCP packet coming from 192.168.1.1, going to 1.2.3.4. It enters the INPUT chain, and gets tested against Rule1 - no match. Rule2 matches, and its target is test, so the next rule examined is the start of test. Rule1 in test matches, but doesn't specify a target, so the next rule is examined, Rule2. This doesn't match, so we have reached the end of the chain. We return to the INPUT chain, where we had just examined Rule2, so we now examine Rule3, which doesn't match either.

So the packet path is:

```
                                v    __________________________
         `INPUT'                |   /    `test'                v
        ------------------------|--/    -----------------------|----
        | Rule1                 |  /|   | Rule1                |   |
        |-----------------------|/-|   |----------------------|---|
        | Rule2                 /  |   | Rule2                |   |
        |--------------------------|   -----------------------v----
        | Rule3                 /--+___________________________/
        ------------------------|---
                                v
```

User-defined chains can jump to other user-defined chains (but don't make loops: your packets will be dropped if they're found to be in a loop).

Extensions to iptables: Targets

The other type of extension is a target. A target extension consists of a kernel module, and an optional extension to `iptables` to provide new command line options. There are several extensions in the default netfilter distribution:

Target Extensions

iptables can use extended target modules: the following are included in the standard distribution.

BALANCE

This allows you to DNAT connections in a round-robin way over a given range of destination addresses.

--to-destination *ipaddr-ipaddr*

Address range to round-robin over.

CLASSIFY

This module allows you to set the skb->priority value (and thus classify the packet into a specific CBQ class).

--set-class *MAJOR:MINOR*
Set the major and minor class value.

CLUSTERIP

This module allows you to configure a simple cluster of nodes that share a certain IP and MAC address without an explicit load balancer in front of them. Connections are statically distributed between the nodes in this cluster.

--new
Create a new ClusterIP. You always have to set this on the first rule for a given ClusterIP.

--hashmode *mode*
Specify the hashing mode. Has to be one of sourceip, sourceip-sourceport, sourceip-sourceport-destport

--clustermac *mac*
Specify the ClusterIP MAC address. Has to be a link-layer multicast address

--total-nodes *num*
Number of total nodes within this cluster.

--local-node *num*
Local node number within this cluster.

--hash-init *rnd*
Specify the random seed used for hash initialization.

CONNMARK

This module sets the netfilter mark value associated with a connection

--set-mark mark[/mask]
Set connection mark. If a mask is specified then only those bits set in the mask is modified.

--save-mark [--mask mask]
Copy the netfilter packet mark value to the connection mark. If a mask is specified then only those bits are copied.

--restore-mark [--mask mask]
Copy the connection mark value to the packet. If a mask is specified then only those bits are copied. This is only valid in the mangle table.

DNAT

This target is only valid in the nat table, in the PREROUTING and OUTPUT chains, and user-defined chains which are only called from those chains. It specifies that the destination address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one type of option:

--to-destination *ipaddr*[-*ipaddr*][:*port-port*]
Which can specify a single new destination IP address, an inclusive range of IP addresses, and optionally, a port range (which is only valid if the rule also specifies -p tcp or -p udp). If no port range is specified, then the destination port will never be modified.

You can add several --to-destination options. If you specify more than one destination address, either via an address range or multiple --to-destination options, a simple round-robin (one after another in cycle) load balancing takes place between these adresses.

DSCP

This target allows to alter the value of the DSCP bits within the TOS header of the IPv4 packet. As this manipulates a packet, it can only be used in the mangle table.

--set-dscp *value*
Set the DSCP field to a numerical value (can be decimal or hex)

--set-dscp-class *class*
Set the DSCP field to a DiffServ class.

ECN

This target allows to selectively work around known ECN blackholes. It can only be used in the mangle table.

--ecn-tcp-remove
Remove all ECN bits from the TCP header. Of course, it can only be used in conjunction with -p tcp.

LOG

Turn on kernel logging of matching packets. When this option is set for a rule, the Linux kernel will print some information on all matching packets (like most IP header fields) via the kernel log (where it can be read with *dmesg* or syslog). This is a "non-terminating target", i.e. rule traversal continues at the next rule. So if you want to LOG the packets you refuse, use two separate rules with the same matching criteria, first using target LOG then DROP (or REJECT).

--log-level *level*

Level of logging (numeric or see syslog.conf(5)).

--log-prefix *prefix*

Prefix log messages with the specified prefix; up to 29 letters long, and useful for distinguishing messages in the logs.

--log-tcp-sequence

Log TCP sequence numbers. This is a security risk if the log is readable by users.

--log-tcp-options

Log options from the TCP packet header.

--log-ip-options

Log options from the IP packet header.

MARK

This is used to set the netfilter mark value associated with the packet. It is only valid in the **mangle** table. It can for example be used in conjunction with iproute2.

--set-mark *mark*

MASQUERADE

This target is only valid in the nat table, in the POSTROUTING chain. It should only be used with dynamically assigned IP (dialup) connections: if you have a static IP address, you should use the SNAT target. Masquerading is equivalent to specifying a mapping to the IP address of the interface the packet is going out, but also has the effect that connections are *forgotten* when the interface goes down. This is the correct behavior when the next dialup is unlikely to have the same interface address (and hence any established connections are lost anyway). It takes one option:

--to-ports *port*[-*port*]
This specifies a range of source ports to use, overriding the default SNAT source port-selection heuristics (see above). This is only valid if the rule also specifies -p tcp or -p udp.

MIRROR

This is an experimental demonstration target which inverts the source and destination fields in the IP header and retransmits the packet. It is only valid in the INPUT, FORWARD and PREROUTING chains, and user-defined chains which are only called from those chains. Note that the outgoing packets are NOT seen by any packet filtering chains, connection tracking or NAT, to avoid loops and other problems.

NETMAP

This target allows you to statically map a whole network of addresses onto another network of addresses. It can only be used from rules in the nat table.

--to *address[/mask]*
Network address to map to. The resulting address will be constructed in the following way: All 'one' bits in the mask are filled in from the new `address'. All bits that are zero in the mask are filled in from the original address.

For example, if you want to alter the destination of incoming connections from 1.2.3.0/24 to 5.6.7.0/24, you can do as follows:

```
# iptables -t nat -A PREROUTING -d 1.2.3.0/24 -j NETMAP  \
            --to 5.6.7.0/24
# iptables -t nat --list
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
NETMAP     all  --  anywhere             1.2.3.0/24         5.6.7.0/24
```

NOTRACK

This target disables connection tracking for all packets matching that rule.

It can only be used in the raw table.

REDIRECT

This target is only valid in the nat table, in the PREROUTING and OUTPUT chains, and user-defined chains which are only called from those chains. It alters the destination IP address to send the packet to the machine itself (locally-generated packets are mapped to the 127.0.0.1 address). It takes one option:

--to-ports *port*[-*port*]

This specifies a destination port or range of ports to use: without this, the destination port is never altered. This is only valid if the rule also specifies -p tcp or -p udp.

REJECT

This is used to send back an error packet in response to the matched packet: otherwise it is equivalent to DROP so it is a terminating TARGET, ending rule traversal. This target is only valid in the INPUT, FORWARD and OUTPUT chains, and user-defined chains which are only called from those chains. The following option controls the nature of the error packet returned:

--reject-with *type*

The type given can be:
icmp-net-unreachable
icmp-host-unreachable
icmp-port-unreachable
icmp-proto-unreachable
icmp-net-prohibited
icmp-host-prohibited or
icmp-admin-prohibited (*)

Which return the appropriate ICMP error message (port-unreachable is the default). The option tcp-reset can be used on rules which only match the TCP protocol: this causes a TCP RST packet to be sent back. This is mainly useful for blocking *ident* (113/tcp) probes which frequently occur when sending mail to broken mail hosts (which won't accept your mail otherwise).

(*) Using icmp-admin-prohibited with kernels that do not support it will result in a plain DROP instead of REJECT

ROUTE

This is used to explicitly override the core network stack's routing decision. mangle table.

The ROUTE target lets you route a received packet through an interface or towards a host, even if the regular destination of the packet is the router itself. The ROUTE target is also able to change the incoming interface of a packet. Packets are directly put on the wire and do not traverse any other table.

This target does not modify the packets and is a final target. It has to be used inside the mangle table.

Whenever possible, you should use the MARK target together with iproute2 instead of this ROUTE target. However, this target is useful to force the use of an interface or a next hop and to change the incoming interface of a packet. People also use it for easiness and to simplify their rules (one rule to route a packet is easier that one MARK rule + one iproute2 rule).

--oif *ifname*

Route the packet through `ifname' network interface

--iif *ifname*

Change the packet's incoming interface to `ifname'

--gw *IP_address*

Route the packet via this gateway

--continue

Behave like a non-terminating target and continue traversing the rules. Not valid in combination with `--iif'

SAME

This match is similar to SNAT and will gives a client the same address for each connection.

--to <ipaddr>-<ipaddr>

Addresses to map source to. May be specified more than once for multiple ranges.

--nodst

Don't use destination-ip in source selection. For example, if you want to modify the source address of the connections to be 1.2.3.4-1.2.3.7 you can do as follows:

```
# iptables -t nat -A POSTROUTING -j SAME --to 1.2.3.4-1.2.3.7
# iptables -t nat –list
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SAME       all  --  anywhere             anywhere           same:1.2.3.4-1.2.3.7
```

SET

This modules adds and/or deletes entries from IP sets which can be defined by ipset(8).

--add-set setname flag[,flag...]
Add the address(es)/port(s) of the packet to the sets

--del-set setname flag[,flag...]
Delete the address(es)/port(s) of the packet from the sets, where flags are src and/or dst and there can be no more than six of them.

The bindings to follow must previously be defined in order to use multilevel adding/deleting by the SET target.

SNAT

This target is only valid in the nat table, in the POSTROUTING chain. It specifies that the source address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one type of option:

--to-source *ipaddr*[-*ipaddr*][:*port-port*]
This specifies a single new source IP address, an inclusive range of IP addresses, and optionally, a port range (which is only valid if the rule also specifies -p tcp or -p udp). If no port range is specified, then source ports below 512 will be mapped to other ports below 512: those between 512 and 1023 inclusive will be mapped to ports below 1024, and other ports will be mapped to 1024 or above. Where possible, no port alteration will occur.

You can add several --to-source options. If you specify more than one source address, either via an address range or multiple --to-source options, a simple round-robin (one after another in cycle) takes place between these addresses.

TCPMSS

This target allows to alter the MSS value of TCP SYN packets, to control the maximum size for that connection (usually limiting it to your outgoing interface's MTU minus 40). Of course, it can only be used in conjunction with -p tcp.

--set-mss *value*
Explicitly set MSS option to specified value.

--clamp-mss-to-pmtu
Automatically clamp MSS value to (path_MTU – 40).

These options are mutually exclusive.

This target is used to overcome criminally braindead ISPs or servers which block ICMP Fragmentation Needed packets. The symptoms of this problem are that everything works fine from your Linux firewall/router, but machines behind it can never exchange large packets:

1. Web browsers connect, then hang with no data received.
2. Small mail works fine, but large emails hang.
3. ssh works fine, but scp hangs after initial handshaking.

Workaround: activate this option and add a rule to your firewall configuration like:

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

TOS

This is used to set the 8-bit Type of Service field in the IP header. It is only valid in the **mangle** table.

--set-tos *tos*
You can use a numeric TOS values, or use
# iptables -j TOS -h
to see the list of valid TOS names.

TRACE

This target has no options. It just turns on packet tracing for all packets that match this rule.

TTL

This is used to modify the IPv4 TTL header field. The TTL field determines how many hops (routers) a packet can traverse until it's time to live is exceeded.

Setting or incrementing the TTL field can potentially be very dangerous, so it should be avoided at any cost.

Don't ever set or increment the value on packets that leave your local network!

--ttl-set *value*
    Set the TTL value to \`value'.

--ttl-dec *value*
    Decrement the TTL value \`value' times.

--ttl-inc *value*
    Increment the TTL value \`value' times.

For example, if you want to set the TTL of all outgoing connections to 126, you can do as follows:

```
# iptables -t mangle -A OUTPUT -j TTL --ttl-set 126
# iptables -t mangle --list
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
TTL        all  --  anywhere             anywhere           TTL set to 126
```

ULOG

This target provides userspace logging of matching packets. When this target is set for a rule, the Linux kernel will multicast this packet through a *netlink* socket. One or more userspace processes may then subscribe to various multicast groups and receive the packets. Like LOG, this is a "non-terminating target", i.e. rule traversal continues at the next rule.

--ulog-nlgroup *nlgroup*
  This specifies the netlink group (1-32) to which the packet is sent. Default value is 1.

--ulog-prefix *prefix*
  Prefix log messages with the specified prefix; up to 32 characters long, and useful for distinguishing messages in the logs.

--ulog-cprange *size*
  Number of bytes to be copied to userspace. A value of 0 always copies the entire packet, regardless of its size. Default is 0.

--ulog-qthreshold *size*
  Number of packet to queue inside kernel. Setting this value to, e.g. 10 accumulates ten packets inside the kernel and transmits them as one netlink multipart message to userspace. Default is 1 (for backwards compatibility).

Special Built-In Targets

There are two special built-in targets: RETURN and QUEUE.

RETURN
  has the same effect of falling off the end of a chain: for a rule in a built-in chain, the policy of the chain is executed. For a rule in a user-defined chain, the traversal continues at the previous chain, just after the rule which jumped to this chain.

QUEUE
  is a special target, which queues the packet for userspace processing. For this to be useful, two further components are required:

- a "queue handler", which deals with the actual mechanics of passing packets between the kernel and userspace; and
- a userspace application to receive, possibly manipulate, and issue verdicts on packets.

The standard queue handler for IPv4 iptables is the ip_queue module, which is distributed with the kernel and marked as experimental.

The following is a quick example of how to use iptables to queue packets for userspace processing:

```
# modprobe iptable_filter
# modprobe ip_queue
# iptables -A OUTPUT -p icmp -j QUEUE
```

With this rule, locally generated outgoing ICMP packets (as created with, say, ping) are passed to the ip_queue module, which then attempts to deliver the packets to a userspace application. If no userspace application is waiting, the packets are dropped.

The status of ip_queue may be checked via:

/proc/net/ip_queue

The maximum length of the queue (i.e. the number packets delivered to userspace with no verdict issued back) may be controlled via:

/proc/sys/net/ipv4/ip_queue_maxlen

The default value for the maximum queue length is 1024. Once this limit is reached, new packets will be dropped until the length of the queue falls below the limit again. Nice protocols such as TCP interpret dropped packets as congestion, and will hopefully back off when the queue fills up. However, it may take some experimenting to determine an ideal maximum queue length for a given situation if the default value is too small.

Operations on an Entire Chain

A very useful feature of `iptables` is the ability to group related rules into chains. You can call the chains whatever you want, but I recommend using lower-case letters to avoid confusion with the built-in chains and targets. Chain names can be up to 31 letters long.

Creating a New Chain

Let's create a new chain. Because I am such an imaginative fellow, I'll call it `test`. We use the `-N' or `--new-chain' options:

```
# iptables -N test
```

It's that simple. Now you can put rules in it as detailed above.

Deleting a Chain

Deleting a chain is simple as well, using the `-X' or `--delete-chain' options. Why `-X'? Well, all the good letters were taken.

```
# iptables -X test
```

There are a couple of restrictions to deleting chains: they must be empty and they must not be the target of any rule. You can't delete any of the three built-in chains.

If you don't specify a chain, then *all* user-defined chains will be deleted, if possible.

Flushing a Chain

There is a simple way of emptying all rules out of a chain, using the `-F' (or `--flush') commands.

```
# iptables -F FORWARD
```

If you don't specify a chain, then *all* chains will be flushed.

Listing a Chain

You can list all the rules in a chain by using the `-L' (or `--list') command.

The `refcnt' listed for each user-defined chain is the number of rules which have that chain as their target. This must be zero (and the chain be empty) before this chain can be deleted.

If the chain name is omitted, all chains are listed, even empty ones.

There are three options which can accompany `-L'. The `-n' (numeric) option is very useful as it prevents iptables from trying to lookup the IP addresses, which (if you are using DNS like most people) will cause large delays if your DNS is not set up properly, or you have filtered out DNS requests. It also causes TCP and UDP ports to be printed out as numbers rather than names.

The `-v' options shows you all the details of the rules, such as the the packet and byte counters, the TOS comparisons, and the interfaces.

Otherwise these values are omitted.

Note that the packet and byte counters are printed out using the suffixes `K', `M' or `G' for 1000, 1,000,000 and 1,000,000,000 respectively. Using the `-x' (expand numbers) flag as well prints the full numbers, no matter how large they are.

Resetting (Zeroing) Counters

It is useful to be able to reset the counters. This can be done with the `-Z' (or `--zero') option.

Consider the following:

```
# iptables -L FORWARD
# iptables -Z FORWARD
```

In the above example, some packets could pass through between the `-L' and `-Z' commands. For this reason, you can use the `-L' and `-Z' *together*, to reset the counters while reading them.

Setting Policy

We glossed over what happens when a packet hits the end of a built-in chain when we discussed how a packet walks through chains earlier. In this case, the policy of the chain determines the fate of the packet. Only built-in chains (INPUT, OUTPUT and FORWARD) have policies, because if a packet falls off the end of a user-defined chain, traversal resumes at the previous chain.

The policy can be either ACCEPT or DROP, for example:

```
# iptables -P FORWARD DROP
```

Layer-7 Traffic Classification

L7-filter is a new packet classifier for the Linux kernel. Unlike other classifiers, it doesn't just look at simple values such as port numbers. Instead, it does regular expression matching on the application layer data to determine what protocols are being used.

Since this classifier is much more processor and memory intensive than others, we recommend that you only use it if you have reason to believe that matching by port (or IP number, etc.) is insufficient for your purposes.

Blocking... Don't! Why?

- l7-filter matching isn't foolproof: there may be both false positives (one protocol looking like another) and false negatives (applications can do obscure things that we didn't count on).
- With the exception of worms and viruses, almost every type of Internet traffic has legitimate uses. For instance, P2P protocols, while widely used to violate copyright, are also an efficient way to distribute open source software and legally free music.
- Insidious programs can respond to being blocked by port-hopping, switching between TCP and UDP, opening a new connection for every trivial operation, or using other evasion tactics. This can make them very hard to identify. Don't encourage program authors to include these "features"!
- When you block a program, you are providing the program authors with a strong incentive to make their protocol impossible to identify by encrypting it end-to-end.
- Blocking with l7-filter provides no security, since any reasonably determined person can easily circumvent it.

Instead of dropping packets you don't like, we recommend using Linux to restrict their bandwidth usage. See the next section.

If you insist on using l7-filter to drop packets, make sure you have investigated other options first, such as the features of your HTTP proxy (useful for worms). Still reading this section? Fine, then. Blocking is easy. Simply use "-j DROP" or "-j REJECT".

