# Linux Advanced Routing & Traffic Control

*Bert Hubert*
PowerDNS.COM, bv
*bert@powerdns.com, http://ds9a.nl/*

## Abstract

Linux contains a wildly powerful system for shaping traffic and distributing it according to elaborate rules. This paper serves a dual purpose: to explain how to do this as a user and how to write a scheduler in the kernel.

## 1 Introduction

In the absence of infinite bandwidth there will always be a need to hand out capacity according to rules. Traditionally this has been a main reason to add non-IP technology to a network, like ATM or frame relay. Since IP is steadily taking over the world, Linux is well placed to play a role in enabling IP to take over traffic controlling functions from other technologies.

Traditionally, traffic control has been very difficult to configure and Linux is no different in this respect. In addition to this most of the important bits have not been documented.

About two years ago, the '2.4 Advanced Routing' HOWTO was started, well before the advent of Linux 2.4 in order to rectify this situation. Not hampered by any understanding a lot was written which was already helpful in configuring traffic control under linux.

By now a set of manpages has been written and the HOWTO properly explains most things.

## 2 Theory

As explained, traffic control is not an easy subject. Its difficulty can be compared to that of a postal service deciding to offer two kinds of service—'fast' and 'slow' where there previously was only 'reasonably fast.' Some kind of system must be devised to prioritize some kinds of traffic, but actively slow down others.

Now, the naive view of traffic shaping ("Hey, just slow the packets down!") corresponds to ordering all mail vehicles to lower speed—which clearly does not solve our problem.

We must be far smarter than that and not resort to wasteful solutions like slowing everything down.

The first thing to realise is that we can only realistically do complicated things with *outgoing* traffic. We have zero control over the rate at which people send us data. Again, this is like receiving (physical) mail. People send it to us and we can only decide not to read it—there is no way to make it come in any slower.

Furthermore even for outgoing traffic we can only treat packets that are *in* a computer we maintain. A prime example which many people encounter is trying to shape traffic going out to a cable modem which is connected via a 10 megabit ethernet. As this 10 megabit connection is lots faster than the cable modem, the Linux machine does not own the queue and hence powerless to prioritize traffic, unless fur-

ther work is undertaken.

So—the theory is like this. Make sure that if there is a need to prioritize traffic, there is a queue which can be processed. Because there is normally only an outgoing queue configure your traffic control such that the data that needs to be prioritized is outgoing.

## 3 Verbiage

As with any complicated subject it is important to get the terminology right. I'm much indebted to Jamal who keeps pointing this out to me—his persistence is formidable and my stubbornness only just matches it.

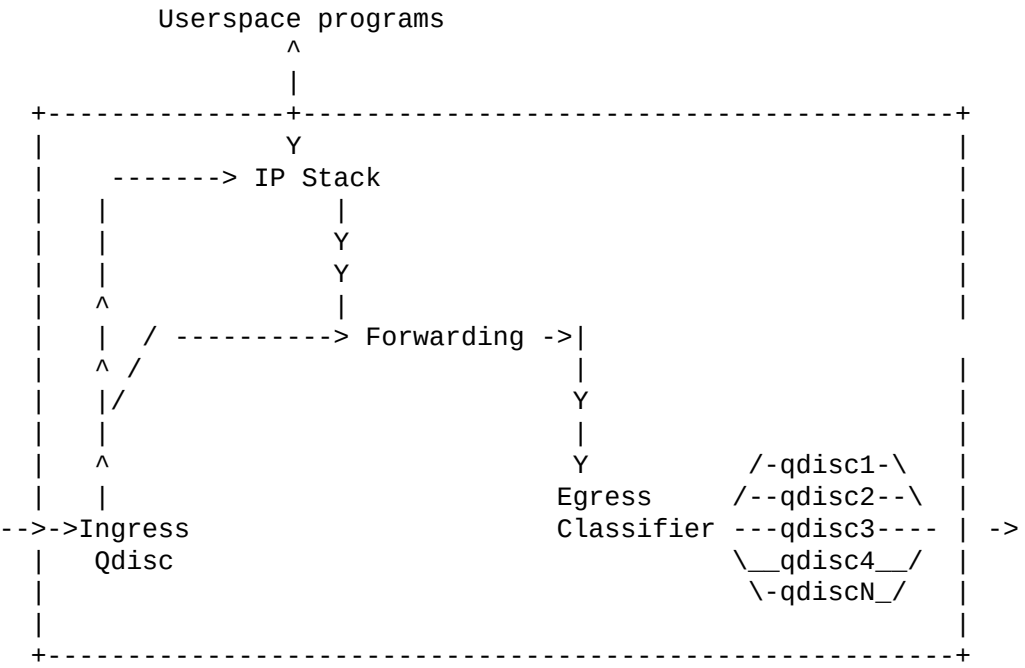
- Queueing Discipline An algorithm that manages the queue of a device, either incoming (ingress) or outgoing (egress).
- Classless qdisc A qdisc with no configurable internal subdivisions.
- Classful qdisc A classful qdisc contains multiple classes. Each of these classes contains a further qdisc, which may again be classful, but need not be. According to the strict definition, pfifo_fast *is* classful, because it contains three bands which are, in fact, classes. However, from the user's configuration perspective, it is classless as the classes can't be touched with the tc tool.
- Classes A classful qdisc may have many classes, which each are internal to the qdisc. Each of these classes may contain a real qdisc.
- Classifier Each classful qdisc needs to determine to which class it needs to send a packet. This is done using the classifier.
- Filter Classification can be performed using filters. A filter contains a number of conditions which if matched, make the filter match.
- Scheduling A qdisc may, with the help of a classifier, decide that some packets need to go out earlier than others. This process is called Scheduling, and is performed for example by the pfifo_fast qdisc mentioned earlier. Scheduling is also called 'reordering,' but this is confusing.
- Shaping The process of delaying packets before they go out to make traffic confirm to a configured maximum rate. Shaping is performed on egress. Colloquially, dropping packets to slow traffic down is also often called Shaping.
- Policing Delaying or dropping packets in order to make traffic stay below a configured bandwidth. In Linux, policing can only drop a packet and not delay it—there is no 'ingress queue.'
- Work-Conserving A work-conserving qdisc always delivers a packet if one is available. In other words, it never delays a packet if the network adaptor is ready to send one (in the case of an egress qdisc).
- non-Work-Conserving Some queues, like for example the Token Bucket Filter, may need to hold on to a packet for a certain time in order to limit the bandwidth. This means that they sometimes refuse to give up a packet, even though they have one available.

