

Linux

Linux QOS

Andy Fletcher

andy@x31.com

TLNX002.1

2011/07/31



This presentation by [Andy Fletcher](#) is licensed under a Creative Commons [Attribution-NonCommercial-ShareAlike 3.0 Unported License](#).

Introduction

Linux implements QOS by traffic shaping and setting priority for the different traffic streams. This is done in two stages

Classification. Packets are analysed and assigned to the different queues. Used for classfull queueing disciplines such as HTB.

Queueing. The queues are processed according to their type and configuration and the packets sent out of the interface. Queues will be discussed first.

QOS is generally performed on outbound queues only. It is possible to have control of inbound traffic but this is limited to policing the traffic streams and dropping packets which exceed the threshold.

Linux supports a special queue called an Intermediate queue (IMQ). Unfortunately this is not in the standard kernel and requires the compiling of a patched source code tree. Intermediate queues can be attached to interfaces so that all received traffic goes into the IMQ. IMQs support the full classification and queueing options of a normal outbound queue.

Ethernet

The default Ethernet outbound queue is **pfifo_fast** which consists of 3 fifo queues which are serviced in strict priority order. Packets are assigned to the queues based on the TOS field in the IP header. This can be changed using the Linux traffic control (tc) commands.

By default Ethernet **pfifo_fast** outbound queues have a default length of 1000 packets. Busy gigabit interfaces may impose a reduced load on the system if the queue length is increased to 10000 packets, however this can increase traffic latency and requires more memory for the packet buffers. If packet latency is critical the queue length can be reduced.

Queue length can be changed by the following command

```
ifconfig eth0 txqueuelen 100
```

Which sets the queue length to 100 packets

Inbound packets are simply passed to the kernel after policing.

Linux

QOS – Queue Types

pfifo_fast

This classless queue consists of three fifo queues. Traffic is placed into the queues based on the TOS field of the packet. Traffic is removed from the queues in strict priority meaning that the lower priority queues will not be serviced whilst there are still packets in the higher priority queues. `pfifo_fast` does not rate limit traffic so traffic shaping must be performed by HTB or similar.

The different fifo queues are called 0, 1 and 2. where 2 is the lowest priority queue. The 16 Linux priorities are mapped to these queues using a 'map'. This map can be seen by the following command.

```
# tc -s -d qdisc show dev eth0  
qdisc pfifo_fast 0: root refcnt 2 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1  
Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)  
rate 0bit 0pps backlog 0b 0p requeues 0
```

Note this mapping cannot be changed. Priority and interactive traffic is put into queue 0, Bulk interactive and best effort is put into queue 1 and the remainder in queue 2. The only parameter which can be changed is the queue length.

Linux

QOS – Queue Types

pfifo and bfifo

These two classless queues are simply fifo queues. The length of the queue is measured in packets (pfifo) or bytes (bfifo). When the queue is full any additional packets are discarded. The following attaches a pfifo with a limit of 10000 packets directly to eth0.

```
# tc qdisc add dev eth0 root pfifo limit 10000
```

The following deletes the queue from eth0 and restores the default pfifo_fast.

```
# tc qdisc del dev eth0 root
```

Note that you have to delete the existing non-default queue before attaching a new one. Pfifo or bfifo do not rate limit traffic so traffic shaping must be performed by HTB or similar.

SFQ (Stochastic Fairness Queue)

SFQ schedules the transmission of packets, based on 'flows'. It ensures fairness so that each flow is able to send data in turn, thus preventing any single flow from drowning out the rest. SFQ will always send a packet if there is one available.

SFQ hashes the packets into 1024 buckets based on the source address, destination address and source port number. The hashing algorithm is changed every perturb period (10 seconds) to prevent two flows from being always bound into the same buckets. SFQ can queue up to 128 packets, any excess beyond this are discarded.

The packets in the SFQ buckets are dequeued in a round robin manner. The quantum is the number of bytes which can be dequeued each time and should be set to the MTU of the interface (default). SFQ does not rate limit traffic so traffic shaping must be performed by HTB or similar. The following adds a SFQ to ppp0.

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

Random Early Detection (RED) (1)

RED is a classless queue which drops packets before the queue is completely full. There is a poorly-documented variant called GRED which can have multiple thresholds.

Once the queue hits a certain average length, packets have a configurable chance of being dropped. This chance increases linearly up to a point called the max average queue length, although the queue might get bigger. TCP sessions will see the occasional dropped packet and will adjust their sending rate accordingly thereby controlling the traffic throughput to what the link can handle.