Bandwidth Restriction Using Traditional Methods

To control the bandwidth that a protocol uses, you can use Netfilter to "mark" the packets and QoS to filter on that mark. To mark:

```
# iptables -t mangle -A POSTROUTING -m layer7 --l7proto imap  \
        -j MARK --set-mark 3
```

The number "3" is arbitrary. It can be any integer. Then use `tc` to filter on that mark:

```
# tc filter add dev eth0 protocol ip parent 1:0 prio 1 handle 3  \
        fw flowid 1:3
```

Dealing with FTP, IRC, etc.

Some protocols open child connections to transfer data. FTP is the most familiar example. If you have loaded the ip_conntrack_ftp kernel module, l7-filter will classify FTP and all its child connections as FTP. The same goes for IRC/IRC-DCC, etc.

If you wish to classify the children differently, use the standard iptables "helper" match. You can use "-m --helper ftp" to match ftp child connections. This is a little silly, of course, because if this works, you don't need l7-filter, at least for the children.

The "unknown" match

l7-filter marks connections that it has given up trying to match as "unknown". In contrast, connections which are unidentified, but still being examined, have no classification. You can match on "unknown" as though it were a normal protocol.
This is useful, because you may want to do something to packets from unidentified connections. But since l7-filter usually must examine several packets of a connection before a match is made, some care is needed. You don't want to say "if not HTTP and not DNS, do X", because that will do X to the TCP handshakes of HTTP, which is probably not what you want. Rather, you want to say "check for HTTP and DNS. If 'unknown', do X". More concretely:

```
# iptables -t mangle -A POSTROUTING -m layer7 --l7proto http
# iptables -t mangle -A POSTROUTING -m layer7 --l7proto dns
# iptables -t mangle -A POSTROUTING -m layer7 --l7proto  \
                unknown -j [...]
```

Something else to know

If you want to update the protocols, you will need to clear the relevant iptables rules and re-enter them. This is because the pattern files are only read by iptables, not directly the kernel.

Bridging

What does a bridge do?
A bridge transparently relays traffic between multiple network interfaces. In plain English this means that a bridge connects two or more physical Ethernets together to form one bigger (logical) Ethernet.

Is it protocol independent?
Yes. The bridge knows nothing about protocols, it only sees Ethernet frames. As such, the bridging functionality is protocol independent, and there should be no trouble relaying IPX, NetBEUI, IP, IPv6, etc.

Why is this code better than a switch?
Please note that this code wasn't written with the intent of having Linux boxen take over from dedicated networking hardware. Don't see the Linux bridging code as a replacement for switches, but rather as an extension of the Linux networking capabilities. Just as there are situations where a Linux router is better than a dedicated router (and vice versa), there are situations where a Linux bridge is better than a dedicated bridge (and vice versa).

Most of the power of the Linux bridging code lies in its flexibility. There is a whole lot of bizarre stuff you can do with Linux already (read Linux Advanced Routing and Traffic Control document to see some of the possiblities), and the bridging code adds some more filter into the mix.

One of the most significant advantages of a Linux solution over a dedicated solution that come to mind is Linux' extensive firewalling capabilities. It is possible to use the full functionality of netfilter (iptables) in combination with bridging, which provides way more functionality than most proprietary offerings do.

Why is this code worse than a switch?
In order to act a a bridge, the network device must be placed into promiscuous mode which means it receives all traffic on a network. On a really busy network, this can eat significant bandwidth out of the processor, memory slowing the system down. The answer is to setup either a separate dedicated Linux box as the bridge, or use a hardware switch.

What is the performance of the bridge?
The performance is limited by the network cards used and the processor. A

research paper was done by James Yu at Depaul University comparing Linux bridging with a Catalyst switch: http://facweb.cti.depaul.edu/jyu/Publications/Yu-Linux-TSM2004.pdf

What can be bridged?
Linux bridging is very flexible; the LAN's can be either traditional Ethernet device's, or pseudo-devices such as PPP, VPN's or VLAN's. The only restrictions are that the devices:

- All devices same maximum packet size (MTU). The bridge doesn't fragment packets.
- Devices must look like Ethernet. i.e have 6 byte source and destination address.
- Support promiscuous operation. The bridge needs to be able to receive all network traffic, not just traffic destined for its own address.
- Allow source address spoofing. The bridge must be able to send data over network as if it came from another host.

What is bridge-nf?
It is the infrastructure that enables {ip,ip6,arp}tables to see bridged IPv4, resp. IPv6, resp. ARP packets. Thanks to bridge-nf, you can use these tools to filter bridged packets, letting you make a transparent firewall. Note that bridge-nf is also referred to as bridge-netfilter and br-nf, the term bridge-nf should be preferred.

Why do I need bridge-nf?
Ebtables only allows basic filtering of the IPv4 and ARP packets, for more advanced filtering you need to use the {ip,ip6,arp}tables applications. Iptables in combination with bridge-nf also allows you to do things like transparent IP NAT.

Connection tracking
What happens when I enable connection tracking (for IPv4 traffic)?

By default, when IPv4 connection tracking is loaded in the kernel (if your kernel is modular, it is the nf_conntrack module), all IP packets will be seen by the connection tracking code. This code is called on the PF_INET/PRE_ROUTING and PF_INET/LOCAL_OUT hooks. For bridged packets, only the PRE_ROUTING connection tracking is important.

What are the disadvantages of connection tracking on a bridging firewall?

1. For an IP packet entering a bridge device, connection tracking is called before the bridge code decides what to do with the packet. This means that IP packets that will be discarded by the bridge code are tracked by connection tracking. For a router, the same is true, but a bridge also sees the traffic between hosts on the same side of a network. It's possible to prevent these packets from being seen by connection tracking: you can either drop them in the ebtables nat PREROUTING chain or use the iptables NOTRACK target.

2. Fragmented IP packets (typically UDP traffic like NFS) are defragmented by the connection tracking code and refragmented before sending them out. This slows down traffic, but the transparency of the firewall isn't diminished.

What happens with IP DNAT on a to be bridged packet?
If IP DNAT happened then the bridge-nf code asks the routing table where the packet should be sent. If it has to be sent over another device (not the bridge device) then the packet is routed (an implicit redirect). If the routing table sends the packet to the bridge device, then the packet is bridged but the MAC destination is correctly changed. To do IP DNAT, you therefore need a correct routing table.

ebtables

What is ebtables?
The ebtables utility enables basic Ethernet frame filtering on a Linux bridge, logging, MAC NAT and brouting. It only provides basic IP filtering, the full-fledged IP filtering on a Linux bridge is done with iptables. The so-called bridge-nf code makes iptables see the bridged IP packets and enables transparent IP NAT. The firewalling tools iptables and ebtables can be used together and are complementary. ebtables tries to provide the bridge firewalling that iptables cannot provide, namely the filtering of non-IP traffic.

Main features:

- Usage analogous to iptables.
- Ethernet filtering.
- MAC NAT: ability to alter the MAC Ethernet source and destination address. This can be useful in some very strange setups (a real-life example is available).
- Brouting: decide which traffic to bridge between two interfaces and which traffic to route between the same two interfaces. The two interfaces belong to a logical bridge device but have their own IP address and can belong to a different subnet.
- Pass packets to userspace programs, using netlink sockets (the ulog watcher).

What can ebtables do?

- Ethernet protocol filtering.
- MAC address filtering.
- Simple IP header filtering.
- ARP header filtering.
- 802.1Q VLAN filtering.
- In/Out interface filtering (logical and physical device).
- MAC address nat.
- Logging.
- Frame counters.
- Ability to add, delete and insert rules; flush chains; zero counters.
- Brouter facility.
- Ability to atomically load a complete table, containing the rules you made, into the kernel. See the man page and the examples section.
- Support for user defined chains.
- Support for marking frames and matching marked frames.

CHAINS

There are three Ethernet frame tables with built-in chains in the Linux kernel. The kernel tables are used to divide functionality into different sets of rules. Each set of rules is called a chain. Each chain is an ordered list of rules that can match Ethernet frames. If a rule matches an Ethernet frame, then a processing specification tells what to do with that matching frame. The processing specification is called a 'target'. However, if the frame does not match the current rule in the chain, then the next rule in the chain is examined and so forth. The user can create new (user-defined) chains which can be used as the 'target' of a rule.

TARGETS

A firewall rule specifies criteria for an Ethernet frame and a frame processing specification called a target. When a frame matches a rule, then the next action performed by the kernel is specified by the target. The target can be one of these values: ACCEPT, DROP, CONTINUE, RETURN, an 'extension' (see below) or a user-defined chain.

ACCEPT means to let the frame through. DROP means the frame has to be dropped. CONTINUE means the next rule has to be checked. This can be handy to know how many frames pass a certain point in the chain or to log those frames. RETURN means stop traversing this chain and resume at the next rule in the previous (calling) chain. For the extension targets please see the TARGET EXTENSIONS section of this man page.

TABLES

As stated earlier, there are three Ethernet frame tables in the Linux kernel. The tables are filter, nat and broute. Of these three tables, the filter table is the default table that the ebtables command operates on. If you are working with the filter table, then you can drop the '-t filter' argument to the ebtables command. However, you will need to provide the -t argument for the other two tables. The -t argument must be the first argument on the ebtables command line, if used.

-t, --table tablename

filter

The default table and contains three built-in chains: INPUT (for frames destined for the bridge itself), OUTPUT (for locally-generated frames) and FORWARD (for frames being bridged).

nat

Used to change the mac addresses and contains three built-in chains: PREROUTING (for altering frames as soon as they come in), OUTPUT (for altering locally generated frames before they are bridged) and POSTROUTING (for altering frames as they are about to go out). A small note on the naming of chains POSTROUTING and PREROUTING: it would be more accurate to call them PREFORWARDING and POSTFORWARDING, but for all those who come from the iptables world to ebtables it is easier to have the same names.

broute

Used to make a brouter, it has one built-in chain: BROUTING. The targets DROP and ACCEPT have special meaning in the broute table. DROP actually means the frame has to be routed, while ACCEPT means the frame has to be bridged. The BROUTING chain is traversed very early. It is only traversed by frames entering on a bridge enslaved NIC that is in forwarding state. Normally those frames would be bridged, but you can decide otherwise here. The redirect target is very handy here.

COMMAND LINE ARGUMENTS

After the initial ebtables -t, table command line argument, the remaining arguments can be divided into several different groups. These groups are commands, miscellaneous commands, rule-specifications, match extensions, and watcher extensions.

COMMANDS

The ebtables command arguments specify the actions to perform on the table defined with the -t argument. If you do not use the -t argument to name a table, the commands apply to the default filter table. With the exception of both the -Z and --atomic-file commands, only one command may be used on the command line at a time.

-A, --append chain

Append a rule to the end of the selected chain.

-D, --delete start[:end] or rule_spec

Delete the specified rule from the selected chain. There are two ways to use this command. The first is by specifying an interval of rule numbers to delete. Using negative numbers is allowed, for more details about using negative numbers, see the -I command. The second usage is by specifying the complete rule as it would have been specified when it was added.

-I, --insert number rule_spec

Insert the specified rule into the selected chain at the specified rule number. If the current number of rules equals N, then the specified number can be between -N and N+1. For a positive number i, it holds that i and i-N-1 specify the same place in the chain where the rule should be inserted. The number 0 specifies the place past the last rule in the chain and using this number is therefore equivalent with using the -A command.

-P, --policy policy

Set the policy for the chain to the given target. The policy can be ACCEPT, DROP or RETURN.

-F, --flush chain

Flush the selected chain. If no chain is selected, then every chain will be flushed. Flushing the chain does not change the policy of the chain, however.

-Z, --zero chain

Set the counters of the selected chain to zero. If no chain is selected, all the counters are set to zero. The -Z command can be used in conjunction with the -L command. When both the -Z and -L commands are used together in this way, the rule counters are printed on the screen before they are set to zero.

-L, --list [chain]

List all rules in the selected chain. If no chain is selected, all chains are listed.

The following three options change the output of the -L list command:

--Ln

Places the rule number in front of every rule.

--Lc

Shows the counters at the end of each rule displayed by the -L command. Both a frame counter (pcnt) and a byte counter (bcnt) are displayed.

--Lx

The output of the --Lx option may be used to create a set of ebtables commands. You may use this set of commands in an ebtables boot or reload script. For example the output could be used at system startup. The --Lx option is incompatible with both of the other --Ln and --Lc chain listing options.

--Lmac2
Shows all MAC addresses with the same length, adding leading zeroes if necessary. The default representation omits zeroes in the addresses when they are not needed.

All necessary ebtables commands for making the current list of user-defined chains in the kernel and any commands issued by the user to rename the standard ebtables chains will be listed, when no chain name is supplied for the -L command while using the --Lx option.

-N, --new-chain chain
Create a new user-defined chain with the given name. The number of user-defined chains is unlimited. A user-defined chain name has maximum length of 31 characters.

-X, --delete-chain chain
Delete the specified user-defined chain. There must be no remaining references to the specified chain, otherwise ebtables will refuse to delete it. If no chain is specified, all user-defined chains that aren't referenced will be removed.

-E, --rename-chain old_chain nmew chain
Rename the specified chain to a new name. Besides renaming a user-defined chain, you may rename a standard chain name to a name that suits your taste. For example, if you like PREBRIDGING more than PREROUTING, then you can use the -E command to rename the PREROUTING chain. If you do rename one of the standard ebtables chain names, please be sure to mention this fact should you post a question on the ebtables mailing lists. It would be wise to use the standard name in your post. Renaming a standard ebtables chain in this fashion has no effect on the structure or function of the ebtables kernel table.

--init-table
Replace the current table data by the initial table data.

--atomic-init file
Copy the kernel's initial data of the table to the specified file. This can be used as the first action, after which rules are added to the file. The file can be specified using the --atomic-file command or through the EBTABLES_ATOMIC_FILE environment variable.

--atomic-save file
Copy the kernel's current data of the table to the specified file. This can be used as the first action, after which rules are added to the file. The file can be specified using the --atomic-file command or through the EBTABLES_ATOMIC_FILE environment variable.

--atomic-commit file
Replace the kernel table data with the data contained in the specified file. This is a useful command that allows you to load all your rules of a certain table into the kernel at once, saving the kernel a lot of precious time and allowing atomic updates of the tables. The file which contains the table data is constructed by using either the --atomic-init or the --atomic-save command to generate a starting file. After that, using the --atomic-file command when constructing rules or setting the EBTABLES_ATOMIC_FILE environment variable allows you to extend the file and build the complete table before committing it to the kernel.

--atomic-file File **-Z**
The counters stored in a file with, say, --atomic-init can be optionally zeroed by supplying the -Z command. You may also zero the counters by setting the EBTABLES_ATOMIC_FILE environment variable.

RULE-SPECIFICATIONS

The following command line arguments make up a rule specification (as used in the add and delete commands). A "!" option before the specification inverts the test for that specification. Apart from these standard rule specifications there are some other command line arguments of interest. See both the MATCH-EXTENSIONS and the WATCHER-EXTENSION(S) below.

-p, --protocol [!] protocol
The protocol that was responsible for creating the frame. This can be a hexadecimal number, above 0x0600, a name (e.g. ARP) or LENGTH. The protocol field of the Ethernet frame can be used to denote the length of the header (802.2/802.3 networks). When the value of that field is below (or equals) 0x0600, the value equals the size of the header and shouldn't be used as a protocol number. Instead, all frames where the protocol field is used as the length field are assumed to be of the same 'protocol'. The protocol name used in ebtables for these frames is LENGTH. The file /etc/ethertypes can be used to show readable characters instead of

hexadecimal numbers for the protocols. For example, 0x0800 will be represented by IPV4. The use of this file is not case sensitive. See that file for more information. The flag --proto is an alias for this option.

-i, --in-interface [!] name
The interface via which a frame is received (for the INPUT, FORWARD, PREROUTING and BROUTING chains). The flag --in-if is an alias for this option.

--logical-in [!] name
The (logical) bridge interface via which a frame is received (for the INPUT, FORWARD, PREROUTING and BROUTING chains).

-o, --out-interface [!] name
The interface via which a frame is going to be sent (for the OUTPUT, FORWARD and POSTROUTING chains). The flag --out-if is an alias for this option.

--logical-out [!] name
The (logical) bridge interface via which a frame is going to be sent (for the OUTPUT, FORWARD and POSTROUTING chains).

-s, --source [!] address[/mask]
The source mac address. Both mask and address are written as 6 hexadecimal numbers separated by colons. Alternatively one can specify Unicast, Multicast, Broadcast or BGA (Bridge Group Address).

Unicast = 00:00:00:00:00:00/01:00:00:00:00:00

Multicast = 01:00:00:00:00:00/01:00:00:00:00:00

Broadcast = ff:ff:ff:ff:ff:ff/ff:ff:ff:ff:ff:ff or

BGA = 01:80:c2:00:00:00/ff:ff:ff:ff:ff:ff.

Note that a broadcast address will also match the multicast specification. The flag --src is an alias for this option.

-d, --destination [!] address[/mask]
The destination mac address. See -s (above) for more details. The flag --dst is an alias for this option.

MATCH-EXTENSIONS

ebtables extensions are precompiled into the userspace tool. So there is no need to explicitly load them with a -m option like in iptables. However, these extensions deal with functionality supported by supplemental kernel modules.

802.3

Specify 802.3 DSAP/SSAP fields or SNAP type. The protocol must be specified as LENGTH (see protocol above).

--802_3-sap [!] sap
DSAP and SSAP are two one byte 802.3 fields. The bytes are always equal, so only one byte (hexadecimal) is needed as an argument.

--802_3-type [!] type
If the 802.3 DSAP and SSAP values are 0xaa then the SNAP type field must be consulted to determine the payload protocol. This is a two byte (hexadecimal) argument. Only 802.3 frames with DSAP/SSAP 0xaa are checked for type.

arp

Specify arp fields. The protocol must be specified as ARP or RARP.

--arp-opcode [!] opcode
The (r)arp opcode (decimal or a string, for more details see ebtables -h arp).

--arp-htype [!] hardware type
The hardware type, this can be a decimal or the string "Ethernet". This is normally Ethernet (value 1).

--arp-ptype [!] protocol type
The protocol type for which the (r)arp is used (hexadecimal or the string "IPv4"). This is normally IPv4 (0x0800).

--arp-ip-src [!] address[/mask]
The ARP IP source address specification.

--arp-ip-dst [!] address[/mask]
The ARP IP destination address specification.

--arp-mac-src [!] address[/mask]
The ARP MAC source address specification.

--arp-mac-dst [!] address[/mask]
The ARP MAC destination address specification.

ip

Specify ip fields. The protocol must be specified as IPv4.

--ip-source [!] address[/mask]
The source ip address. The flag --ip-src is an alias for this option.

--ip-destination [!] address[/mask]
The destination ip address. The flag --ip-dst is an alias for this option.

--ip-tos [!] tos
The ip type of service, in hexadecimal numbers. IPv4.