Now that we have our terminology straight, let's see where all these things are. Figure 1 shows us.

## 4 Configuration example

This example is short but useful with actual physical phone modems:

Figure 1: This schematic is due to Jamal as well.




```
# tc qdisc add dev ppp0 root \
       sfq perturb 10
```

Ok—what does this do? We've configured ppp0 to have a root queueing discipline called SFQ, which stands for Stochastic Fairness Queue. What this means is that the kernel now assigns each outgoing packet to a 'bucket' and dequeues a packet from each bucket in turn.

This is good for making sure that an outgoing upload does not interfere with, say, ssh traffic.

To inspect the configuration:

```
# tc -s -d qdisc ls dev ppp0
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.

## 5 Available Queueing Disciplines

The Linux kernel comes with many Queueing Disciplines or qdiscs. Some of there are non-functional or so underdocumented that they are not in use. There are also qdiscs that have not been merged yet.

- 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.

- Token Bucket Filter

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

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

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

  Each arriving token collects one incoming data packet from the data queue and is then deleted from the bucket.

- Stochastic Fairness Queueing

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

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

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

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

- Prio 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 stereoids, whereby each band is a separate class instead of a simple FIFO.

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

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

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

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

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

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

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

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

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

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

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

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

- Hierarchical Token Bucket (outside of the kernel) Martin Devera (<devik>) rightly realised that CBQ is complex and does not seem optimized for many typical situations. His Hierarchial approach is well suited for setups where you have a fixed amount of bandwidth which you want to divide for different purposes, giving each purpose a guaranteed bandwidth, with the possibility of specifying how much bandwidth can be borrowed.

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

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

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

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

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

- Clark-Shenker-Zhang algorithm (CSZ) This is so theoretical that not even Alexey (the main CBQ author) claims to understand it. From his source:

  "David D. Clark, Scott Shenker and Lixia Zhang Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism."

  As I understand it, the main idea is to create WFQ flows for each guaranteed service and to allocate the rest of bandwith to dummy flow-0. Flow-0 comprises the predictive services and the best effort traffic; it is handled by a priority scheduler with the highest priority band allocated for predictive services, and the rest—to the best effort packets.

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

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

- On the ingress 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.

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

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

  These issues with tail-drop are becoming increasingly troublesome on the Internet because the use of network unfriendly applications is increasing. The Linux kernel offers us RED, short for Random Early Detect, also called Random Early Drop, as that is how it works.

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

the throughput and latency of other connections.

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

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

- Generic Random Early Detection

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

  Each virtual queue can have its own Drop Parameters specified.

  "Ask Jamal"

- Weighted Round Robin (WRR) This qdisc is not included in the standard kernels but can be downloaded. Currently the qdisc is only tested with Linux 2.2 kernels, but it will probably work with 2.4/2.5 kernels too.

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

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

# 6 Kernel API

Only classless qdiscs are covered here. Writing a classful qdisc is an advanced topic.

To the kernel, a qdisc looks like Figure 2.

Packets are enqueued by the kernel and immediately after as many packets as possible are bursted out of the qdisc to the hardware.

- next Pointer in the linked list – leave alone
- cl_ops NULL for a classless qdisc
- id Name of this interface
- priv_size Size of the private data of this backend
- enqueue Called by the kernel to queue a new packet for transmission
- dequeue Called to get a packet for the hardware to send out now
- requeue Called by the kernel to put back a packet at the head of the queue. The next call to dequeue will most likely return it.
- drop Called by the kernel to indicate that a packet should be dropped & freed from the queue, without returning it

Figure 2: The `Qdisc_ops` structure