Explicit Congestion Notification (ECN) can be configured so that instead of dropping packets the queue will mark the packets so that the hosts engaged in the session can reduce traffic flow.

The average queue size is used for determining the dropping probability. When the average queue size is below min bytes, no packet will ever be dropped. When it exceeds min, the probability of doing so climbs linearly up to probability, until the average queue size hits max bytes.

Linux

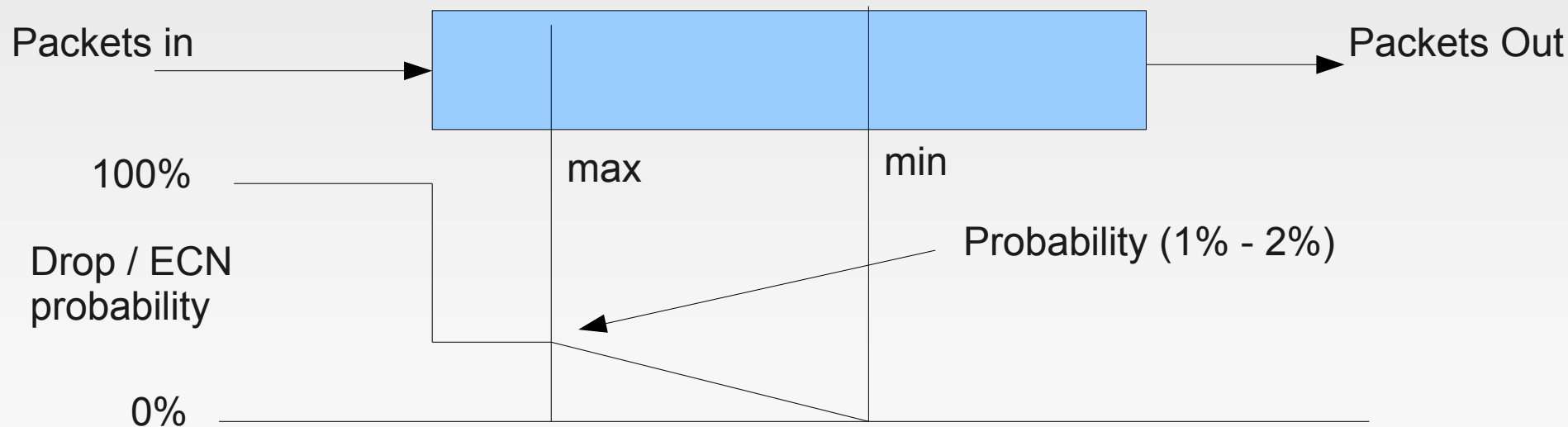
QOS – Queue Types

Random Early Detection (RED) (2)

Because probability is normally not set to 100%, the queue size might conceivably rise above max bytes, so the limit parameter is provided to set a hard maximum for the size of the queue.

RED queues are good for high bandwidth links as they are low overhead and are not limited in the number of streams which they can handle.

The diagram below shows the queue operation



Linux

QOS – Queue Types

Random Early Detection (RED) (3)

RED has lots of parameters which can be used to completely screw up the operation of it. The following adds a RED queue to a minimal HTB qdisc limited to 8Mbit.

```
tc qdisc del root dev eth0  
tc qdisc add dev eth0 root handle 1: htb default 11  
tc class add dev eth0 parent 1: classid 1:1 htb rate 8000kbit ceil 8000kbit  
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 8000kbit ceil 8000kbit  
  
tc qdisc add dev eth0 parent 1:11 red limit 180KB min 15KB max 45KB burst 25 \  
    avpkt 1000 bandwidth 10Mbit probability 0.02 ecn
```

Parameters:

Limit, maximum length of queue, drop all packets, set 4*max
Min, size when drop/ecn starts (0%).
Max, size when drop/ecn probability is a maximum (1%-2%). set >2*min
Burst, used to determine sensitivity set to (min+min+max)/(3*avpkt).
Avpkt, Average packet length, 1000 is a good value
Bandwidth, set to indicate interface speed. Is not enforced.
Probability, drop/ecn marking. keep in range 1% to 2%
Ecn, if set then ecn will be used if the originating source supports it

Token Bucket Filter (TBF)

TBF is a classless filter which will shape traffic on an interface. It is useful if all traffic on an interface needs shaping in the same manner. It is not suitable for high speed interfaces – use HTB + RED in this case.