--ip-protocol [!] protocol
The ip protocol. The flag --ip-proto is an alias for this option.

--ip-source-port [!] port[:port]
The source port or port range for the ip protocols 6 (TCP) and 17 (UDP). If the first port is omitted, "0" is assumed; if the last is omitted, “65535” is assumed. The flag --ip-sport is an alias for this option.

--ip-destination-port [!] port[:port]
The destination port or port range for ip protocols 6 (TCP) and 17 (UDP). The flag --ip-dport is an alias for this option.

mark_m

--mark [!] [value][/mask]
Matches frames with the given unsigned mark value. If a mark value and mask is specified, the logical AND of the mark value of the frame and the user-specified mask is taken before comparing it with the user-specified mark value. If only a mask is specified (start with '/') the logical AND of the mark value of the frame and the user-specified mark is taken and the result is compared with zero.

pkttype

--pkttype-type [!] type
Matches on the Ethernet "class" of the frame, which is determined

by the generic networking code. Possible values: broadcast (MAC destination is broadcast address), multicast (MAC destination is multicast address), host (MAC destination is the receiving network device) or otherhost (none of the above).

stp

Specify stp BPDU (bridge protocol data unit) fields. The destination address must be specified as the bridge group address (BGA).

--stp-type [!] type

The BPDU type (0-255), special recognized types: config: configuration BPDU (=0) and tcn: topology change notification BPDU (=128).

--stp-flags [!] flag

The BPDU flag (0-255), special recognized flags: topology-change: the topology change flag (=1) topology-change-ack: the topology change acknowledgement flag (=128).

--stp-root-prio [!] [prio][:prio]

The root priority (0-65535) range.

--stp-root-addr [!] [address][/mask]

The root mac address, see the option -s for more details.

--stp-root-cost [!] [cost][:cost]

The root path cost (0-4294967295) range.

--stp-sender-prio [!] [prio][:prio]

The BPDU's sender priority (0-65535) range.

--stp-sender-addr [!] [address][/mask]

The BPDU's sender mac address, see the option -s for more details.

--stp-port [!] [port][:port]

The port identifier (0-65535) range.

--stp-msg-age [!] [age][:age]

The message age timer (0-65535) range.

--stp-max-age [!] [age][:age]

The max age timer (0-65535) range.

--stp-hello-time [!] [time][:time]
The hello time timer (0-65535) range.

--stp-forward-delay [!] [delay][:delay]
The forward delay timer (0-65535) range.

vlan

Specify 802.1Q Tag Control Information fields. The protocol must be specified as 802_1Q (0x8100).

--vlan-id [!] id
The VLAN identifier field (VID). Decimal number from 0 to 4095.

--vlan-prio [!] prio
The user_priority field. Decimal number from 0 to 7. The VID should be set to 0 ("null VID") or unspecified (for this case the VID is deliberately set to 0).

--vlan-encap [!] type
The encapsulated Ethernet frame type/length. Specified as hexadecimal number from 0x0000 to 0xFFFF or as a symbolic name from /etc/ethertypes.

WATCHER-EXTENSION(S)

Watchers are things that only look at frames passing by. These watchers only see the frame if the frame matches the rule.

log

The fact that the log module is a watcher lets us log stuff while giving a target by choice. Note that the log module therefore is not a target.

--log
Use this if you won't specify any other log options, so if you want to use the default settings: log-prefix="", no arp logging, no ip logging, log-level=info.

--log-level level
defines the logging level. For the possible values: ebtables -h log. The default level is info.

--log-prefix text

defines the prefix to be printed before the logging information.

--log-ip

will log the ip information when a frame made by the ip protocol matches the rule. The default is no ip information logging.

--log-arp

will log the (r)arp information when a frame made by the (r)arp protocols matches the rule. The default is no (r)arp information logging.

TARGET EXTENSIONS

arpreply

The arpreply target can be used in the PREROUTING chain of the nat table. If this target sees an arp request it will automatically reply with an arp reply. The used MAC address for the reply can be specified. When the arp message is not an arp request, it is ignored by this target.

--arpreply-mac address

Specifies the MAC address to reply with: the Ethernet source MAC and the ARP payload source MAC will be filled in with this address.

--arpreply-target target

Specifies the standard target. After sending the arp reply, the rule still has to give a standard target so ebtables knows what to do. The default target is DROP.

dnat

The dnat target can only be used in the BROUTING chain of the broute table and the PREROUTING and OUTPUT chains of the nat table. It specifies that the
destination mac address has to be changed.

--to-destination address

The flag --to-dst is an alias for this option.

--dnat-target target

Specifies the standard target. After doing the dnat, the rule still has to give a standard target so ebtables knows what to do. The default target is ACCEPT. Making it CONTINUE could let you use multiple target extensions on the same frame. Making it DROP only makes sense in the BROUTING chain but using the redirect target is more logical there. RETURN is also allowed. Note that using RETURN in a base chain is not allowed.

mark

The mark target can be used in every chain of every table. It is possible to use the marking of a frame/packet in both ebtables and iptables, if the br-nf code is compiled into the kernel. Both put the marking at the same place. So, you can consider this fact as a feature, or as something to watch out for.

--set-mark value

Mark the frame with the specified unsigned value.

--mark-target target

Specifies the standard target. After marking the frame, the rule still has to give a standard target so ebtables knows what to do. The default target is ACCEPT. Making it CONTINUE can let you do other things with the frame in other rules of the chain.

redirect

The redirect target will change the MAC target address to that of the bridge device the frame arrived on. This target can only be used in the BROUTING chain of the broute table and the PREROUTING chain of the nat table.

--redirect-target target

Specifies the standard target. After doing the MAC redirect, the rule still has to give a standard target so ebtables knows what to do. The default target is ACCEPT. Making it CONTINUE could let you use multiple target extensions on the same frame. Making it DROP in the BROUTING chain will let the frames be routed. RETURN is also allowed. Note that using RETURN in a base chain is not allowed.

snat

The snat target can only be used in the POSTROUTING chain of the nat table. It specifies that the source mac address has to be changed.

--to-source address

The flag --to-src is an alias for this option.

--snat-target target

Specifies the standard target. After doing the snat, the rule still has to give a standard target so ebtables knows what to do. The default target is ACCEPT. Making it CONTINUE could let you use multiple target extensions on the same frame. Making it DROP doesn't make sense, but you could do that too. RETURN is also allowed. Note that using RETURN in a base chain is not allowed.

Bridging and Firewalling

A Linux bridge is more powerful than a pure hardware bridge because it can also filter and shape traffic.

Network cards

Before you start make sure both network cards are set up and working properly. Don't set the IP address, and don't let the startup scripts run DHCP on the ethernet interfaces either. The IP address needs to be set after the bridge has been configured.

The command ifconfig should show both network cards, and they should have be DOWN.

Module loading

In most cases, the bridge code is built as a module. If the module is configured and installed correctly, it will get automatically loaded on the first brctl command.

If your bridge-utilities have been correctly built and your kernel and bridge-module are OK, then issuing a brctl should show a small command synopsis.

brctl Commands:

addbr <bridge>

This command is used to create a logical bridge instance with the name <*bridge*>. You will need at least one logical instance to do any bridging at all. You can interpret the logical bridge as a container for the interfaces taking part in the bridging. Each bridging instance is represented by a new network interface.

Example:

```
# brctl addbr mybridge
```

delbr <bridge>

Delete instance <bridge> from the ethernet bridge. The network interface corresponding to the bridge must be down before it can be deleted!

Example:

```
# brctl delbr mybridge
```

addif <bridge> <device>

Adds the network device *device* to take part in the bridging of <bridge>. All the devices contained in a bridge act as one big network. It is not possible to add a device to multiple bridges or bridge a bridge device, because it just wouldn't make any sense! The bridge will take a short amount of time when a device is added to learn the Ethernet addresses on the segment before starting to forward.

Example:

```
# brctl addif mybridge eth0
```

delif <bridge> <device>

Detatch interface <device> from bridge.

Example:

```
# brctl delif mybridge eth0
```

setageing <bridge> <time>

Sets the ethernet (MAC) address ageing time, in seconds. After <time> seconds of not having seen a frame coming from a certain address, the bridge will time out (delete) that address from the Forwarding DataBase (fdb).

Example:

```
# brctl setaging mybridge 5
```

setbridgeprio <bridge> <prio>

Each bridge has a relative priority and cost. Each interface is associated with a port (number) in the STP code. Each has a priority and a cost, that is used to decide which is the shortest path to forward a packet. The lowest cost path is always used unless the other path is down. If you have multiple bridges and interfaces then you may need to adjust the priorities to achieve optimum performance. The priority value is an unsigned 16-bit quantity (a number between 0 and 65535), and has no dimension. Lower priority values are 'better'. The bridge with the lowest priority will be elected 'root bridge'. The root bridge is the "central" bridge in the spanning tree.

Example:

# brctl setbridgeprio mybridge *10*

setfd <bridge> <time>

Forwarding delay time is the time spent in each of the Listening and Learning states before the Forwarding state is entered. This delay is so that when a new bridge comes onto a busy network it looks at some traffic before participating. Time is in seconds.

Example:

# brctl setfd *mybridge 10*

sethello <bridge> <time>

Periodically, a hello packet is sent out by the Root Bridge and the Designated Bridges. Hello packets are used to communicate information about the topology throughout the entire Bridged Local Area Network. Time is in seconds.

Example:

# brctl sethello *mybridge 10*

setmaxage <bridge> <time>

If a another bridge in the spanning tree does not send out a hello packet for a long period of time, it is assumed to be dead. The time value is in seconds.

Example:

# brctl maxage *mybridge 10*

setpathcost <bridge> <port> <cost>

Each interface in a bridge could have a different speed and this value is used when deciding which link to use. Faster interfaces should have lower costs. For multiple ports with the same cost there is also a priority

Example:

# brctl *setpathcost mybridge eth1 5*

setportprio <bridge> <port> <prio>

Each bridge port has a relative priority. The priority value is an unsigned 16-bit quantity (a number between 0 and 65535), and has no dimension. Lower priority values are 'better'. This metric is used in the designated port and root port selection algorithms.

Example:

# brctl setportprio mybridge eth0 3

show

Show a list of bridges.

Example:

# brctl show

showmacs <bridge>

Show a list of mac addresses relating to <bridge>.

Example:

# brctl showmacs mybridge

showstp <bridge>

Show <bridge>'s stp info.

Example:

# brctl showstp mybridge

stp <bridge> <state>

If you are running multiple or redundant bridges, then you need to enable the Spanning Tree Protocol to handle multiple hops and avoid cyclic routes.

For example:

```
# brctl stp on
```

You can see the STP paramaters with:

```
 # brctl showstp br549
```

Example: Showing devices in a bridge

The *brctl show* command gives you a summary about the overall bridge status, and the instances running as shown below:

```
# brctl addbr br549
# brctl addif br549 eth0
# brctl addif br549 eth1
# brctl show
bridge name     bridge id               STP enabled     interfaces
br549           8000.00004c9f0bd2       no              eth0  eth1
```

Once a bridge is running the *brctl showmacs* will show information about network addresses of traffic being forwarded (and the bridge itself).

```
# brctl showmacs br549
port no mac addr                is local?       ageing timer
  1     00:00:4c:9f:0b:ae       no               17.84
  1     00:00:4c:9f:0b:d2       yes               0.00
  2     00:00:4c:9f:0b:d3       yes               0.00
  1     00:02:55:1a:35:09       no               53.84
  1     00:02:55:1a:82:87       no               11.53
...
```

The aging time is the number of seconds a MAC address will be kept in the forwarding database after having received a packet from this MAC address. The entries in the forwarding database are periodically timed out to ensure they won't stay around forever. Normally there should be no need to modify this parameter, but it can be changed with (time is in seconds).

```
# brctl setageing 0
```

Setting ageing time to zero makes all entries permanent.

Sample setup

The basic setup of a bridge is done like:

```
# ifconfig eth0 0.0.0.0
# ifconfig eth1 0.0.0.0
# brctl addbr mybridge
# brctl addif mybridge eth0
# brctl addif mybridge eth1
# ifconfig mybridge up
```

This will set the host up as a pure bridge, it will not have an IP address for itself, so it can not be remotely accessed (or hacked) via TCP/IP.

Optionally you can configure the virtual interface mybridge to take part in your network. It behaves like one interface (like a normal network card). Exactly that way you configure it, replacing the previous command with something like:

```
# ifconfig mybridge 192.168.100.5 netmask 255.255.255.0
```

Using BWM Tools

Installing BWM Tools

Before you can use BWM Tools, you must make sure you have all the dependencies installed...

- glib2 >= 2.2.0
- libxml2 >= 2.5.0
- rrdtool >= 1.0.49 (required for graphing)

Next you need to download BWM Tools, compile it and install it.

Here is step-by-step instructions on how to do this...

1. Download the latest version of BWM Tools, the latest version can be found on the project homepage: http://bwm-tools.pr.linuxrulz.org

2. Uncompress the archive using either

   # tar jxvf <archive name>.tar.bz2

   or

   # tar zxvf <archive name>.tar.gz

   depending weather its a .tar.bz2 or .tar.gz respectively.

3. Run

   # ./configure

   in the source directory. Optionally a --prefix=... parameter can be passed which will determine where BWM Tools will be installed.

4. Once the configure process is complete, issue a make command, this will compile BWM Tools.

5. When BWM Tools has finished compiling, type

   # make install

   This will by default install BWM Tools into /usr/local, unless of course if you specified a --prefix=... above.

BWM Tools Utilities

bwm_firewall

This utility is used to create an IPTables compatible dump file. Parameters and usage are described below...

-c or **--config=**<config_file>
Specify the configuration file to use, defaults to <prefix>/etc/bwm_tools/firewall.xml

-f or **--file**[=<output_file>]
Generate IPTables restore file from the BWM Tools configuration file. If this option isn't given an optional filename it will default to <prefix>/etc/bwm_tools/firewall.xml

-l or **--load**
Load configuration directly into kernel, if specified with the **-f** option above, it will write the IP Tables restore file and load it. If this option is specified alone, no IP Tables restore file will be created, the configuration will be loaded directly into the kernel.

-r or **--reset-counters**
Reset iptables counters, usable with "iptables-restore -c"

bwmd

This daemon is the main bandwidth shaper, it connects to the kernel and intercepts all packets -j QUEUE'd for shaping.

-c or --**config=**<config_file>
Specify the configuration file to use, defaults to <prefix>/etc/bwm_tools/firewall.xml

-f or **--foreground**
Run bwmd in the foreground and print out debugging information.

bwm_monitor

This is the bandwidth monitor which connects to the bwmd daemon and allows one to view live bandwidth statistics. This utility has no options. The bwmd daemon must be running before using this utility.

bwm_graph

This utility handles the logged traffic and allows one to extract RRD datafiles and to optionally generate graphs at the same time. The usage of this utility is described later on in the document.

Configuring BWM Tools

Configuration of BWM Tools is done via an XML configuration file, this file is normally located in /etc/bwm_tools/firewall.xml

The layout of the file is pretty simple and is split up into various sections, these are detailed in the following sections...

The <global> section:

This section contains global tags pertaining to either the operation of BWM Tools or definitions used in other sections. These tags are detailed below...

<modules>: Module Management

This section is used to manage modules when bwmd starts. Valid tags within this one are described below.

<load />: Load a module

This tag allows us to load modules when bwmd starts.

Valid parameters are as follows...

name="..."

This is the name of the module to load

params="..."

Parameters to load module with

Here is how these tags can be used to load the ip_queue kernel module required by bwmd for shaping. Including ftp connection tracking to allow users to ftp through a tightly secured firewall.

```
<firewall>
        <global>
                <modules>
                        <load name="ip_queue"/>
                        <load name="ip_nat_ftp"/>
                        <load name="ip_conntrack_ftp"/>
                </modules>
        </global>
.
.
.
</firewall>
```

<class>: Class definitions
This tag contains matching specifications used in both firewalling, network address translation and bandwidth shaping/graphing/logging.

Valid parameters for this tag are as follows...

name="..."
This is the name of the class used in other tags throughout the configuration.

Valid tags within this one are described below.

<address />: Address/match specification
Valid parameters for this tag are specified below...

name="..."
This is a descriptive name for the address, isn't really used anywhere.

cmd-line="..."
Optional command line arguments for iptables, for example cmd-line="-m helper --helper <string>"

dst="..."
: Optional destination IP address

dst-iface="..."
: Optional destination interface

dst-port="..."
: Optional destination port

proto="..."
: Optional protocol specification, any valid protocol in /etc/protocols

src="..."
: Optional source IP address

src-iface="..."
: Optional source interface

src-port="..."
: Optional source port

Here is an example how the above tags it can be used to match connections over a specific number...

```
<firewall>
        <global>
        .
        .
        .
                <class name="excess_connections_to_webserver">
                        <address name="excess_to_server1"
                                dst="192.168.0.100" proto="tcp" dst-port="80"
                                 cmd-line="-m connlimit --connlimit-above 10"/>
                </class>
        </global>
.
.
.
</firewall>
```

The <acl> section:
This is basically the firewall section, you can add all your firewall rules here or

just leave it blank to use your current firewall.

The following tags are valid within this one...

<table>: Specify the table to work on
This tag is used to enclose the directives you plan to use with a specific table. Examples of tables are... filter, nat, mangle

There is only one parameter valid here..

name="..."
This is the name of the table we will be working with

Valid tags within this one can be found below...

<chain>: Specify a chain to work on
This tag is used to specify what chain the rules defined between the starting and ending tags apply to. Examples of already defined chains are `INPUT`, `OUTPUT` and `FORWARD`.

Valid parameters below...

name="..."
This is the name of the chain we will be working with

default="..."
This specifies the default target for the chain

Valid tags within this one can be found below...

<rule>: Specify a rule to apply to a set of classes
This tag is used to specify what classes apply to what rule, and are in order inserted into the actual iptables chains as iptables rules.

Valid parameters below...

name="..."
Optional name of rule

cmd-line="..."
Optional extra command line parameters to pass to iptables

target="..."
This is the target for the rule, used as the -j <target> parameter when generating iptables rules.

Between the opening and closing tags, classes defined in the <global> section are listed, these classify which traffic applies to which rule.

Multiple classes can be listed, one per line.

Using the above, here is an example of a simple firewall which allows http and ssh traffic, assuming your IP address is 10.0.0.2 of course...

```
<firewall>
        # Global configuration and access classes
        <global>
                <class name="http_traffic">
                        <address dst="10.0.0.2" proto="tcp" dst-port="80"/>
                </class>
                <class name="ssh_traffic">
                        <address dst="10.0.0.2" proto="tcp" dst-port="22"/>
                </class>
        </global>

        # Access control lists
        <acl>
                <table name="filter">
                        <chain name="INPUT" default="DROP">
                                <rule name="allowed_traffic" target="ACCEPT">
                                        http_traffic
                                        ssh_traffic
                                </rule>
                        </chain>
                        <chain name="FORWARD" default="DROP">
                        </chain>
                        <chain name="OUTPUT" default="ACCEPT">
                        </chain>
                </table>
        </acl>
</firewall>
```

The <nat> section:
The NAT section is used to define network address translation rules, these rules allow one to translate the source or destination IP address within packets. A common use for this is when a webserver is behind a firewall, requests are made

to a globally routable IP address and translated to the internal IP address of the webserver and visa versa.