```
struct Qdisc_ops
{
    struct Qdisc_ops        *next;
    struct Qdisc_class_ops  *cl_ops;
    char                    id[IFNAMSIZ];
    int                     priv_size;

    int                     (*enqueue)(struct sk_buff *, struct Qdisc *);
    struct sk_buff *        (*dequeue)(struct Qdisc *);
    int                     (*requeue)(struct sk_buff *, struct Qdisc *);
    int                     (*drop)(struct Qdisc *);

    int                     (*init)(struct Qdisc *, struct rtattr *arg);
    void                    (*reset)(struct Qdisc *);
    void                    (*destroy)(struct Qdisc *);
    int                     (*change)(struct Qdisc *, struct rtattr *arg);

    int                     (*dump)(struct Qdisc *, struct sk_buff *);
};
```

- init Called before use
- reset Should purge the queue and reset settings
- destroy Cleanup
- change Accept reconfigured settings over the Netlink
- dump Report statistics over the Netlink

### 6.1 How the kernel interacts with the qdisc

enqueue is called from dev_queue_xmit() in net/core/dev.c:

```
/* Grab device queue */
spin_lock_bh(&dev->queue_lock);
q = dev->qdisc;
if (q->enqueue) {
    int ret = q->enqueue(skb, q);

    qdisc_run(dev);

    spin_unlock_bh(&dev->queue_lock);
    return ret == NET_XMIT_BYPASS ?
                  NET_XMIT_SUCCESS :
                  ret;
}
```

qdisc_run(), which lives in include/net/pkt_sched.h, is then immediately called to get the packets out on the wire (or ether, for that matter):

```
static inline void
qdisc_run(struct net_device *dev)
{
  while (!netif_queue_stopped(dev)
    && qdisc_restart(dev)<0)
      /* NOTHING */;
}
```

Getting nearer to the wire, qdisc_restart() is in net/sched/sch_generic.c:

```
int
qdisc_restart(struct net_device *dev)
{
  struct Qdisc *q = dev->qdisc;
  struct sk_buff *skb;

  /* Dequeue packet */
  if ((skb = q->dequeue(q)) != NULL)
  {
    if (spin_trylock(&dev->xmit_lock))
    {
```

```
/* Remember that the driver
   is grabbed by us. */
dev->xmit_lock_owner =
  smp_processor_id();

/* And release queue */
spin_unlock(&dev->queue_lock);

if (!netif_queue_stopped(dev))
{
  if (netdev_nit)
    dev_queue_xmit_nit(skb,
      dev);

  if (dev->hard_start_xmit(skb,
      dev) == 0) {
    dev->xmit_lock_owner = -1;
    spin_unlock(&dev->xmit_lock);

    spin_lock(&dev->queue_lock);
    return -1;
  }
}
/* code for when the queue
   IS stopped */
...
```

hard_start_xmit(skb,dev) actually moves (or shakes) the electrons.

### 6.2 Minimal qdisc

The kernel actually contains a 'noop' qdisc which sees some use in efficiently dropping packets on the floor. Or as Alexey says it:

```
/* "NOOP" scheduler:
the best scheduler,
recommended for all
interfaces under all
circumstances.  It is
difficult to invent
anything faster or
cheaper.  */
```

However, this is too minimal to serve as an example.

We'll look at the pfifo qdisc which performs simple taildrop after $n$ packets. Somewhat simplified and commented source of pfifo_enqueue:

```
int pfifo_enqueue(struct sk_buff *skb,
    struct Qdisc *sch)
{
  /* get our private data */
  struct fifo_sched_data *q =
    (struct
     fifo_sched_data *)sch->data;

  /* is there room for
     another packet */
  if (sch->q.qlen <= q->limit) {
    /* add it at the tail
       end of our q */
    __skb_queue_tail(&sch->q, skb);

    /* accounting - note that this is
       not in the private part */
    sch->stats.bytes += skb->len;
    sch->stats.packets++;
    /* there might be accounting in
       q-> too, but not for pfifo */
    return 0;
  }
  /* if we get here, there is no room
     and we drop & cleanup */
  sch->stats.drops++;

  kfree_skb(skb);
  /* sorry, no room */
  return NET_XMIT_DROP;
}
```

The dequeue function is simpler:

```
struct sk_buff
   *pfifo_dequeue(struct Qdisc* sch)
{
   return __skb_dequeue(&sch->q);
}
```

The pfifo queue cannot be configured—it takes its queuelength from the adapter's txqueuelen.

## References

[LARTC] *Linux Advanced Routing & Traffic Control HOWTO* bert hubert. `http://lartc.org/` (2002)

# Proceedings of the
# Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*