The command below attaches a TBF with a sustained maximum rate of 0.5mbit/s, a peakrate of 1.0mbit/s, a 5kilobyte buffer, with a pre-bucket queue size limit calculated so the TBF causes at most 70ms of latency, and perfect peakrate behaviour.

```
# tc qdisc add dev eth0 root tbf rate 0.5mbit burst 5kb latency 70ms \  
peakrate 1mbit minburst 1540
```

Parameters:

Limit, maximum number of bytes in queue
Burst, bucket size in bytes
Mpu, Minimum packet size, 64 for Ethernet
Rate, target throughput
Peakrate, maximum throughput
mtu/minburst, set to mtu of the link

Hierarchy Token Bucket (HTB) (1)

HTB is a classful queuing system which can be used to attach multiple queues via subclasses to an interface. HTB then assigns bandwidth to each queue depending on their position within the class tree. Queues may borrow unused bandwidth from their peers thereby maximising overall throughput.

HTB is intuitive in design and has a lot of flexibility in the options which can be set. Parameters which can be configured include guaranteed bandwidth, peak bandwidth, sharing priority

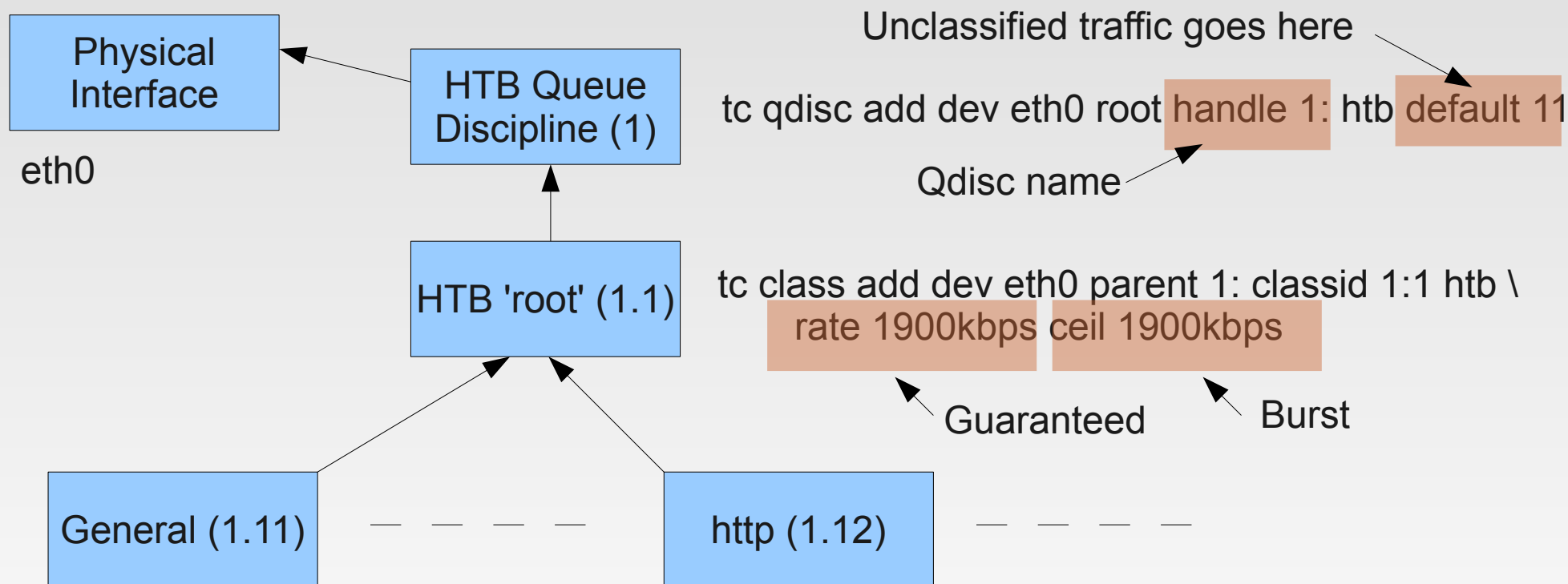
There is another classful queuing system called Class Based Queuing (CBQ) which is not discussed in this document as it isn't as flexible as HTB and is much harder to tune.

Linux

QOS – Queue Types

Hierarchy Token Bucket (HTB) (2)

HTB traffic management is intuitive to design as it is constructed in a tree structure. The diagram below shows the concept of a HTB enabled interface.