There are 3 tags available, <snat>, <dnat> and <masq>, these three tags are used for source network address translation, destination address translation and masquerading respectively.

Valid options for these tags are as follows...

<snat>: Source address translation
SNAT is used for source network address translation, an example of which is again a webserver behind a firewall. Where SNAT comes in handy is when the webserver makes a query through the firewall, instead of the traffic on the internet comming from the webservers internal IP 192.168.1.100 which is not going to work, the firewall translates 192.168.1.100 to a globally routable IP address.

There are no parameters for this tag, although the following sub-tags and parameters are available...

<rule>: Specify a rule
This tag is used to specify what classes apply to what rule, and are in order inserted into the actual iptables chains as iptables rules.

This tag takes the following parameters...

name="..."
Optional name of rule

to-src="..."
Translate all traffic matched in the class specification to this source IP address.

Between the opening and closing tags, classes defined in the <global> section are listed, these classify which traffic applies to which rule.

Multiple classes can be listed, one per line.

<dnat>: Destination network address translation
DNAT is used for destination network address translation, an example of which is yet again a webserver behind a firewall. Where DNAT comes in handy is when requests are made to the webservers globally routable IP, this IP address is routed through the firewall and translated to the webservers internal IP address. Optional traffic filtering can be carried out

on the traffic, this is in most instances the case and prevents alot of harmfull traffic from interferring with the webservers operation.

There are no parameters for this tag, although the following sub-tags and parameters are available...

<rule>: Specify a rule
This tag is used to specify what classes apply to what rule, and are in order inserted into the actual iptables chains as iptables rules.

This tag takes the following parameters...

name="..."
Optional name of rule

to-dst="..."
Translate all traffic matched in the class specification to this source IP address.

Between the opening and closing tags, classes defined in the <global> section are listed, these classify which traffic applies to which rule.

Multiple classes can be listed, one per line.

<masq>: Masquerade
Masquerading is normally used for source address translation in the scenario where you have a dynamic IP and never know what address to do the translation to. An example of which is a home PC acting as a DSL router.

There are no parameters for this tag, although the following sub-tags and parameters are available...

<rule>: Specify a rule
This tag is used to specify what classes apply to what rule, and are in order inserted into the actual iptables chains as iptables rules.

This tag takes the following parameters...

name="..."
Optional name of rule

to-ports="..."
This specifies a range of source ports to use, overriding

the default SNAT source port-selection heuristics. For this parameter to work you MUST have defined a protocol in all the classes specified. For example proto="tcp".

Between the opening and closing tags, classes defined in the <global> section are listed, these classify which traffic applies to which rule.

Multiple classes can be listed, one per line.

An example using the above tags would look something like this...

```
<firewall>
        # Global configuration and access classes
        <global>
                <class name="traf_from_webserver">
                        <address src="192.168.0.100"/>
                </class>
                <class name="traf_to_webserver">
                        <address dst="<globally routable IP here>"/>
                </class>
        </global>

        # Network address translation
        <nat>
                <snat>
                        <rule to-src="<globally routable IP here>">
                                traf_from_webserver
                        </rule>
                </snat>
                <dnat>
                        <rule to-dst="192.168.0.100">
                                traf_to_webserver
                        </rule>
                </dnat>
        </nat>
</firewall>
```

Here is an example if you pc is acting as a DSL router...

```
<firewall>
        # Global configuration and access classes
        <global>
                <class name="traf_going_to_dsl">
                        <address src="192.168.0.0/24"/>
        </class>
    </global>

    # Network address translation
    <nat>
        <masq>
            <rule name="masq_traffic_going_out">
                traf_going_to_dsl
            </rule>
        </masq>
    </nat>
</firewall>
```

The <traffic> section:
This section is used to define traffic shaping rules. These traffic shaping rules are called flows, the concept of flows is a single-parent child relationship. For instance you can define 1 major flow, within this flow you can define separate priorities and limits for different traffic such as mail, browsing and p2p traffic. This example setup might be used for a DSL internet connection where one would like to prioritize internet browsing.

This tags have no parameters.

The below tags are valid within this section.

<flow>: Specify a traffic flow
This tag is used to specify a traffic flow and takes the following parameters...

name="..."
Mandatory flow name, this is used to identify the flow when reporting and monitoring

nfmark="..."
Mandatory/Optional parameter to specify the NFMARK of the traffic that applies to this flow. This must be used at the deepest level of flow embedding to match traffic. Each nfmark value MUST be unique!

stats-len="..."
Optional parameter to specify the period in seconds that the average bandwidth rate and packet rate is based on. If 0 is specified here there will be no average

queue-size="..."

Optional parameter to specify the size of the entire packet queue. If 0 is specified, queue size is unlimited. If -1 is specified, the queue will not be used.

queue-len="..."

Optional parameter to specify the maximum number of packets that can be in the entire queue at any one time. If -1 is specified the queue will not be used.

max-rate="..."

Optional parameter to specify the maximum rate in bytes/s before packets are queued, packets are not queued if they can be bursted. If 0 is specified, no traffic limiting will occur. If however the report-timeout="..." parameter is also specified then only logging will occur.

burst-rate="..."

Optional parameter to specify the maximum rate in bytes/s which packets can be bursted. Bursting can only occur until the parent has maxed out its max-rate. Unlimited bursting will occur when burst-rate = 0, remember unlimited meaning until the parent has maxed its max-rate. This value must be greater than max-rate.

burst-threshold="..."

Optional parameter to specify at what percentage we will stop bursting to our parent flow with regards to the parents current rate of usage. If this is set to 75, bursting to our parent will only be allowed until parent has maxed out 75% of its allowed maximum bandwidth utilization. If other flows max 70% of the parents bandwidth, we will be allowed to max our max-rate and burst until our parent reaches 75% of its max-rate. Remember burst-threshold pertains to the parents max-rate parameter, not the parents burst-rate.

report-timeout="..."

Optional parameter to specify if and in what time increments the traffic statistics are logged to file. For example, if this parameter is set to 60, bwmd will log traffic stats to file every 60 seconds. Minimum value for this parameter is 30.

prio-classifier="..."

Optional parameter to specify an automatic traffic prioritization classifier. This parameter defaults to the none classifier, where no prioritization takes place. Available classifiers are discussed below...

port:

Classifier With this classification prioritization happens automatically with the following ports mapped to their corrosponding priorities. (1 = highest, 100 = lowest)...

TCP Traffic

`port 113 (AUTH)'

`Priority 20'

`port 22, 23 (SSH, TELNET)'

`Priority 25'

`port 80, 443, 8080, 3128, 3130 (HTTP, HTTPS, PROXY PORTS)'

`Priority 65'

`port 2401 (CVS)'

`Priority 70'

`port 110, 143 (POP3, IMAP4)'

`Priority 75'

`port 20, 21 (FTP)'

`Priority 80'

UDP Traffic

`port 53 (DNS)'

`Priority 10'

`port 123 (NTP)'

`Priority 15'

`port 1645/6, 1812/3 (RADIUS)'

`Priority 30'

`port 33434-33465 (Normally traceroute)'

`Priority 5'

The default priority for traffic not matching any of the above is 50.

none:

This classifier This is the default classifier, no priorization will occur and all trafic will be dumped in the default priority 50 queue.

Between the opening and closing tags, classes defined in the <global> section can be listed, if you want to list multiple classes use one per line, these classes classify which traffic applies to which rule.

Please note listing classes is required only if you are using BWM Tools to generate your firewall for you, otherwise just make sure you MARK your traffic correctly and the MARK value matches the nfmark="..." parameter value used above.

Alternatively <flow> ... </flow> tags can be embedded to form a more complex hierarchy.

On a last note, if you are infact not using BWM Tools to generate your firewall and don't want to embed flows in multiple hierarchical levels you can specify the flow tag quickly in the following way <flow ... />.

To continue on the line of complexity, one can specify the following sub-tags, within the `<flow> ... </flow>` tags...

<queue>: This tag is used to finer tune queuing
This tag can be specified to finer tune into which queue the traffic is put and has the following parameters...

prio="..."
Mandatory parameter to specify the priority of the matched traffic. (1 = highest, 100 = lowest).

nfmark="..."
Mandatory parameter to specify the mark value of the traffic.

Below is an example of using the <queue> ... </queue> tags to give VNC traffic highest priority...

```
<flow name="line_in" max-rate="32000">
        <flow name="p2p_traffic_in" max-rate="8000" burst-
        rate="24000" nfmark="100">
                class_p2p_traffic_in
        </flow>
        <flow name="vnc_in" max-rate="24000" burst-rate="32000">
                <queue prio="1" nfmark="101">
                        class_vnc_in
                </queue>
        </flow>
</flow>
```

Between the opening and closing tags, classes defined in the

<global> section can be listed, if you want to list multiple classes use one per line, these classes classify which traffic applies to which rule or queue.

Please note listing classes is required only if you are using BWM Tools to generate your firewall for you, otherwise just make sure you MARK your traffic correctly and the MARK value matches the nfmark="..." parameter value used above.

On a last note, if you are infact not using BWM Tools to generate your firewall and want to specify a queue quickly, you can do so in the following way <queue ... />.

<group>: Group flows for reporting

The <group> tag is used for reporting only. It is for grouping flows together into 1 reporting name. This tag takes the following parameters...

name="..."
Mandatory flow name, this is used to identify the flow when reporting and monitoring.

report-timeout="..."
Optional parameter to specify if and in what time increments the traffic statistics are logged to file. For example, if this parameter is set to 60, bwmd will log traffic stats to file every 60 seconds.

Minimum value for this parameter is 30.

stats-len="..."
Optional parameter to specify the period in seconds that the average bandwidth rate and packet rate is based on. If 0 is specified here there will be no average.

Integrating BWM Tools with your system

This section will describe how to integrate BWM Tools into your system, be it you use BWM Tools to entirely manage your firewall, NAT and traffic shaping or just to do the traffic shaping.

There are two possible scenarios here detailed below...

Scenario 1:
You want to use BWM Tools for both your firewall and traffic shaping.

This is the easiest scenario to deal with, only having 4 steps below to get your firewall, NAT and traffic shaping up and running...

1. Configure your classes, ACL's, NAT and traffic shaping rules as described in the previous sections. The end target for all accepted traffic must be bwmd in the INPUT chain or OUTPUT chain if you doing single box or a router configuration respectively.

2. Run BWM Firewall to generate an `iptables-restore` compatible configuration file.

   BWM Firewall takes the BWM Tools XML configuration file and translates the various sections and tags into a firewall which can be loaded directly with iptables-restore.

   # bwm_firewall -c <config file> -f <output file>

3. Once you've generated the iptables restore file you must load it atomically into the kernel with the following command...

   # iptables-restore < output_file_above

4. The last step is to fire up `bwmd` with your choice of options.

   # bwmd -c <config file>

Scenario 2:
You want to use another firewalling application and have BWM Tools do only the traffic shaping.

Here there are a few things to remember...

1. BWM Tools works with the NFMARK parameter attached to packets. Marking packets can only be done in the mangle table in iptables.

2. BWM Tools uses the userpace queuing mechanism, all packets to be shaped must be targeted at QUEUE in the filter table. This is done by either adding a rule to the INPUT and OUTPUT chain in the case of a single box which you need to shape traffic to and from respectively. While in the case of a firewall where traffic passes through you would

add a rule to the FORWARD chain.

3. Therefore in order for BWM Tools to shape traffic, packets must be MARK'ed with a number corresponding to the number specified in the nfmark="..." parameter defined in the <flow> tag and targeted in iptables to QUEUE instead of ACCEPT as per above.

Imagine you would like your Linux router to rate limit all traffic from and to IP 192.168.1.100, an example of this can be found below...

Configuring iptables:

```
# iptables -t filter -A FORWARD -m mark ! --mark 0x0 -j QUEUE
# iptables -t mangle -A FORWARD -s 192.168.1.100 -j MARK  \
    --set-mark 100
# iptables -t mangle -A FORWARD -d 192.168.1.100 -j MARK  \
    --set-mark 101
```

Configuring bwmd:

```
<firewall>
    <global>
        <modules>
            <load name="ip_queue"/>
        </modules>
    </global>
    # Traffic flows
    <traffic>
        <flow name="pc_in" max-rate="64000" report-timeout="60"
                nfmark="100" />
        <flow name="pc_out" max-rate="64000" report-timeout="60"
                nfmark="101" />
    </traffic>
</firewall>
```

Graphing
BWM Tools supports graphing of traffic flows which have been specified with the report-timeout="".

Generating a graph can be achieved using bwm_graph or by using the RRD files generated by bwm_graph.

These two methods are discussed below...

Generating RRD files
The following section will explain how to have bwm_graph generate only RRD files and not graphs. This can be done quickly and simply using the following command-line options...

-f or **--flows:**

This option is used to specify the flows to include when generating the RRD files.

An example of this option can be found below...

```
# bwm_graph  \
        --flows="flow_name_1,flow_name2,flow_name3" ...
```

There is an optional parameter to specify which counter will be used when outputting the RRD file. For this there are 3 possibilities, all 3 are the totals per report-timeout="..." seconds specified in the relevant flow tag.

pkt
Number of packets processed

size_bit
Bits transferred in above period

size_byte
Bytes transferred

dropped
Packets dropped

bursted
Packets bursted

The counter to use is specified in the following manner...

```
# bwm_graph  \
        --flows="flow_name_1(size_bit),flow_name_2(size_byte)”
```

-s or **--start=**"YYYY/MM/DD HH:MM:SS"

This option is used to specify the date and/or time which to start our report from.

The format for date and/or time specification is yyyy/mm/dd hh:mm:ss.

An example of this option is as follows...

```
# bwm_graph ... --start="2003/01/20 01:20" ...
```

-e or **--end=**"YYYY/MM/DD HH:MM:SS"

This option is used to specify the date and/or time which our report will end.

The format for this option is the same as the -s and --start options.

An example of how to use all 3 above options to specify both the flows to work on and the reporting period can be done something like this...

```
# bwm_graph \
        --flows="flow_name_1(size_bit),flow_name_2(size_bit)"  \
        --start="2003/01/20" --end="2003/01/21"
```

Creating a pretty graph using bwm_graph

bwm_graph has a builtin interface to rrdtool. Using this interface one can easily have bwm_graph generate pretty looking graphs itself.

The graphing capability of bwm_graph is in addition to the generation of RRD files, meaning that you are required to use all 3 mandatory options discussed in "Generating RRD files" above.

The following graphing options can be used...

--graph-filename=<filename>

This parameter is used to specify an output filename for the generated .png image.

--graph-avg

Write counter averages on the graph.

--graph-date
Write the start datetime and end datetime of the reporting period on the graph.

--graph-title=<graph_title>
Specify a title for your graph

--graph-total
Write out counter totals on the graph

--graph-vert-title=<graph_title>
Specify a vertical title for the graph

RRDTool

What is RRDtool?

RRDtool refers to Round Robin Database tool. Round robin is a technique that works with a fixed amount of data, and a pointer to the current element. Think of a circle with some dots plotted on the edge -- these dots are the places where data can be stored. Draw an arrow from the center of the circle to one of the dots -- this is the pointer. When the current data is read or written, the pointer moves to the next element. As we are on a circle there is neither a beginning nor an end, you can go on and on and on. After a while, all the available places will be used and the process automatically reuses old locations. This way, the dataset will not grow in size and therefore requires no maintenance. RRDtool works with with Round Robin Databases (RRDs). It stores and retrieves data from them.

What data can be put into an RRD?

You name it, it will probably fit as long as it is some sort of time-series data. This means you have to be able to measure some value at several points in time and provide this information to RRDtool. If you can do this, RRDtool will be able to store it. The values must be numerical but don't have to be integers.

What can I do with this tool?

RRDtool originated from MRTG (Multi Router Traffic Grapher). MRTG started as a tiny little script for graphing the use of a university's connection to the Internet. MRTG was later (ab-)used as a tool for graphing other data sources including temperature, speed, voltage, number of printouts and the like.
Most likely you will start to use RRDtool to store and process data collected via SNMP. The data will most likely be bytes (or bits) transfered from and to a network or a computer. But it can also be used to display tidal waves, solar radiation, power consumption, number of visitors at an exhibition, noise levels near an airport, temperature on your favorite holiday location, temperature in the fridge and whatever you imagination can come up with.

You only need a sensor to measure the data and be able to feed the numbers into RRDtool. RRDtool then lets you create a database, store data in it, retrieve that data and create graphs in PNG format for display on a web browser. Those PNG images are dependent on the data you collected and could be, for instance, an overview of the average network usage, or the peaks that occurred.

It is pretty easy to gather status information from all sorts of things, ranging from the temperature in your office to the number of octets which have passed through the FDDI interface of your router. But it is not so trivial to store this data in an efficient and systematic manner. This is where RRDtool comes in handy. It lets you *log and analyze* the data you gather from all kinds of data-sources (DS). The data analysis part of RRDtool is based on the ability to quickly generate graphical representations of the data values collected over a definable time period.

Functions

While the man pages talk of command line switches you have to set in order to make RRDtool work it is important to note that RRDtool can be remotely controlled through a set of pipes. This saves a considerable amount of startup time when you plan to make RRDtool do a lot of things quickly. There is also a number of language bindings for RRDtool which allow you to use it directly from perl, python, tcl, php, etc.

create

Set up a new Round Robin Database (RRD).

update

Store new data values into an RRD.

updatev

Operationally equivalent to update except for output.

graph

Create a graph from data stored in one or several RRDs. Apart from generating graphs, data can also be extracted to stdout.

dump

Dump the contents of an RRD in plain ASCII. In connection with restore you can use this to move an RRD from one computer architecture to another.

restore

Restore an RRD in XML format to a binary RRD.

fetch

Get data for a certain time period from a RRD. The graph function uses fetch to retrieve its data from an RRD.

tune

Alter setup of an RRD.

last

Find the last update time of an RRD.

info

Get information about an RRD.

rrdresize
Change the size of individual RRAs. This is dangerous!

xport
Export data retrieved from one or several RRDs.

rrdcgi
This is a standalone tool for producing RRD graphs on the fly.

How does rrdtool work?

Data Acquisition

When monitoring the state of a system, it is convenient to have the data available at a constant time interval. Unfortunately, you may not always be able to fetch data at exactly the time you want to. Therefore RRDtool lets you update the logfile at any time you want. It will automatically interpolate the value of the data-source (DS) at the latest official time-slot (intervall) and write this interpolated value to the log. The original value you have supplied is stored as well and is also taken into account when interpolating the next log entry.

Consolidation

You may log data at a 1 minute interval, but you might also be interested to know the development of the data over the last year. You could do this by simply storing the data in 1 minute intervals for the whole year. While this would take considerable disk space it would also take a lot of time to analyze the data when you wanted to create a graph covering the whole year. RRDtool offers a solution to this problem through its data consolidation feature. When setting up an Round Robin Database (RRD), you can define at which interval this consolidation should occur, and what consolidation function (CF) (average, minimum, maximum, total, last) should be used to build the consolidated values (see rrdcreate). You can define any number of different consolidation setups within one RRD. They will all be maintained on the fly when new data is loaded into the RRD.

Round Robin Archives

Data values of the same consolidation setup are stored into Round Robin Archives (RRA). This is a very efficient manner to store data for a certain amount of time, while using a known and constant amount of storage space.

It works like this: If you want to store 1'000 values in 5 minute interval, RRDtool will allocate space for 1'000 data values and a header area. In the header it will store a pointer telling which slots (value) in the storage area was last written to. New values are written to the Round Robin Archive in, you guessed it, a round robin manner. This automatically limits the history to the last 1'000 values (in our example). Because you can define several RRAs within a single RRD, you can setup another one, for storing 750 data values at a 2 hour interval, for example, and thus keep a log for the last two months at a lower resolution.

The use of RRAs guarantees that the RRD does not grow over time and that old data is automatically eliminated. By using the consolidation feature, you can still keep data for a very long time, while gradually reducing the resolution of the data along the time axis.

Using different consolidation functions (CF) allows you to store exactly the type of information that actually interests you: the maximum one minute traffic on the LAN, the minimum temperature of your wine cellar, the total minutes of down time, etc.

Unknown Data
As mentioned earlier, the RRD stores data at a constant interval. Sometimes it may happen that no new data is available when a value has to be written to the RRD. Data acquisition may not be possible for one reason or other. With RRDtool you can handle these situations by storing an **UNKNOWN** value into the database. The value '**UNKNOWN**' is supported through all the functions of the tool. When consolidating a data set, the amount of **UNKNOWN** data values is accounted for and when a new consolidated value is ready to be written to its Round Robin Archive (RRA), a validity check is performed to make sure that the percentage of unknown values in the data point is above a configurable level. If not, an **UNKNOWN** value will be written to the RRA.

Graphing
RRDtool allows you to generate reports in numerical and graphical form based on the data stored in one or several RRDs. The graphing feature is fully configurable. Size, color and contents of the graph can be defined freely.

rrdcreate

Usage

rrdtool create *filename* [**--start|-b** *start_time*] [**--step|-s** *step*] [**DS:***ds-name***:***DST***:***dst_arguments*] [**RRA:***CF***:***cf_arguments*]

Options

The create function of RRDtool lets you set up new Round Robin Database (RRD) files. The file is created at its final, full size and filled with **UNKNOWN** data.

filename

The name of the RRD you want to create. RRD files should end with the extension *.rrd*. However, RRDtool will accept any filename.

--start|-b *start time* (default: now - 10s)

Specifies the time in seconds since 1970-01-01 UTC when the first value should be added to the RRD. RRDtool will not accept any data timed before or at the time specified.
See also AT-STYLE TIME SPECIFICATION section in the *rrdfetch* documentation for other ways to specify time.

--step|-s *step* (default: 300 seconds)

Specifies the base interval in seconds with which data will be fed into the RRD.

DS:*ds-name***:***DST***:***dst_arguments*

A single RRD can accept input from several data sources (DS), for example incoming and outgoing traffic on a specific communication line. With the DS configuration option you must define some basic properties of each data source you want to store in the RRD.

ds-name

The name you will use to reference this particular data source from an RRD. A *ds-name* must be 1 to 19 characters long in the characters [a-zA-Z0-9_].

DST

This defines the Data Source Type. The remaining arguments of a data source entry depend on the data source type. For GAUGE, COUNTER, DERIVE, and ABSOLUTE the format for a data source entry is:

**DS:ds-name:GAUGE | COUNTER | DERIVE | \
ABSOLUTE:heartbeat:min:max**

For COMPUTE data sources, the format is:

DS:*ds-name*:*COMPUTE*:*rpn-expression*

In order to decide which data source type to use, review the definitions that follow. Also consult the section on ``HOW TO MEASURE'' for further insight.

Data sources:

GAUGE

This is used for things like temperatures or number of people in a room or the value of a RedHat share.

COUNTER

This is for continuous incrementing counters like the ifInOctets counter in a router. The COUNTER data source assumes that the counter never decreases, except when a counter overflows.

The update function takes the overflow into account. The counter is stored as a per-second rate. When the counter overflows, RRDtool checks if the overflow happened at the 32bit or 64bit border and acts accordingly by adding an appropriate value to the result.

DERIVE

will store the derivative of the line going from the last to the current value of the data source. This can be useful for gauges, for example, to measure the rate of people entering or leaving a room. Internally, derive works exactly like COUNTER but without overflow checks. So if your counter does not reset at 32 or 64 bit you might want to use DERIVE and combine it with a MIN value of 0.

NOTE on COUNTER vs DERIVE

If you cannot tolerate ever mistaking the occasional counter reset for a legitimate counter wrap, and would prefer ``Unknowns'' for all legitimate counter wraps and resets, always use DERIVE with min=0. Otherwise, using COUNTER with a suitable max will return correct values for all legitimate counter wraps, mark some counter resets as ``Unknown'', but can mistake some counter resets for a legitimate counter wrap.

For a 5 minute step and 32-bit counter, the probability of mistaking a counter reset for a legitimate wrap is arguably about 0.8% per 1Mbps of maximum bandwidth. Note that this equates to 80% for 100Mbps interfaces, so for high bandwidth interfaces and a 32bit counter, DERIVE with min=0 is probably preferable. If you are using a 64bit counter, just about any max setting will eliminate the possibility of mistaking a reset for a counter wrap.

ABSOLUTE

We use this for counters which get reset upon reading. This is used for fast counters which tend to overflow. So instead of reading them normally you reset them after every read to make sure you have a maximum time available before the next overflow. Another usage is for things you count like number of messages since the last update.

COMPUTE

This for storing the result of a formula applied to other data sources in the RRD. This data source is not supplied a value on update, but rather its Primary Data Points (PDPs) are computed from the PDPs of the data sources according to the rpn-expression that defines the formula. Consolidation functions are then applied normally to the PDPs of the COMPUTE data source (that is the rpn-expression is only applied to generate PDPs). In database software, such data sets are referred to as ``virtual'' or ``computed'' columns.

heartbeat

Defines the maximum number of seconds that may pass between two updates of this data source before the value of the data source is assumed to be **UNKNOWN**.

min* and *max

Defines the expected range values for data supplied by a data source. If *min* and/or *max* any value outside the defined range will be regarded as **UNKNOWN**. If you do not know or care about min and max, set them to U for unknown. Note that min and max always refer to the processed values of the DS. For a traffic-COUNTER type DS this would be the maximum and minimum data-rate expected from the device.

If information on minimal/maximal expected values is available, always set the min and/or max properties. This will help RRDtool in doing a simple sanity check on the data supplied when running update.

rpn-expression

Defines the formula used to compute the PDPs of a COMPUTE data source from other data sources in the same <RRD>. It is similar to defining a CDEF argument for the graph command. Please refer to that manual page for a list and description of RPN operations supported. For COMPUTE data sources, the following RPN operations are not supported: COUNT, PREV, TIME, and LTIME. In addition, in defining the RPN expression, the COMPUTE data source may only refer to the names of data source listed previously in the create command. This is similar to the restriction that CDEFs must refer only to DEFs and CDEFs previously defined in the same graph command.

RRA:*CF***:***cf arguments*

The purpose of an RRD is to store data in the round robin archives (RRA). An archive consists of a number of data values or statistics for each of the defined data-sources (DS) and is defined with an RRA line.

When data is entered into an RRD, it is first fit into time slots of the length defined with the -s option, thus becoming a *primary data point*.

The data is also processed with the consolidation function (*CF*) of the archive. There are several consolidation functions that consolidate primary data points via an aggregate function: AVERAGE, MIN, MAX, LAST. The format of RRA line for these consolidation functions is:

RRA:*AVERAGE | MIN | MAX | LAST*:*xff*:*steps*:*rows*

xff The xfiles factor defines what part of a consolidation interval may be made up from **UNKNOWN** data while the consolidated value is still regarded as known. It is given as the ratio of allowed **UNKNOWN** PDPs to the number of PDPs in the interval. Thus, it ranges from 0 to 1 (exclusive).

steps defines how many of these *primary data points* are used to build a *consolidated data point* which then goes into the archive. R*ows* define how many generations of data values are kept in an RRA.

Aberrant behavior detection with holt-winters forecasting

In addition to the aggregate functions, there are a set of specialized functions that enable RRDtool to provide data smoothing (via the Holt-Winters forecasting algorithm), confidence bands, and the flagging aberrant behavior in the data source time series:

RRA:*HWPREDICT*:*rows*:*alpha*:*beta*:*seasonal period*[:*rra-num*]
RRA:*SEASONAL*:*seasonal period*:*gamma*:*rra-num*
RRA:*DEVSEASONAL*:*seasonal period*:*gamma*:*rra-num*
RRA:*DEVPREDICT*:*rows*:*rra-num*
RRA:*FAILURES*:*rows*:*threshold*:*window length*:*rra-num*

These RRAs differ from the true consolidation functions in several ways. First, each of the RRAs is updated once for every primary data point. Second, these RRAs are interdependent. To generate real-time confidence bounds, a matched set of HWPREDICT, SEASONAL, DEVSEASONAL, and DEVPREDICT must exist. Generating smoothed values of the primary data points requires both a HWPREDICT RRA and SEASONAL RRA. Aberrant behavior detection requires FAILURES, HWPREDICT, DEVSEASONAL, and SEASONAL.

The actual predicted, or smoothed, values are stored in the HWPREDICT RRA. The predicted deviations are stored in DEVPREDICT (think a standard deviation which can be scaled to yield a confidence band). The FAILURES RRA stores binary indicators. A 1 marks the indexed observation as failure; that is, the number of confidence bounds violations in the preceding window of observations met or exceeded a specified threshold.

The SEASONAL and DEVSEASONAL RRAs store the seasonal coefficients for the Holt-Winters forecasting algorithm and the seasonal deviations, respectively. There is one entry per observation time point in the seasonal cycle. For example, if primary data points are generated every five minutes and the seasonal cycle is 1 day, both SEASONAL and DEVSEASONAL will have 288 rows.

In order to simplify the creation for the novice user, in addition to supporting explicit creation of the HWPREDICT, SEASONAL, DEVPREDICT, DEVSEASONAL, and FAILURES RRAs, the RRDtool create command supports implicit creation of the other four when HWPREDICT is specified alone and the final argument *rra-num* is omitted.

rows

This specifies the length of the RRA prior to wrap around. Remember that there is a one-to-one correspondence between primary data points and entries in these RRAs. For the HWPREDICT CF, *rows* should be larger than the *seasonal period*. If the DEVPREDICT RRA is implicitly created, the default number of rows is the same as the HWPREDICT *rows* argument. If the FAILURES RRA is implicitly created, *rows* will be set to the *seasonal period* argument of the HWPREDICT RRA. Of course, the RRDtool *resize* command is available if these defaults are not sufficient and the creator wishes to avoid explicit creations of the other specialized function RRAs.

seasonal period

This is the number of primary data points in a seasonal cycle. If SEASONAL and DEVSEASONAL are implicitly created, this argument for those RRAs is set automatically to the value specified by HWPREDICT. If they are explicitly created, the creator should verify that all three *seasonal period* arguments agree.

alpha

This is the adaption parameter of the intercept (or baseline) coefficient in the Holt-Winters forecasting algorithm. *alpha* must lie between 0 and 1. A value closer to 1 means that more recent observations carry greater weight in predicting the baseline component of the forecast. A value closer to 0 means that past history carries greater weight in predicting the baseline component.

beta

This is the adaption parameter of the slope (or linear trend) coefficient in the Holt-Winters forecasting algorithm. *beta* must lie between 0 and 1 and plays the same role as *alpha* with respect to the predicted linear trend.

gamma

This is the adaption parameter of the seasonal coefficients in the Holt-Winters forecasting algorithm (HWPREDICT) or the adaption parameter in the exponential smoothing update of the seasonal deviations. It must lie between 0 and 1. If the SEASONAL and DEVSEASONAL RRAs are created implicitly, they will both have the same value for *gamma*: the value specified for the HWPREDICT *alpha* argument. Note that because there is one seasonal coefficient (or deviation) for each time point during the seasonal cycle, the adaptation rate is much slower than the baseline. Each seasonal coefficient is only updated (or adapts) when the observed value occurs at the offset in the seasonal cycle corresponding to that coefficient.

If SEASONAL and DEVSEASONAL RRAs are created explicitly, *gamma* need not be the same for both. Note that *gamma* can also be changed via the RRDtool *tune* command.

rra-num

This parameter provides the links between related RRAs. If HWPREDICT is specified alone and the other RRAs are created implicitly, then there is no need to worry about this argument. If RRAs are created explicitly, then carefully pay attention to this argument. For each RRA which includes this argument, there is a dependency between that RRA and another RRA. The *rra-num* argument is the 1-based index in the order of RRA creation (that is, the order they appear in the *create* command). The dependent RRA for each RRA requiring the *rra-num* argument is listed here:

HWPREDICT *rra-num* is the index of the SEASONAL RRA.
SEASONAL *rra-num* is the index of the HWPREDICT RRA.
DEVPREDICT *rra-num* is the index of the DEVSEASONAL RRA.
DEVSEASONAL *rra-num* is the index of the HWPREDICT RRA.
FAILURES *rra-num* is the index of the DEVSEASONAL RRA.

threshold

Is the minimum number of violations (observed values outside the confidence bounds) within a window that constitutes a failure. If the FAILURES RRA is implicitly created, the default value is 7.

window length

This is the number of time points in the window. Specify an integer greater than or equal to the threshold and less than or equal to 28. The time interval this window represents depends on the interval between primary data points. If the FAILURES RRA is implicitly created, the default value is 9.

The heartbeat and the step

Here is an explanation by Don Baarda on the inner workings of RRDtool. It may help you to sort out why all this *UNKNOWN* data is popping up in your databases:

RRDtool gets fed samples at arbitrary times. From these it builds Primary Data Points (PDPs) at exact times on every ``step'' interval. The PDPs are then accumulated into RRAs.

The ``heartbeat'' defines the maximum acceptable interval between samples. If the interval between samples is less than ``heartbeat'', then an average rate is calculated and applied for that interval. If the interval between samples is longer than ``heartbeat'', then that entire interval is considered ``unknown''. Note that there are other things that can make a sample interval ``unknown'', such as the rate exceeding limits, or even an ``unknown'' input sample.

The known rates during a PDP's ``step'' interval are used to calculate an average rate for that PDP. Also, if the total ``unknown'' time during the ``step'' interval exceeds the ``heartbeat'', the entire PDP is marked as ``unknown''. This means that a mixture of known and ``unknown'' sample times in a single PDP ``step'' may or may not add up to enough ``unknown'' time to exceed ``heartbeat'' and hence mark the whole PDP ``unknown''. So ``heartbeat'' is not only the maximum acceptable interval between samples, but also the maximum acceptable amount of ``unknown'' time per PDP (obviously this is only significant if you have ``heartbeat'' less than ``step'').

The ``heartbeat'' can be short (unusual) or long (typical) relative to the ``step'' interval between PDPs. A short ``heartbeat'' means you require multiple samples per PDP, and if you don't get them mark the PDP unknown. A long heartbeat can span multiple ``steps'', which means it is acceptable to have multiple PDPs calculated from a single sample. An extreme example of this might be a ``step'' of 5 minutes and a ``heartbeat'' of one day, in which case a single sample every day will result in all the PDPs for that entire day period being set to the same average rate.

How to measure

Here are a few hints on how to measure:

Temperature

Usually you have some type of meter you can read to get the temperature. The temperature is not really connected with a time. The only connection is that the temperature reading happened at a certain time. You can use the GAUGE data source type for this. RRDtool will then record your reading together with the time.

Mail Messages

Assume you have a method to count the number of messages transported by your mailserver in a certain amount of time, giving you data like '5 messages in the last 65 seconds'. If you look at the count of 5 like an ABSOLUTE data type you can simply update the RRD with the number 5 and the end time of your monitoring period. RRDtool will then record the number of messages per second. If at some later stage you want to know the number of messages transported in a day, you can get the average messages per second from RRDtool for the day in question and multiply this number with the number of seconds in a day. Because all math is run with Doubles, the precision should be acceptable.

It's always a Rate

RRDtool stores rates in amount/second for COUNTER, DERIVE and ABSOLUTE data. When you plot the data, you will get on the y axis amount/second which you might be tempted to convert to an absolute amount by multiplying by the delta-time between the points. RRDtool plots continuous data, and as such is not appropriate for plotting absolute amounts as for example ``total bytes'' sent and received in a router. What you probably want is plot rates that you can scale to bytes/hour, for example, or plot absolute amounts with another tool that draws bar-plots, where the delta-time is clear on the plot for each point (such that when you read the graph you see for example GB on the y axis, days on the x axis and one bar for each day).

Example 1

```
# rrdtool create temperature.rrd --step 300 \
          DS:temp:GAUGE:600:-273:5000 \
          RRA:AVERAGE:0.5:1:1200 \
          RRA:MIN:0.5:12:2400 \
          RRA:MAX:0.5:12:2400 \
          RRA:AVERAGE:0.5:12:2400
```

This sets up an RRD called *temperature.rrd* which accepts one temperature value every 300 seconds. If no new data is supplied for more than 600 seconds, the temperature becomes **UNKNOWN**. The minimum acceptable value is -273 and the maximum is 5'000.

A few archive areas are also defined. The first stores the temperatures supplied for 100 hours (1'200 * 300 seconds = 100 hours). The second RRA stores the minimum temperature recorded over every hour (12 * 300 seconds = 1 hour), for 100 days (2'400 hours). The third and the fourth RRA's do the same for the maximum and average temperature, respectively.

Example 2

```
# rrdtool create monitor.rrd --step 300 \
          DS:ifOutOctets:COUNTER:1800:0:4294967295 \
          RRA:AVERAGE:0.5:1:2016 \
          RRA:HWPREDICT:1440:0.1:0.0035:288
```

This example is a monitor of a router interface. The first RRA tracks the traffic flow in octets; the second RRA generates the specialized functions RRAs for aberrant behavior detection. Note that the *rra-num* argument of HWPREDICT is missing, so the other RRAs will implicitly be created with default parameter values. In this example, the forecasting algorithm baseline adapts quickly; in fact the most recent one hour of observations (each at 5 minute intervals) accounts for 75% of the baseline prediction. The linear trend forecast adapts much more slowly. Observations made during the last day (at 288 observations per day) account for only 65% of the predicted linear trend. Note: these computations rely on an exponential smoothing formula described in the LISA 2000 paper.