In this case we have a limit of 1900Kbit from this interface. It isn't essential to attach a single 'root' to (1) but we cannot borrow unused bandwidth between the classes at this level so it is easier to simply add a second 'root' class (1.1) and hang everything off that.

Linux

QOS – Queue Types

Hierarchy Token Bucket (HTB) (3)

In this example we have 3 types of traffic we wish to manage, ssh, http and general. We want to guarantee bandwidth for each type of traffic whilst allowing them to burst to the maximum for the link (in this case 1900KBit)

```
tc qdisc add dev eth0 root handle 1: htb default 11
```

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 1900kbps ceil 1900kbps
```

http traffic:

```
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 1000kbps ceil 1900kbps
```

general traffic:

```
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 200kbps ceil 1000kbps
```

ssh traffic:

```
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 400kbps ceil 1900kbps
```

Sum of guaranteed rates should not exceed that of parent class

Peak rates should not exceed that of parent class

Hierarchy Token Bucket (HTB) (4)

When a class is using spare bandwidth above its guaranteed figure the spare bandwidth is by default shared in proportion to the guaranteed bandwidth of each requesting class so a class with 100Kbit guaranteed bandwidth would get twice the contended spare bandwidth compared to another class with 50Kbit guaranteed bandwidth.

This default sharing of bandwidth can be changed by assigning priorities to the classes using the prio parameter. Higher priority (lower value) classes will be allocated spare bandwidth before the lower priority ones. The example below shows class 1:11 with priority 2.

```
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 200kbps ceil 1000kbps prio 2
```

Linux

QOS – Queue Types

Hierarchy Token Bucket (HTB) (5)

Classes can be extended below the level shown in last slide as far as you want. Ensure that the class id is unique in each case, its convenient to start at next multiple of 10 every time you go down a level.

In the last slide only the classes have been created, the next stage is to attach the queues to the classes so that packets have somewhere to go. In this case we add a SFQ for ssh, a RED for http and general.

http traffic:

```
tc qdisc add dev eth0 parent 1:10 red limit 180KB min 15KB max 45KB burst 25 \  
  avpkt 1000 bandwidth 1900kbit probability 0.02 ecn
```

general traffic:

```
tc qdisc add dev eth0 parent 1:11 red limit 180KB min 15KB max 45KB burst 25 \  
  avpkt 1000 bandwidth 1900kbit probability 0.02 ecn
```

ssh traffic:

```
tc qdisc add dev eth0 parent 1:12 sfq perturb 10
```

Classifiers

Classful queue disciplines need some way of assigning traffic to the different queues which are attached to an interface. In the absence of classifiers traffic will only be assigned to the default queue.

Traffic is classified using the “tc filter” command. This will identify a subset of the traffic and assign the traffic to the class and thereby the queue which should process it.

The most common ways to select traffic are:

- using the u32 classifier
- marking the packets using iptables then assigning the packets to the correct class

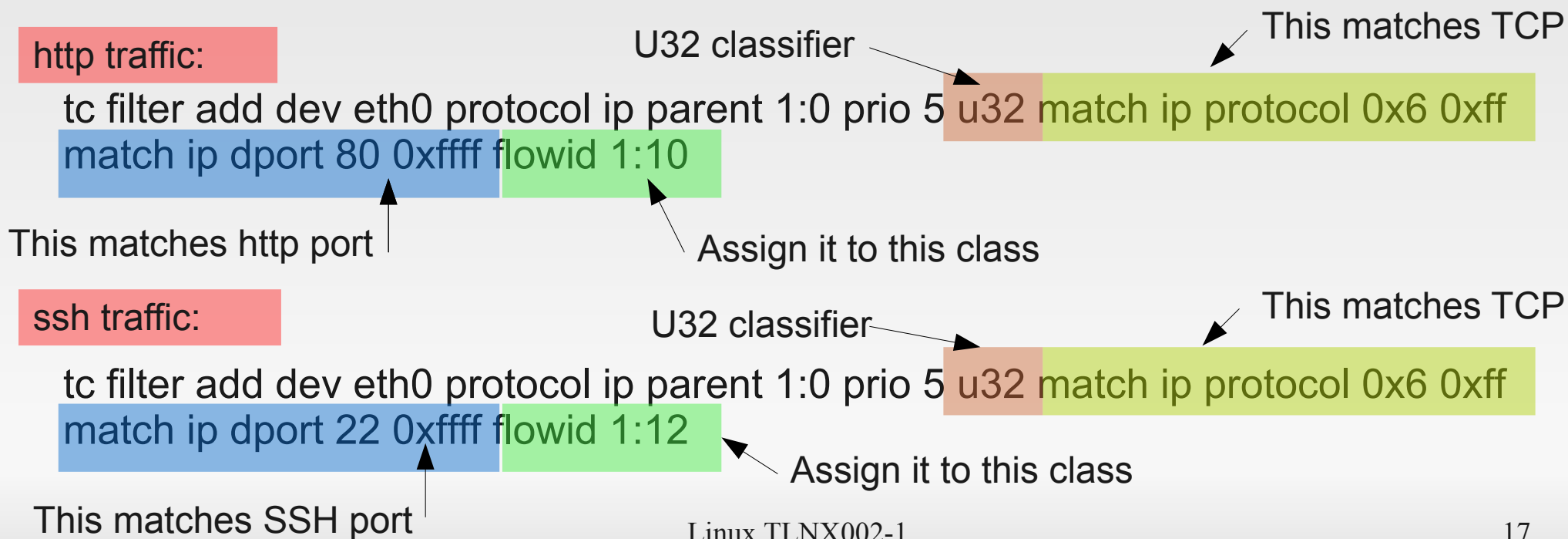
Linux

QOS – Filters

Applying filters

TC filters can be added in multiple lists which can be executed depending on the result of tests or as a simple list of filters which are executed until a match is found. Only the latter case is presented in this document.

The example below shows the ssh and http filters for the HTB example, no filter has been created for 'general' as this is the default for unclassified traffic.



The U32 Classifier

The U32 classifier performs an action depending on if there is a match or not in the packet.

U32 simply matches up to 32 bits of the packet at a defined multiple of 32 bit offset from the start of the packet. If there is a match then the defined action is performed.

There may be multiple 'match' commands in a single U32 filter command.

If all the 'matches' succeed in the U32 command then one of the following actions are taken.

- Assign the packet to a defined class, example classid 1:11 (also flowid 1:11)
- Jump to another set of filter commands with the packet, example link 1:0:
- Jump into a computed position in a table containing filter commands (hash)

Only the first (assignment of packet to a class) is considered in this presentation..

See the closing remarks for a link to a detailed description of U32.

Linux

QOS – Classifier (2)

The U32 Classifier

The basic syntax for the U32 classifier is as below

```
match u32 (value) (mask) at (offset)
```

Where

(value) is a 32 bit value to be matched

(mask) is a 32 bit mask. Only bits set to 1 are matched.

(offset) is the number of bytes the match starts from the start of the packet
(must be a multiple of 4)

However U32 has predefined ways of specifying fields and addresses to avoid having to do lots of bitwise arithmetic, some of them are shown below:

```
match ip src 192.168.8.0/24 (match a source IP address range)
match ip dst 192.168.8.0/24 (match a destination IP range)
match ip tos 0x10 1e (match the TOS byte bits 4-1)
match ip sport 80 ffff (match from port – assume normal ip header size)
match ip dport 443 ffff (match to port – assume normal ip header size)
match ip nofrag (match unfragmented packet)
```

Linux

QOS – Classifier

Using iptables to mark packets

iptables can be used to mark packets for queuing using the MARK action. This allows a wide range of conditions to be used to control traffic.

There are also cases with NAT when it is not possible to identify the internal host responsible for an outgoing packet at the external interface. This can cause problems if different internal hosts are to be assigned traffic capacity. One solution is to mark the packets on ingress to the gateway and to use the marks to select the outbound queue.

The example below uses iptables to mark outbound packets then uses the marks (handles) to direct the packets to the correct queue.

```
iptables -A OUTPUT -o eth0 -p tcp -m tcp -m multiport --dports 80,443 \  
-j MARK --set-mark 10
```

```
iptables -A OUTPUT -o eth0 -p tcp -m tcp --dport 22 -j MARK --set-mark 12
```

```
tc filter add dev wlan0 protocol ip parent 1:0 prio 5 handle 10 fw flowid 1:10  
tc filter add dev wlan0 protocol ip parent 1:0 prio 5 handle 12 fw flowid 1:12
```

Linux

QOS – Examples

Example with three queues

The example discussed in the HTB section is shown below with classifiers based on U32. In this case interface wlan0 is used.