The seasonal cycle is one day (288 data points at 300 second intervals), and the seasonal adaption parameter will be set to 0.1. The RRD file will store 5 days (1'440 data points) of forecasts and deviation predictions before wrap around. The file will store 1 day (a seasonal cycle) of 0-1 indicators in the FAILURES RRA.

The same RRD file and RRAs are created with the following command, which explicitly creates all specialized function RRAs.

```
# rrdtool create monitor.rrd --step 300 \
          DS:ifOutOctets:COUNTER:1800:0:4294967295 \
          RRA:AVERAGE:0.5:1:2016 \
          RRA:HWPREDICT:1440:0.1:0.0035:288:3 \
          RRA:SEASONAL:288:0.1:2 \
          RRA:DEVPREDICT:1440:5 \
          RRA:DEVSEASONAL:288:0.1:2 \
          RRA:FAILURES:288:7:9:5
```

Of course, explicit creation need not replicate implicit create, a number of arguments could be changed.

Example 3

```
rrdtool create proxy.rrd --step 300 \
        DS:Total:DERIVE:1800:0:U  \
        DS:Duration:DERIVE:1800:0:U  \
        DS:AvgReqDur:COMPUTE:Duration,Requests,0,EQ,1,Requests,IF,/ \
        RRA:AVERAGE:0.5:1:2016
```

This example is monitoring the average request duration during each 300 sec interval for requests processed by a web proxy during the interval. In this case, the proxy exposes two counters, the number of requests processed since boot and the total cumulative duration of all processed requests. Clearly these counters both have some rollover point, but using the DERIVE data source also handles the reset that occurs when the web proxy is stopped and restarted.
In the RRD, the first data source stores the requests per second rate during the interval. The second data source stores the total duration of all requests processed during the interval divided by 300. The COMPUTE data source divides each PDP of the AccumDuration by the corresponding PDP of TotalRequests and stores the average request duration. The remainder of the RPN expression handles the divide by zero case.

rrdupdate

Usage

rrdtool {**update** | **updatev**} *filename* [**--template**|**-t** *ds-name*[**:***ds-name*]...] **N**|*timestamp***:***value*[**:***value*...] *at-timestamp***@***value*[**:***value*...] [*timestamp***:***value*[**:***value*...] ...]

Options

The update function feeds new data values into an RRD. The data is time aligned (interpolated) according to the properties of the RRD to which the data is written.

updatev

This alternate version of update takes the same arguments and performs the same function. The *v* stands for *verbose,* which describes the output returned. updatev returns a list of any and all consolidated data points (CDPs) written to disk as a result of the invocation of update. The values are indexed by timestamp (time_t), RRA (consolidation function and PDPs per CDP), and data source (name). Note that depending on the arguments of the current and previous call to update, the list may have no entries or a large number of entries.

filename

The name of the RRD you want to update.

--template|-t *ds-name*[**:***ds-name*]...

By default, the update function expects its data input in the order the data sources are defined in the RRD, excluding any COMPUTE data sources (i.e. if the third data source DST is COMPUTE, the third input value will be mapped to the fourth data source in the RRD and so on). This is not very error resistant, as you might be sending the wrong data into an RRD.

The template switch allows you to specify which data sources you are going to update and in which order. If the data sources specified in the template are not available in the RRD file, the update process will abort with an error message.

While it appears possible with the template switch to update data sources asynchronously, RRDtool implicitly assigns non-COMPUTE data sources missing from the template the **UNKNOWN** value.

Do not specify a value for a COMPUTE DST in the update function. If this is done accidentally (and this can only be done using the template switch), RRDtool will ignore the value specified for the COMPUTE DST.

N|*timestamp***:***value*[**:***value*...]

The data used for updating the RRD was acquired at a certain time. This time can either be defined in seconds since 1970-01-01 or by using the letter 'N', in which case the update time is set to be the current time. Negative time values are subtracted from the current time. An AT_STYLE TIME SPECIFICATION may also be used by delimiting the end of the time specification with the '@' character instead of a ':'. Getting the timing right to the second is especially important when you are working with data-sources of type COUNTER, DERIVE or ABSOLUTE.

The remaining elements of the argument are DS updates. The order of this list is the same as the order the data sources were defined in the RRA. If there is no data for a certain data-source, the letter U (e.g., N:0.1:U:1) can be specified.

The format of the value acquired from the data source is dependent on the data source type chosen. Normally it will be numeric, but the data acquisition modules may impose their very own parsing of this parameter as long as the colon (:) remains the data source value separator.

Example

# rrdtool update demo1.rrd N:3.44:3.15:U:23

Update the database file demo1.rrd with 3 known and one **UNKNOWN** value. Use the current time as the update time.

# rrdtool update demo2.rrd 887457267:U 887457521:22 887457903:2.7

Update the database file demo2.rrd which expects data from a single data-source, three times. First with an **UNKNOWN** value then with two regular readings. The update interval seems to be around 300 seconds.

rrdgraph

Usage

rrdtool graph *filename* [option...] [data_definition...] [data_calculation...] [variable_definition...] [graph_element ...] [print_element...]

Description

The graph function of RRDtool is used to present the data from an RRD to a human viewer. Its main purpose is to create a nice graphical representation, but it can also generate a numerical report.

Overview

rrdtool graph needs data to work with, so you must use one or more data_definition statements to collect this data. You are not limited to one database, it's perfectly legal to collect data from two or more databases (one per statement, though).

If you want to display averages, maxima, percentiles, etcetera it is best to collect them now using the variable_definition statement. Currently this makes no difference, but in a future version of rrdtool you may want to collect these values before consolidation.

The data fetched from the RRA is then consolidated so that there is exactly one datapoint per pixel in the graph. If you do not take care yourself, RRDtool will expand the range slightly if necessary. Note, in that case the first and/or last pixel may very well become unknown!

Sometimes data is not exactly in the format you would like to display it. For instance, you might be collecting bytes per second, but want to display bits per second. This is what the data_calculation command is designed for. After consolidating the data, a copy is made and this copy is modified using a rather powerful RPN command set.

When you are done fetching and processing the data, it is time to graph it (or print it). This ends the rrdtool graph sequence.

Options

filename

The name and path of the graph to generate. It is recommended to end this in `.png`, `.svg` or `.eps`, but RRDtool does not enforce this.

filename can be `'-'` to send the image to `stdout`. In this case, no other output is generated.

Time range

[-s|--start *time*] [-e|--end *time*] [-S|--step *seconds*]

The start and end of the time series you would like to display, and which RRA the data should come from. Defaults are: 1 day ago until now, with the best possible resolution. Start and end can be specified in several formats. By default, rrdtool graph calculates the width of one pixel in the time domain and tries to get data from an RRA with that resolution. With the step option you can alter this behaviour. If you want rrdtool graph to get data at a one-hour resolution from the RRD, set step to 3'600. Note: a step smaller than one pixel will silently be ignored.

Labels

[**-t|--title** *string*] [**-v|--vertical-label** *string*]

A horizontal string at the top of the graph and/or a vertically placed string at the left hand side of the graph.

Size

[**-w|--width** *pixels*] [**-h|--height** *pixels*] [**-j|--only-graph**]

The width and height of the canvas (the part of the graph with the actual data and such). This defaults to 400 pixels by 100 pixels.

If you specify the --only-graph option and set the height < 32 pixels you will get a tiny graph image (thumbnail) to use as an icon for use in an overview, for example. All labeling will be stripped off the graph.

Limits

[**-u|--upper-limit** *value*] [**-l|--lower-limit** *value*] [**-r|--rigid**]

By default the graph will be autoscaling so that it will adjust the y-axis to the range of the data. You can change this behaviour by explicitly setting the limits. The displayed y-axis will then range at least from lower-limit to upper-limit. Autoscaling will still permit those boundaries to be stretched unless the rigid option is set.

[**-A|--alt-autoscale**]

Sometimes the default algorithm for selecting the y-axis scale is not satisfactory. Normally the scale is selected from a predefined set of ranges and this fails miserably when you need to graph something like `260 + 0.001 * sin(x)`. This option calculates the minimum and maximum y-axis from the actual minimum and maximum data values. Our example would display slightly less than `260-0.001` to slightly more than `260+0.001` (this feature was contributed by Sasha Mikheev).

[**-M|--alt-autoscale-max**]

Where `--alt-autoscale` will modify both the absolute maximum AND minimum values, this option will only affect the maximum value. The minimum value, if not defined on the command line, will be 0. This option can be useful when graphing router traffic when the WAN line uses compression, and thus the throughput may be higher than the WAN line speed.

[**-N|--no-gridfit**]

In order to avoid anti-aliasing effects gridlines are placed on integer pixel values. This is by default done by extending the scale so that gridlines happens to be spaced using an integer number of pixels and also start on an integer pixel value. This might extend the scale too much for some logarithmic scales and for linear scales where --alt-autoscale is needed. Using --no-gridfit disables modification of the scale.

Grid

X-Axis

[-x|--x-grid *GTM***:***GST***:***MTM***:***MST***:***LTM***:***LST***:***LPR***:***LFM***]**
[-x|--x-grid none]

The x-axis label is quite complex to configure. If you don't have very special needs it is probably best to rely on the autoconfiguration to get this right. You can specify the string `none` to suppress the grid and labels altogether.

The grid is defined by specifying a certain amount of time in the *?TM* positions. You can choose from `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH` or `YEAR`. Then you define how many of these should pass between each line or label. This pair (*?TM:?ST*) needs to be specified for the base grid (*G??*), the major grid (*M??*) and the labels (*L??*). For the labels you also must define a precision in *LPR* and a *strftime* format string in *LFM*. *LPR* defines where each label will be placed. If it is zero, the label will be placed right under the corresponding line (useful for hours, dates etcetera). If you specify a number of seconds here the label is centered on this interval (useful for Monday, January etcetera).

--x-grid MINUTE:10:HOUR:1:HOUR:4:0:%X

This places grid lines every 10 minutes, major grid lines every hour, and labels every 4 hours. The labels are placed under the major grid lines as they specify exactly that time.

--x-grid HOUR:8:DAY:1:DAY:1:0:%A

This places grid lines every 8 hours, major grid lines and labels each day. The labels are placed exactly between two major grid lines as they specify the complete day and not just midnight.

Y-Axis

[-y|--y-grid *grid step***:***label factor***]**
[-y|--y-grid none]

Y-axis grid lines appear at each *grid step* interval. Labels are placed every *label factor* lines. You can specify `-y none` to suppress the grid and labels altogether. The default for this option is to automatically select sensible values.

[**-Y|--alt-y-grid**]
Place the Y grid dynamically based on the graph's Y range. The algorithm ensures that you always have a grid, that there are enough but not too many grid lines, and that the grid is metric. That is the grid lines are placed every 1, 2, 5 or 10 units. This parameter will also ensure that you get enough decimals displayed even if your graph goes from 69.998 to 70.001. (contributed by Sasha Mikheev).

[**-o|--logarithmic**]
Logarithmic y-axis scaling.

[**-X|--units-exponent** *value*]
This sets the 10**exponent scaling of the y-axis values. Normally, values will be scaled to the appropriate units (k, M, etc.). However, you may wish to display units always in k (Kilo, 10e3) even if the data is in the M (Mega, 10e6) range, for instance. Value should be an integer which is a multiple of 3 between -18 and 18 inclusively. It is the exponent on the units you wish to use. For example, use 3 to display the y-axis values in k (Kilo, 10e3, thousands), use -6 to display the y-axis values in u (Micro, 10e-6, millionths). Use a value of 0 to prevent any scaling of the y-axis values.

This option is very effective at confusing the heck out of the default rrdtool autoscaler and grid painter. If rrdtool detects that it is not successful in labeling the graph under the given circumstances, it will switch to the more robust --alt-y-grid mode.

[**-L|--units-length** *value*]
How many digits should rrdtool assume the y-axis labels to be? You may have to use this option to make enough space once you start fideling with the y-axis labeling.

Miscellaneous

[**-z|--lazy**]
Only generate the graph if the current graph is out of date or not existent.

[**-f**|**--imginfo** *printfstr*]

After the image has been created, the graph function uses printf together with this format string to create output similar to the PRINT function, only that the printf function is supplied with the parameters *filename*, *xsize* and *ysize*. In order to generate an IMG tag suitable for including the graph into a web page, the command line would look like this:

```
--imginfo '<IMG SRC="/img/%s" WIDTH="%lu"  \
           HEIGHT="%lu" ALT="Demo">'
```

[**-c**|**--color** *COLORTAG*#*rrggbb*[*aa*]]

Override the default colors for the standard elements of the graph. The *COLORTAG* is one of BACK background, CANVAS for the background of the actual graph, SHADEA for the left and top border, SHADEB for the right and bottom border, GRID, MGRID for the major grid, FONT for the color of the font, AXIS for the axis of the graph, FRAME for the line around the color spots and finally ARROW for the arrow head pointing up and forward. Each color is composed out of three hexadecimal numbers specifying its rgb color component (00 is off, FF is maximum) of red, green and blue. Optionally you may add another hexadecimal number specifying the transparency (FF is solid). You may set this option several times to alter multiple defaults.

A green arrow is made by: --color ARROW#00FF00

[**--zoom** *factor*]

Zoom the graphics by the given amount. The factor must be > 0

[**-n**|**--font** *FONTTAG*:*size*:[*font*]]

This lets you customize which font to use for the various text elements on the RRD graphs. DEFAULT sets the default value for all elements, TITLE for the title, AXIS for the axis labels, UNIT for the vertical unit label, LEGEND for the graph legend.

Use Times for the title:

```
--font TITLE:13:/usr/lib/fonts/times.ttf
```

If you do not give a font string you can modify just the sice of the default font: --font TITLE:13:.

If you specify the size 0 then you can modify just the font without touching the size. This is especially usefull for altering the default font without resetting the default fontsizes: --font DEFAULT:0:/usr/lib/fonts/times.ttf.

RRDtool comes with a preset default font. You can set the environment variable RRD_DEFAULT_FONT if you want to change this.

Truetype fonts are only supported for PNG output. See below.

[**-R**|**--font-render-mode** {*normal*,*light*,*mono*}]
: This lets you customize the strength of the font smoothing, or disable it entirely using *mono*. By default, *normal* font smoothing is used.

[**-B**|**--font-smoothing-threshold** *size*]
: This specifies the largest font size which will be rendered bitmapped, that is, without any font smoothing. By default, no text is rendered bitmapped.

[**-E**|**--slope-mode**]
: RRDtool graphs are composed of stair case curves by default. This is in line with the way RRDtool calculates its data. Some people favor a more 'organic' look for their graphs even though it is not all that true.

[**-a**|**--imgformat PNG**|**SVG**|**EPS**|**PDF**]
: Image format for the generated graph. For the vector formats you can choose among the standard Postscript fonts Courier-Bold, Courier-BoldOblique, Courier-Oblique, Courier, Helvetica-Bold, Helvetica-BoldOblique, Helvetica-Oblique, Helvetica, Symbol, Times-Bold, Times-BoldItalic, Times-Italic, Times-Roman, and ZapfDingbats.

[**-i**|**--interlaced**]
: If images are interlaced they become visible on browsers more quickly.

[**-g**|**--no-legend**]
: Suppress generation of the legend; only render the graph.

[**-F**|**--force-rules-legend**]
: Force the generation of HRULE and VRULE legends even if those HRULE or VRULE will not be drawn because out of graph

boundaries (mimics behaviour of pre 1.0.42 versions).

[**-T|--tabwidth** *value*]
: By default the tab-width is 40 pixels, use this option to change it.

[**-b|--base** *value*]
: If you are graphing memory (and NOT network traffic) this switch should be set to 1024 so that one Kb is 1024 byte. For traffic measurement, 1 kb/s is 1000 b/s.

Data and variables

DEF:*vname***=***rrdfile***:***ds-name***:***CF*[**:step=***step*][**:start=***time*][**:end=***time*]

CDEF:*vname***=***RPN expression*

VDEF:*vname***=***RPN expression*

You need at least one **DEF** statement to generate anything. The other statements are useful but optional.

If you have ever used a traditional HP calculator you already know RPN. The idea behind RPN is that you have a stack and push your data onto this stack. Whenever you execute an operation, it takes as many elements from the stack as needed. Pushing is done implicitly, so whenever you specify a number or a variable, it gets pushed onto the stack automatically.

At the end of the calculation there should be one and only one value left on the stack. This is the outcome of the function and this is what is put into the vname. For CDEF instructions, the stack is processed for each data point on the graph. VDEF instructions work on an entire data set in one run. Note, that currently VDEF instructions only support a limited list of functions.

Example: **VDEF:maximum=mydata,MAXIMUM**

This will set variable ``maximum'' which you now can use in the rest of your RRD script.

Example: **CDEF:mydatabits=mydata,8,***

This means: push variable mydata, push the number 8, execute the operator *. The operator needs two elements and uses those to return one value. This value is then stored in mydatabits. As you may have guessed, this instruction means nothing more than mydatabits = mydata * 8. The real power of RPN lies in the fact that it is always clear in which order to process the input. For expressions like a = b + 3 * 5 you need to multiply 3 with 5 first before you add b to get a. However, with parentheses you could change this order: a = (b + 3) * 5. In RPN, you would do a = b, 3, +, 5, * without the need for parentheses.

Boolean operators

> **LT, LE, GT, GE, EQ, NE**
>
> Pop two elements from the stack, compare them for the selected condition and return 1 for true or 0 for false. Comparing an unknown or an infinite value will always result in 0 (false).
>
> **UN, ISINF**
>
> Pop one element from the stack, compare this to unknown respectively to positive or negative infinity. Returns 1 for true or 0 for false.
>
> **IF**
>
> Pops three elements from the stack. If the element popped last is 0 (false), the value popped first is pushed back onto the stack, otherwise the value popped second is pushed back. This does, indeed, mean that any value other than 0 is considered to be true.
>
> Example: **A,B,C,IF**
>
> Should be read as if (A) then (B) else (C)

Comparing values

> **MIN, MAX**
>
> Pops two elements from the stack and returns the smaller or

larger, respectively. Note that infinite is larger than anything else. If one of the input numbers is unknown then the result of the operation will be unknown too.

LIMIT

Pops two elements from the stack and uses them to define a range. Then it pops another element and if it falls inside the range, it is pushed back. If not, an unknown is pushed.

The range defined includes the two boundaries (so: a number equal to one of the boundaries will be pushed back). If any of the three numbers involved is either unknown or infinite this function will always return an unknown

Example: **CDEF:a=alpha,0,100,LIMIT**

This will return unknown if alpha is lower than 0 or if it is higher than 100.

Arithmetics

+, -, *, /, %

Add, subtract, multiply, divide, modulo

SIN, COS, LOG, EXP, SQRT

Sine and cosine (input in radians), log and exp (natural logarithm), square root.

ATAN

Arctangent (output in radians).

ATAN2

Arctangent of y,x components (output in radians). This pops one element from the stack, the x (cosine) component, and then a second, which is the y (sine) component. It then pushes the arctangent of their ratio, resolving the ambiguity between quadrants.

Example: **CDEF:angle=Y,X,ATAN2,RAD2DEG**

Will convert X,Y components into an angle in degrees.

FLOOR, CEIL

Round down or up to the nearest integer.

DEG2RAD, RAD2DEG

Convert angle in degrees to radians, or radians to degrees.

Set Operations

SORT, REV

Pop one element from the stack. This is the count of items to be sorted (or reversed). The top count of the remaining elements are then sorted (or reversed) in place on the stack.

Example:
CDEF:x=v1,v2,v3,v4,v5,v6,6,SORT,POP,5,REV,POP,+,+,+,4,/

Will compute the average of the values v1 to v6 after removing the smallest and largest.

TREND

Create a ``sliding window'' average of another data series.

Usage: **CDEF:smoothed=x,1800,TREND**

This will create a half-hour (1800 second) sliding window average of x.

Special values

UNKN

Pushes an unknown value on the stack

INF, NEGINF

Pushes a positive or negative infinite value on the stack. When such a value is graphed, it appears at the top or bottom of the graph, no matter what the actual value on the y-axis is.

PREV

Pushes an unknown value if this is the first value of a data set or otherwise the result of this CDEF at the previous time step. This allows you to do calculations across the data. This function cannot be used in VDEF instructions.

PREV(vname)

Pushes an unknown value if this is the first value of a data set or otherwise the result of the vname variable at the previous time step. This allows you to do calculations across the data. This function cannot be used in VDEF instructions.

COUNT

Pushes the number 1 if this is the first value of the data set, the number 2 if it is the second, and so on. This special value allows you to make calculations based on the position of the value within the data set. This function cannot be used in VDEF instructions.

Time

Time inside RRDtool is measured in seconds since the epoch. The epoch is defined to be Thu Jan 1 00:00:00 UTC 1970.

NOW

Pushes the current time on the stack.

TIME

Pushes the time the currently processed value was taken at onto the stack.

LTIME

Takes the time as defined by TIME, applies the time zone offset valid at that time including daylight saving time if your OS supports it, and pushes the result on the stack. There is an elaborate example in the examples section below on how to use this.

Processing the stack directly

DUP, POP, EXC

Duplicate the top element, remove the top element, exchange the two top elements.

Variables

These operators work only on VDEF statements. Note that currently ONLY these work for VDEF.

MAXIMUM, MINIMUM, AVERAGE

Return the corresponding value, MAXIMUM and MINIMUM also return the first occurrence of that value in the time component.

Example: V**DEF:avg=mydata,AVERAGE**

LAST, FIRST

Return the last/first value including its time. The time for FIRST is actually the start of the corresponding interval, whereas LAST returns the end of the corresponding interval.

Example: **VDEF:first=mydata,FIRST**

TOTAL

Returns the rate from each defined time slot multiplied with the step size. This can, for instance, return total bytes transfered when you have logged bytes per second. The time component returns the number of seconds.

Example: **VDEF:total=mydata,TOTAL**

PERCENT

This should follow a DEF or CDEF vname. The vname is popped, another number is popped which is a certain percentage (0..100). The data set is then sorted and the value returned is chosen such that percentage percent of the values is lower or equal than the result. Unknown values are considered lower than any finite number for this purpose so if this operator returns an unknown you have quite a lot of them in your data. Infinite numbers are lesser, or more, than the finite numbers and are always more than the Unknown numbers. (NaN < -INF < finite values < INF)

Example: **VDEF:perc95=mydata,95,PERCENT**

LSLSLOPE, LSLINT, LSLCORREL

Return the parameters for a Least Squares Line (y = mx +b) which approximate the provided dataset. LSLSLOPE is the slope (m) of the line related to the COUNT position of the data. LSLINT is the y-intercept (b), which happens also to be the first data point on the graph. LSLCORREL is the Correlation Coefficient (also know as Pearson's Product Moment Correlation Coefficient). It will range from 0 to +/-1 and represents the quality of fit for the approximation.

Example: **VDEF:slope=mydata,LSLSLOPE**

Graph and print elements

You need at least one graph element to generate an image and/or at least one print statement to generate a report.

rrddump

Usage

rrdtool dump *filename.rrd* > *filename.xml*
or
rrdtool dump *filename.rrd filename.xml*

Description

The dump function writes the contents of an RRD in human readable (?) XML format to a file or to stdout. This format can be read by rrdrestore. Together they allow you to transfer your files from one computer architecture to another as well to manipulate the contents of an RRD file in a somewhat more convenient manner.

Options

filename.rrd

The name of the RRD you want to dump.

filename.xml

The (optional) filename that you want to write the XML output to. If not specified, the XML will be printed to stdout.

Examples

To transfer an RRD between architectures, follow these steps:

- On the same system where the RRD was created, use **rrdtool dump** to export the data to XML format.
- Transfer the XML dump to the target system.
- Run **rrdtool restore** to create a new RRD from the XML dump. See **rrdrestore** for details.

rrdrestore

Usage

rrdtool restore *filename.xml filename.rrd* [**--range-check|-r**]

Description

The **restore** function reads the XML representation of an RRD and converts it to the native **RRD** format.

Options

filename.xml
The name of the XML file you want to restore.

filename.rrd
The name of the RRD to restore.

--range-check|-r
Make sure the values in the RRAs do not exceed the limits defined for the various data sources.

--force-overwrite|-f
Allows RRDtool to overwrite the destination RRD.

rrdfetch

Usage

rrdtool fetch *filename CF* [**--resolution|-r** *resolution*] [**--start|-s** *start*] [**--end|-e** *end*]

Description

The fetch function is normally used internally by the graph function to get data from RRDs. fetch will analyze the RRD and try to retrieve the data in the resolution requested. The data fetched is printed to stdout. **UNKNOWN** data is often represented by the string ``NaN'' depending on your OS's printf function.

Options

filename
The name of the RRD you want to fetch the data from.

CF

The consolidation function that is applied to the data you want to fetch (AVERAGE,MIN,MAX,LAST)

--resolution|-r *resolution* (default is the highest resolution)

The interval you want the values to have (seconds per value). rrdfetch will try to match your request, but it will return data even if no absolute match is possible. NB. See note below.

--start|-s *start* (default end-1day)

Start of the time series. A time in seconds since epoch (1970-01-01) is required. Negative numbers are relative to the current time. By default, one day worth of data will be fetched.

--end|-e *end* (default now)

the end of the time series in seconds since epoch.

RESOLUTION INTERVAL

In order to get RRDtool to fetch anything other than the finest resolution RRA both the start and end time must be specified on boundaries that are multiples of the desired resolution. Consider the following example:

```
# rrdtool create subdata.rrd -s 10 DS:ds0:GAUGE:300:0:U \
            RRA:AVERAGE:0.5:30:3600 \
            RRA:AVERAGE:0.5:90:1200 \
            RRA:AVERAGE:0.5:360:1200 \
            RRA:MAX:0.5:360:1200 \
            RRA:AVERAGE:0.5:8640:600 \
            RRA:MAX:0.5:8640:600
```

This RRD collects data every 10 seconds and stores its averages over 5 minutes, 15 minutes, 1 hour, and 1 day, as well as the maxima for 1 hour and 1 day.

Consider now that you want to fetch the 15 minute average data for the last hour. You might try

```
# rrdtool fetch subdata.rrd AVERAGE -r 900 -s -1h
```

However, this will almost always result in a time series that is NOT in the 15 minute RRA. Therefore, the highest resolution RRA, i.e. 5 minute averages, will be chosen which in this case is not what you want.

Hence, make sure that

1. both start and end time are a multiple of 900
2. both start and end time are within the desired RRA

So, if time now is called ``t'', do

*end time == int(t/900)*900,*
start time == end time - 1hour,
resolution == 900.

Using the bash shell, this could look be:

```
# TIME=$(date +%s)
# RRDRES=900
# rrdtool fetch subdata.rrd AVERAGE -r $RRDRES \
          -e $(echo $(($TIME/$RRDRES*$RRDRES))) -s e-1h
```

Or in Perl:

```
# perl -e '$ctime = time; $rrdres = 900; \
     system "rrdtool fetch subdata.rrd AVERAGE \
          -r $rrdres -e @{[int($ctime/$rrdres)*$rrdres]} -s e-
1h"'
```

AT-STYLE TIME SPECIFICATION

Apart from the traditional *Seconds since epoch*, RRDtool does also understand at-style time specification. The specification is called ``at-style'' after the Unix command `at(1)` that has moderately complex ways to specify time to run your job at a certain date and time. The at-style specification consists of two parts: the TIME REFERENCE specification and the TIME OFFSET specification.

TIME REFERENCE SPECIFICATION

The time reference specification is used, well, to establish a reference moment in time (to which the time offset is then applied to). When present, it should come first, when omitted, it defaults to now. On its own part, time reference consists of a *time-of-day* reference (which should come first, if present) and a *day* reference.

The *time-of-day* can be specified as HH:MM, HH.MM, or just HH. You can suffix it with am or pm or use 24-hours clock. Some special times of day are understood as well, including midnight (00:00), noon (12:00) and British teatime (16:00).

The *day* can be specified as *month-name day-of-the-month* and optional a 2- or 4-digit *year* number (e.g. March 8 1999). Alternatively, you can use *day-of-week-name* (e.g. Monday), or one of the words: yesterday, today, tomorrow. You can also specify the *day* as a full date in several numerical formats, including MM/DD/[YY]YY, DD.MM.[YY]YY, or YYYYMMDD.

NOTE1: this is different from the original `at(1)` behavior, where a single-number date is interpreted as MMDD[YY]YY.

NOTE2: if you specify the *day* in this way, the *time-of-day* is REQUIRED as well.

Finally, you can use the words now, start, or end as your time reference. Now refers to the current moment (and is also the default time reference). Start (end) can be used to specify a time relative to the start (end) time for those tools that use these categories.

Month and day of the week names can be used in their naturally abbreviated form (e.g., Dec for December, Sun for Sunday, etc.). The words now, start, end can be abbreviated as n, s, e.

TIME OFFSET SPECIFICATION

The time offset specification is used to add/subtract certain time intervals to/from the time reference moment. It consists of a *sign* (+ or -) and an *amount*. The following time units can be used to specify the *amount*: years, months, weeks, days, hours, minutes, or seconds. These units can be used in singular or plural form, and abbreviated naturally or to a single letter (e.g. +3days, -1wk, -3y). Several time units can be combined (e.g., -5mon1w2d) or concatenated (e.g., -5h45min = -5h-45min = -6h+15min = -7h+1h30m-15min, etc.)

NOTE3: If you specify time offset in days, weeks, months, or years, you will end with the time offset that may vary depending on your time reference, because all those time units have no single well defined time interval value (1 year contains either 365 or 366 days, 1 month is 28 to 31 days long, and even 1 day may be not equal to 24 hours twice a year, when DST-related clock adjustments take place). To cope with this, when you use days, weeks, months, or years as your time offset units your time reference date is adjusted accordingly without too much further effort to ensure anything about it (in the hope that `mktime(3)` will take care of this later). This may lead to some surprising (or even invalid!) results, e.g. 'May 31 -1month' =

'Apr 31' (meaningless) = 'May 1' (after `mktime(3)` normalization); in the EET timezone '3:30am Mar 29 1999 -1 day' yields '3:30am Mar 28 1999' (Sunday) which is an invalid time/date combination (because of 3am -> 4am DST forward clock adjustment, see the below example).

In contrast, hours, minutes, and seconds are well defined time intervals, and these are guaranteed to always produce time offsets exactly as specified (e.g. for EET timezone, '8:00 Mar 27 1999 +2 days' = '8:00 Mar 29 1999', but since there is 1-hour DST forward clock adjustment that occurs around 3:00 Mar 28 1999, the actual time interval between 8:00 Mar 27 1999 and 8:00 Mar 29 1999 equals 47 hours; on the other hand, '8:00 Mar 27 1999 +48 hours' = '9:00 Mar 29 1999', as expected)

NOTE4: The single-letter abbreviation for both months and minutes is m. To disambiguate them, the parser tries to read your mind :) by applying the following two heuristics:

1. If m is used in context of (i.e. right after the) years, months, weeks, or days it is assumed to mean months, while in the context of hours, minutes, and seconds it means minutes. (e.g., in -1y6m or +3w1m m is interpreted as months, while in -3h20m or +5s2m m the parser decides for minutes).

2. Out of context (i.e. right after the + or - sign) the meaning of m is guessed from the number it directly follows. Currently, if the number's absolute value is below 25 it is assumed that m means months, otherwise it is treated as minutes. (e.g., -25m == -25 minutes, while +24m == +24 months)

Final NOTES: Time specification is case-insensitive. Whitespace can be inserted freely or omitted altogether. There are, however, cases when whitespace is required (e.g., 'midnight Thu'). In this case you should either quote the whole phrase to prevent it from being taken apart by your shell or use '_' (underscore) or ',' (comma) which also count as whitespace (e.g., midnight_Thu or midnight,Thu).

TIME SPECIFICATION EXAMPLES

Oct 12

October 12 this year

-1month **or** ***-1m***

current time of day, only a month before (may yield surprises, see

NOTE3 above).

noon yesterday -3hours
: yesterday morning; can also be specified as *9am-1day*.

23:59 31.12.1999
: 1 minute to the year 2000.

12/31/99 11:59pm
: 1 minute to the year 2000 for imperialists.

12am 01/01/01
: start of the new millennium

end-3weeks* or *e-3w
: 3 weeks before end time (may be used as start time specification).

start+6hours* or *s+6h
: 6 hours after start time (may be used as end time specification).

931225537
: 18:45 July 5th, 1999 (yes, seconds since 1970 are valid as well).

19970703 12:45
: 12:45 July 3th, 1997 (my favorite, and its even got an ISO number (8601)).

rrdtune

Usage

rrdtool tune *filename* [**--heartbeat|-h** *ds-name*:*heartbeat*] [**--minimum|-i** *ds-name*:*min*] [**--maximum|-a** *ds-name*:*max*] [**--data-source-type|-d** *ds-name*:*DST*] [**--data-source-rename|-r** *old-name*:*new-name*] [**--deltapos** *scale-value*] [**--deltaneg** *scale-value*] [**--failure-threshold** *failure-threshold*] [**--window-length** *window-length*] [**--alpha** *adaption-parameter*] [**--beta** *adaption-parameter*] [**--gamma** *adaption-parameter*] [**--gamma-deviation** *adaption-parameter*] [**--aberrant-reset** *ds-name*]

Description

The tune option allows you to alter some of the basic configuration values stored in the header area of a Round Robin Database (RRD).

One application of the tune function is to relax the validation rules on an RRD. This allows to fill a new RRD with data available in larger intervals than what you would normally want to permit. Be very careful with tune operations for COMPUTE data sources. Setting the *min*, *max*, and *heartbeat* for a COMPUTE data source without changing the data source type to a non-COMPUTE DST WILL corrupt the data source header in the RRD.

A second application of the tune function is to set or alter parameters used by the specialized function RRAs for aberrant behavior detection.

Options

filename

The name of the RRD you want to tune.

--heartbeat|-h *ds-name*:*heartbeat*

Modify the *heartbeat* of a data source. By setting this to a high value the RRD will accept things like one value per day.

--minimum|-i *ds-name*:*min*

Alter the minimum value acceptable as input from the data source. Setting *min* to 'U' will disable this limit.

--maximum|-a *ds-name*:*max*

Alter the maximum value acceptable as input from the data source. Setting *max* to 'U' will disable this limit.

--data-source-type|-d *ds-name*:*DST*

Alter the type DST of a data source.

--data-source-rename|-r *old-name*:*new-name*

*R*ename a data source.

--deltapos *scale-value*

Alter the deviation scaling factor for the upper bound of the confidence band used internally to calculate violations for the FAILURES RRA. The default value is 2. Note that this parameter is not related to graphing confidence bounds which must be specified as a CDEF argument to generate a graph with confidence bounds. The graph scale factor need not to agree with the value used internally by the FAILURES RRA.

--deltaneg *scale-value*

Alter the deviation scaling factor for the lower bound of the confidence band used internally to calculate violations for the FAILURES RRA. The default value is 2. As with --deltapos, this argument is unrelated to the scale factor chosen when graphing confidence bounds.

--failure-threshold *failure-threshold*
Alter the number of confidence bound violations that constitute a failure for purposes of the FAILURES RRA. This must be an integer less than or equal to the window length of the FAILURES RRA. This restriction is not verified by the tune option, so one can reset failure-threshold and window-length simultaneously. Setting this option will reset the count of violations to 0.

--window-length *window-length*
Alter the number of time points in the temporal window for determining failures. This must be an integer greater than or equal to the window length of the FAILURES RRA and less than or equal to 28. Setting this option will reset the count of violations to 0.

--alpha *adaption-parameter*
Alter the intercept adaptation parameter for the Holt-Winters forecasting algorithm. This parameter must be between 0 and 1.

--beta *adaption-parameter*
Alter the slope adaptation parameter for the Holt-Winters forecasting algorithm. This parameter must be between 0 and 1.

--gamma *adaption-parameter*
Alter the seasonal coefficient adaptation parameter for the SEASONAL RRA. This parameter must be between 0 and 1.

--gamma-deviation *adaption-parameter*
Alter the seasonal deviation adaptation parameter for the DEVSEASONAL RRA. This parameter must be between 0 and 1.

--aberrant-reset *ds-name*
This option causes the aberrant behavior detection algorithm to reset for the specified data source; that is, forget all it is has learnt so far. Specifically, for the HWPREDICT RRA, it sets the intercept and slope coefficients to unknown. For the SEASONAL RRA, it sets all seasonal coefficients to unknown. For the DEVSEASONAL RRA, it sets all seasonal deviation coefficients to unknown. For the FAILURES RRA, it erases the violation history. Note that reset does not erase past

predictions (the values of the HWPREDICT RRA), predicted deviations (the values of the DEVPREDICT RRA), or failure history (the values of the FAILURES RRA). This option will function even if not all the listed RRAs are present.

Due to the implementation of this option, there is an indirect impact on other data sources in the RRD. A smoothing algorithm is applied to SEASONAL and DEVSEASONAL values on a periodic basis. During bootstrap initialization this smoothing is deferred. For efficiency, the implementation of smoothing is not data source specific. This means that utilizing reset for one data source will delay running the smoothing algorithm for all data sources in the file. This is unlikely to have serious consequences, unless the data being collected for the non-reset data sources is unusually volatile during the reinitialization period of the reset data source.

Use of this tuning option is advised when the behavior of the data source time series changes in a drastic and permanent manner.