```
tc qdisc del dev wlan0 root
tc qdisc add dev wlan0 root handle 1: htb default 11
tc class add dev wlan0 parent 1: classid 1:1 htb rate 1900kbps ceil 1900kbps
# http traffic class and queue
tc class add dev wlan0 parent 1:1 classid 1:10 htb rate 1000kbps ceil 1900kbps
tc qdisc add dev wlan0 parent 1:10 red limit 180KB min 15KB max 45KB burst 25 \
avpkt 1000 bandwidth 1900kbit probability 0.02 ecn
# general traffic class and queue
tc class add dev wlan0 parent 1:1 classid 1:11 htb rate 200kbps ceil 1000kbps
tc qdisc add dev wlan0 parent 1:11 red limit 180KB min 15KB max 45KB burst 25 \
    avpkt 1000 bandwidth 1900kbit probability 0.02 ecn
# ssh traffic class and queue
tc class add dev wlan0 parent 1:1 classid 1:12 htb rate 400kbps ceil 1900kbps
tc qdisc add dev wlan0 parent 1:12 sfq perturb 10
# Classifiers ..
tc filter add dev wlan0 protocol ip parent 1:0 prio 5 u32 match ip protocol 0x6 0xff \
    match ip dport 80 0xffff flowid 1:10
tc filter add dev wlan0 protocol ip parent 1:0 prio 5 u32 match ip protocol 0x6 0xff \
    match ip dport 22 0xffff flowid 1:12
```

Linux

QOS – Policing

Policing of inbound traffic

Unless an IMQ is used inbound traffic flows can only be policed with overlimit traffic being dropped.

The example below drops traffic inbound which exceeds 1900Kbit.

First attach the qdisc to the interface. The handle ffff is a convenient number, others can be used.

```
tc qdisc add dev eth0 handle ffff: ingress
```

Now attach a filter to the inbound qdisc

```
tc filter add dev eth0 parent ffff: protocol ip prio 50 u32 match ip src \  
0.0.0.0/0 police rate 1900kbit burst 10k drop flowid :1
```

IMQ

An alternative to basic policing is the use of IMQ. To use IMQ you need to do the following.

- Download a kernel source tree and associated utilities
- Build and test the kernel
- Download the IMQ patches from <http://www.linuximq.net/> apply them.
- Build and test the new kernel
- Add the module imq to /etc/modules
- Install the new module into the running system “modprobe imq”
- Rebuild iptables with the IMQ patches

By default you have 2 IMQ available for use (imq0, imq1), you can raise this to 16 using the parameter “numdevs=16” with the modprobe.

IMQ are attached to interfaces using the new iptables IMQ module and target “-j IMQ --todev 0”. In this case 0 represents imq0.

Queues, classes and filters can be added to IMQ in exactly the same way as any other interface.

Performance statistics

The performance of the queues and classifiers can be monitored using the three commands below.

The queues attached to the interface wlan0

```
tc -s -d qdisc show dev wlan0
```

The classes attached to interface wlan0. Includes information on average traffic throughput, dropped packets and borrowed bandwidth

```
tc -s -d class show dev wlan0
```

The traffic classifiers attached to interface wlan0. Returns nothing if no classifiers have been defined

```
tc -s -d filter show dev wlan0
```

Linux

QOS – Units

Units (1)

When specifying speeds and sizes the following methods of specifying values should be used.

Bandwidths or rates can be specified in:

kbps	Kilobytes per second
mbps	Megabytes per second
kbit	Kilobits per second
mbit	Megabits per second
bps or a bare number.	Bytes per second

Amounts of data can be specified in:

kb or k	Kilobytes
mb or m	Megabytes
mbit	Megabits
kbit	Kilobits
b or a bare number	Bytes.

Linux

QOS – Units

Units (2)

When specifying time the following methods of specifying values should be used.

Lengths of time can be specified in:

s, sec or secs	Whole seconds
ms, msec or msecs	Milliseconds
us, usec, usecs or a bare number	Microseconds.

Closing Remarks

Only a small part of the functionality of HTB, Classifiers and iptables have been covered here. The following man pages contain much more information as do resources on the Internet.

- man tc
- man tc-htb
- man tc-sfq
- man tc-red
- man tc-tbf
- man tc-pfifo
- man tc-bfifo(8)
- man tc-pfifo_fast

There is the Linux Advanced Routing & Traffic Control HOWTO site. Its a little incomplete in some areas but a good starting point:

<http://lartc.org/>

The following page has a comprehensive description of the U32 classifier:

http://b42.cz/notes/u32_classifier/

Linux

QOS

Questions?