Example 1

# rrdtool tune data.rrd -h in:100000 -h out:100000 -h through:100000

Set the minimum required heartbeat for data sources 'in', 'out' and 'through' to 10'000 seconds which is a little over one day in data.rrd. This would allow to feed old data from MRTG-2.0 right into RRDtool without generating *UNKNOWN* entries.

Example 2

# rrdtool tune monitor.rrd --window-length 5 --failure-threshold 3

If the FAILURES RRA is implicitly created, the default window-length is 9 and the default failure-threshold is 7. This command now defines a failure as 3 or more violations in a temporal window of 5 time points.

rrdlast

Usage

rrdtool last *filenam*

Description

The last function returns the UNIX timestamp of the most recent update of the RRD.

Options

filename

The name of the RRD that contains the data.

rrdresize

Usage

rrdtool resize *filename rra-num* **GROW/SHRINK** *rows*

Description

The **resize** function is used to modify the number of rows in an **RRA**.

Options

filename

The name of the **RRD** you want to alter.

rra-num

The **RRA** you want to alter. You can find the number using **rrdtool info**.

GROW

Used if you want to add extra rows to an RRA. The extra rows will be inserted as the rows that are oldest.

SHRINK

Used if you want to remove rows from an RRA. The rows that will be removed are the oldest rows.

rows

The number of rows you want to add or remove.

Notes

The new .rrd file, with the modified RRAs, is written to the file **resize.rrd** in the current directory. **The original .rrd file is not modified**.

It is possible to abuse this tool and get strange results by first removing some rows and then reinserting the same amount (effectively clearing them to be Unknown). You may thus end up with unknown data in one RRA while at the same timestamp this data is available in another RRA.

rrdcgi

Usage

#!/path/to/**rrdcgi** [**--filter**]

Description

rrdcgi is a sort of very limited script interpreter. Its purpose is to run as a cgi-program and parse a web page template containing special <RRD:: tags. rrdcgi will interpret and act according to these tags. In the end it will printout a web page including the necessary CGI headers.

rrdcgi parses the contents of the template in 3 steps. In each step it looks only for a subset of tags. This allows nesting of tags.

The argument parser uses the same semantics as you are used from your C-shell.
--filter

Assume that rrdcgi is run as a filter and not as a cgi.

Keywords

RRD::CV *name*
Inserts the CGI variable of the given name.

RRD::CV::QUOTE *name*
Inserts the CGI variable of the given name but quotes it, ready for use as an argument in another RRD:: tag. So even when there are spaces in the value of the CGI variable it will still be considered to be one argument.

RRD::CV::PATH *name*
Inserts the CGI variable of the given name, quotes it and makes sure

it starts neither with a '/' nor contains '..'. This is to make sure that no problematic pathnames can be introduced through the CGI interface.

RRD::GETENV *variable*
: Get the value of an environment variable.

<RRD::GETENV REMOTE_USER>
: might give you the name of the remote user given you are using some sort of access control on the directory.

RRD::GOODFOR *seconds*
: Specify the number of seconds this page should remain valid. This will prompt the rrdcgi to output a Last-Modified, an Expire and if the number of seconds is *negative* a Refresh header.

RRD::INCLUDE *filename*
: Include the contents of the specified file into the page returned from the cgi.

RRD::SETENV *variable value*
: If you want to present your graphs in another time zone than your own, you could use

  <RRD::SETENV TZ UTC>

  to make sure everything is presented in Universal Time. Note that the values permitted to TZ depend on your OS.

RRD::SETVAR *variable value*
: Analog to SETENV but for local variables.

RRD::GETVAR *variable*
: Analog to GETENV but for local variables.

RRD::TIME::LAST *rrd-file strftime-format*
: This gets replaced by the last modification time of the selected RRD. The time is *strftime*-formatted with the string specified in the second

argument.

RRD::TIME::NOW *strftime-format*

This gets replaced by the current time of day. The time is *strftime*-formatted with the string specified in the argument.
Note that if you return : (colons) from your strftime format you may have to escape them using \ if the time is to be used as an argument to a GRAPH command.

RRD::TIME::STRFTIME *START|END start-spec end-spec strftime-format*

This gets replaced by a strftime-formatted time using the format *strftime-format* on either *start-spec* or *end-spec* depending on whether *START* or *END* is specified. Both *start-spec* and *end-spec* must be supplied as either could be relative to the other. This is intended to allow pretty titles on graphs with times that are easier for non RRDtool folks to figure out than ``-2weeks''.
Note that again, if you return : (colon) from your strftime format, you may have to escape them using \ if the time is to be used as an argument to a GRAPH command.

RRD::GRAPH *rrdgraph arguments*

This tag creates the RRD graph defined by its argument and then is replaced by an appropriate <IMG ... > tag referring to the graph. The --lazy option in RRD graph can be used to make sure that graphs are only regenerated when they are out of date. The arguments to the RRD::GRAPH tag work as described in the rrdgraph manual page.

Use the --lazy option in your RRD::GRAPH tags, to reduce the load on your server. This option makes sure that graphs are only regenerated when the old ones are out of date.

If you do not specify your own --imginfo format, the following will be used:

<IMG SRC="%s" WIDTH="%lu" HEIGHT="%lu">

Note that %s stands for the filename part of the graph generated, all directories given in the PNG file argument will get dropped.

RRD::PRINT *number*

If the preceding RRD::GRAPH tag contained and PRINT arguments, then you can access their output with this tag. The *number* argument refers to the number of the PRINT argument. This first PRINT has *number* 0.

Example 1

The example below creates a web pages with a single RRD graph.

```
#!/usr/local/bin/rrdcgi
<HTML>
<HEAD><TITLE>RRDCGI Demo</TITLE></HEAD>
<BODY>
<H1>RRDCGI Example Page</H1>
<P>
<RRD::GRAPH demo.png --lazy --title="Temperatures"
        DEF:cel=demo.rrd:exhaust:AVERAGE
        LINE2:cel#00a000:"D. Celsius">

</P>
</BODY>
</HTML>
```

Example 2

This script is slightly more elaborate, it allows you to run it from a form which sets RRD_NAME. RRD_NAME is then used to select which RRD you want to use as source for your graph.

```
#!/usr/local/bin/rrdcgi
<HTML>
<HEAD><TITLE>RRDCGI Demo</TITLE></HEAD>
<BODY>
<H1>RRDCGI Example Page for <RRD::CV RRD_NAME></H1>
<H2>Selection</H2>
<FORM><INPUT NAME=RRD_NAME TYPE=RADIO
VALUE=roomA> Room A,
     <INPUT NAME=RRD_NAME TYPE=RADIO VALUE=roomB>
Room B.
     <INPUT TYPE=SUBMIT></FORM>
<H2>Graph</H2>
<P>
<RRD::GRAPH <RRD::CV::PATH RRD_NAME>.png --lazy
```

```
        --title "Temperatures for "<RRD::CV::QUOTE RRD_NAME>
        DEF:cel=<RRD::CV::PATH RRD_NAME>.rrd:exhaust:AVERAGE
        LINE2:cel#00a000:"D. Celsius">

</P>
</BODY>
</HTML>
```

Example 3

This example shows how to handle the case where the RRD, graphs and cgi-bins are seperate directories

```
#!/.../bin/rrdcgi
<HTML>
<HEAD><TITLE>RRDCGI Demo</TITLE></HEAD>
<BODY>
<H1>RRDCGI test Page</H1>
<RRD::GRAPH
 /.../web/pngs/testhvt.png
 --imginfo '<IMG SRC=/.../pngs/%s WIDTH=%lu HEIGHT=%lu >'
 --lazy --start -1d --end now
 DEF:http_src=/.../rrds/test.rrd:http_src:AVERAGE
 AREA:http_src#00ff00:http_src
>
</BODY>
</HTML>
```

Note 1: Replace /.../ with the relevant directories
Note 2: The SRC=/.../pngs should be paths from the view of the webserver/browser

Advanced configuration example

This example demonstarates a firewall configuration which is used for an organization connected to a Cisco router, which in turn is used as the gateway to the internet. The server is configured to accept SMTP traffic from outside including incoming POP3 connections. This firewall will block all smtp traffic sourcing from inside going outside, this blocks most mass mailing worms.

```
<firewall>

#
```

```
#    Global configuration and access classes
#
<global>
        # Modules we need to load
        <modules>
                <load name="ip_queue"/>
                <load name="ip_conntrack_ftp"/>
                <load name="ip_nat_ftp"/>
        </modules>

        #
        # BEGIN - STANDARD CLASSES
        #
        <class name="local_iface">
                <address src-iface="lo"/>
        </class>

        <class name="valid_connections">
                <address cmd-line="-m state --state
                                ESTABLISHED,RELATED"/>
        </class>

        <class name="syn_packets">
                <address proto="tcp" cmd-line="--syn -m state --state NEW"/>
        </class>

        <class name="udp_packets">
                <address proto="udp"/>
        </class>

        <class name="icmp_packets">
                <address proto="icmp"/>
        </class>

        <class name="rsvp_packets">
                <address proto="2"/>
        </class>

        <class name="invalid_tcp_packets">
                <address proto="tcp" cmd-line="--tcp-flags ALL
                                FIN,URG,PSH"/>
                <address proto="tcp" cmd-line="--tcp-flags ALL ALL"/>
                <address proto="tcp" cmd-line="--tcp-flags ALL
                                SYN,RST,ACK,FIN,URG"/>
                <address proto="tcp" cmd-line="--tcp-flags ALL NONE"/>
        <address proto="tcp" cmd-line="--tcp-flags SYN,RST
                        SYN,RST"/>
        <address proto="tcp" cmd-line="--tcp-flags SYN,FIN
                        SYN,FIN"/>
</class>

<class name="valid_icmp_packets">
        <address proto="icmp" cmd-line="--icmp-type 0"/>
        <address proto="icmp" cmd-line="--icmp-type 3"/>
        <address proto="icmp" cmd-line="--icmp-type 8"/>
        <address proto="icmp" cmd-line="--icmp-type 11"/>
</class>

<class name="traceroute_packets">
        <address proto="udp" dst-port="33434:33465"/>
</class>

<class name="service_ftp">
        <address proto="tcp" dst-port="21"/>
</class>

<class name="service_ssh">
        <address proto="tcp" dst-port="22"/>
</class>

<class name="service_smtp">
        <address proto="tcp" dst-port="25"/>
</class>

<class name="service_dns">
        <address proto="tcp" dst-port="53"/>
        <address proto="udp" dst-port="53"/>
</class>

<class name="service_http">
        <address proto="tcp" dst-port="80"/>
</class>

<class name="service_https">
        <address proto="tcp" dst-port="443"/>
</class>

<class name="service_pop3">
        <address proto="tcp" dst-port="110"/>
</class>
```

```
<class name="service_tinc">
        <address proto="udp" dst-port="655"/>
        <address proto="tcp" dst-port="655"/>
</class>

<class name="service_ident">
        <address proto="tcp" dst-port="113"/>
</class>

<class name="service_imap">
        <address proto="tcp" dst-port="143"/>
</class>

<class name="service_pserver">
        <address proto="tcp" dst-port="2401"/>
</class>

<class name="service_httpproxy">
        <address proto="tcp" dst-port="3128"/>
        <address proto="tcp" dst-port="8080"/>
</class>

<class name="service_postgresql">
        <address proto="tcp" dst-port="5432"/>
</class>

<class name="service_time">
        <address proto="udp" dst-port="123" src-port="123"/>
</class>

<class name="service_rip">
        <address proto="udp" dst-port="520" src-port="520"/>
</class>

<class name="service_datametrics">
        <address proto="udp" dst-port="1645"/>
        <address proto="udp" dst-port="1646"/>
</class>

<class name="service_radius">
        <address proto="udp" dst-port="1812"/>
        <address proto="udp" dst-port="1813"/>
</class>
```

```
        <class name="service_dhcp">
                <address proto="udp" dst-port="67:68"/>
        </class>

        <class name="30_per_min">
                <address cmd-line="-m limit --limit 30/min --limit-burst 10"/>
        </class>

        <class name="blank">
                <address />
        </class>

        #
        # END - STANDARD CLASSES
        #

        <class name="valid_internal_traffic">
                <address src-iface="eth1" src="192.168.101.0/26"
                                dst-iface="eth0"/>
        </class>

        <class name="nat_internal_traffic">
                <address src="192.168.101.0/26" dst="! 192.168.101.0/24"/>
        </class>

        <class name="internal_traffic">
                <address src-iface="eth1" dst-iface="eth0"/>
        </class>

        <class name="proxy_redirect">
                <address src="192.168.101.0/24" proto="tcp" dst="!
                                192.168.101.0/24" dst-port="80"/>
        </class>

        <class name="internal_local">
                <address src="192.168.101.0/24" />
        </class>

        # eth0 loop is normally used when doing strange NAT stuff
        <class name="eth0_loop">
                <address src-iface="eth0" dst-iface="eth0"/>
        </class>

</global>
```

```
#
# Access control lists
#
<acl>
        <table name="filter">

                #
                # CUSTOM RULES
                #

                <chain name="accept_input_all">
                </chain>

                <chain name="accept_input_tcp">
                        <rule target="accept_traffic">
                                service_smtp;
                                service_pop3;
                        </rule>
                </chain>

                <chain name="accept_input_udp">
                </chain>

                <chain name="accept_input_icmp">
                </chain>

                <chain name="invalid_forwarding">
                        <rule target="REJECT">
                                service_smtp;
                        </rule>
                </chain>

                <chain name="accept_forward_all">
                        <rule target="invalid_forwarding">
                                internal_traffic;
                        </rule>
                </chain>

                <chain name="accept_forward_tcp">
                        <rule target="accept_traffic">
                                valid_internal_traffic;
                        </rule>
                </chain>

                <chain name="accept_forward_udp">
                        <rule target="accept_traffic">
                                valid_internal_traffic;
                        </rule>
                </chain>

                <chain name="accept_forward_icmp">
                        <rule target="accept_traffic">
                                valid_internal_traffic;
                        </rule>
                </chain>

                <chain name="accept_output_all">
                        <rule target="accept_traffic">
                                blank;
                        </rule>
                </chain>

                <chain name="accept_output_tcp">
                </chain>

                <chain name="accept_output_udp">
                </chain>

                <chain name="accept_output_icmp">
                </chain>


                #
                # SYSTEM INPUT RULES - CUSTOMIZE ABOVE
                #
                <chain name="accept_input_all">
                        <rule target="accept_traffic">
                                local_iface;
                        </rule>
                </chain>

                <chain name="accept_input_tcp">
                        <rule target="accept_traffic">
                                service_ssh;
                        </rule>
                </chain>

                <chain name="accept_input_udp">
                </chain>
```

```
<chain name="accept_input_icmp">
        <rule target="accept_traffic">
                valid_icmp_packets;
                traceroute_packets;
        </rule>
</chain>

#
# SYSTEM FORWARD RULES - CUSTOMIZE ABOVE
#
<chain name="accept_forward_all">
</chain>

<chain name="accept_forward_tcp">
</chain>

<chain name="accept_forward_udp">
</chain>

<chain name="accept_forward_icmp">
</chain>

#
# SYSTEM LOGGING RULES
#
<chain name="log_input">
        <rule target='LOG --log-prefix "FW:filter:INPUT "'>
                30_per_min;
        </rule>
</chain>

<chain name="log_forward">
        <rule target='LOG --log-prefix "FW:filter:FORWARD "'>
                30_per_min;
        </rule>
</chain>

<chain name="log_output">
        <rule target='LOG --log-prefix "FW:filter:OUTPUT "'>
                30_per_min;
        </rule>
</chain>

<chain name="log_drop_packets">
        <rule target='LOG --log-prefix "FW:filter:check_pkts"'>
```

```
                30_per_min;
        </rule>
        <rule target="DROP">
                blank;
        </rule>
</chain>

#
# MAIN SYSTEM RULES
#

# Remove bwmd rule if you not using it
<chain name="accept_traffic">
        <rule target="ACCEPT">
                blank;
        </rule>
</chain>

<chain name="accept_state">
        <rule target="accept_traffic">
                valid_connections;
        </rule>
</chain>

<chain name="check_packets">
        <rule target="log_drop_packets">
                invalid_tcp_packets;
        </rule>
</chain>


#
# MAIN SYSTEM CHAINS
#
<chain name="INPUT" default="DROP">
        <rule target="check_packets">
                blank;
        </rule>
        <rule target="accept_state">
                blank;
        </rule>
        <rule target="accept_input_all">
                blank;
        </rule>
        <rule target="accept_input_tcp">
```

```
                syn_packets;
        </rule>
        <rule target="accept_input_udp">
                udp_packets;
        </rule>
        <rule target="accept_input_icmp">
                icmp_packets;
        </rule>
        <rule target="log_input">
                blank;
        </rule>
</chain>

<chain name="FORWARD" default="DROP">
        <rule target="check_packets">
                blank;
        </rule>
        <rule target="accept_state">
                blank;
        </rule>
        <rule target="accept_forward_all">
                blank;
        </rule>
        <rule target="accept_forward_tcp">
                syn_packets;
        </rule>
        <rule target="accept_forward_udp">
                udp_packets;
        </rule>
        <rule target="accept_forward_icmp">
                icmp_packets;
        </rule>
        <rule target="log_forward">
                blank;
        </rule>
</chain>

<chain name="OUTPUT" default="DROP">
        <rule target="check_packets">
                blank;
        </rule>
        <rule target="accept_state">
                blank;
        </rule>
        <rule target="accept_output_all">
```

```
                                        blank;
                                </rule>
                                <rule target="accept_output_tcp">
                                        syn_packets;
                                </rule>
                                <rule target="accept_output_udp">
                                        udp_packets;
                                </rule>
                                <rule target="accept_output_icmp">
                                        icmp_packets;
                                </rule>
                                <rule target="log_output">
                                        blank;
                                </rule>
                        </chain>
                </table>
        </acl>

        <nat>
                <snat>
                        <rule to-src="your.external.ip.here">
                                nat_internal_traffic;
                        </rule>
                </snat>
        </nat>

</firewall>
```

Credits

Parts of this document were composed from the sources listed below…

Linux Advanced Routing & Traffic Control – http://www.lartc.org

Netfilter Howto's – http://www.netfilter.org/

iptables(8)

ebtables website – http://ebtables.sourceforge.net

ebtables(8)

Ethernet bridge – http://bridge.sourceforge.net

Application Layer Packet Classifier for Linux – http://l7-filter.sourceforge.net

BWM Tools – http://bwm-tools.pr.linuxrulz.org

Squid – http://www.squid-cache.org

Dante – http://www.inet.no/dante/

ISC Bind (named) – http://www.isc.org/sw/bind/

DJBDNS – http://cr.yp.to/djbdns.html

"BWM Tools, How to use it?" (by Kobe Lenjou) – http://www.murder4al.be

RRDTool - http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/

brctl(8)