QoS in Linux with TC and Filters

Phil Sutter (phil@nwl.cc)

January 2016

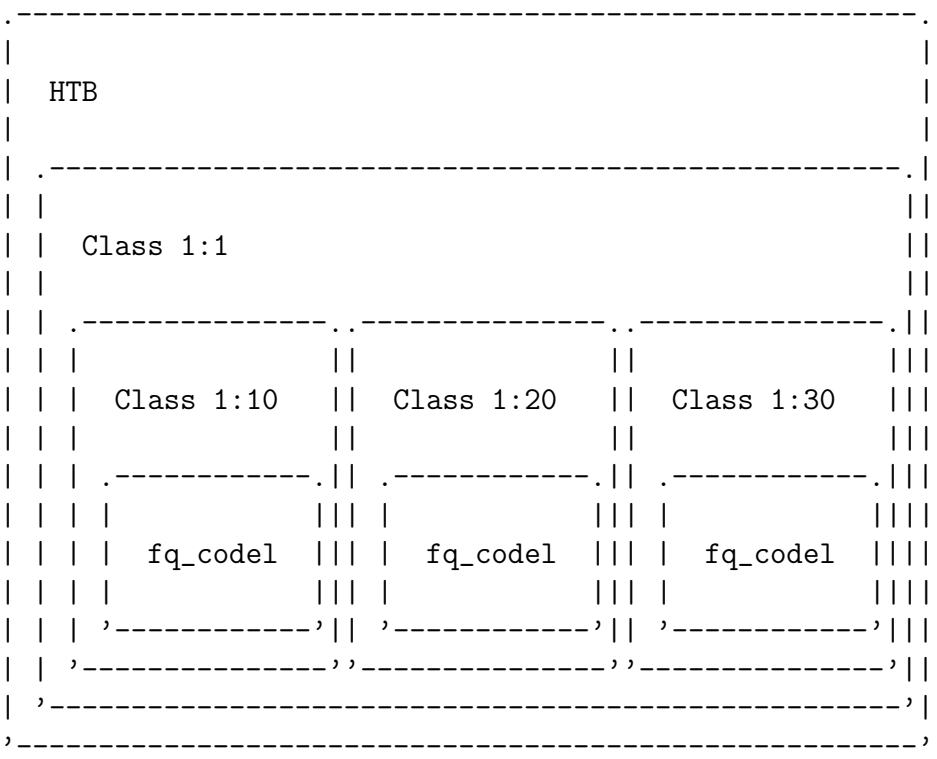
Standard practice when transmitting packets over a medium which may block (due to congestion, e.g.) is to use a queue which temporarily holds these packets. In Linux, this queueing approach is where QoS happens: A Queueing Discipline (qdisc) holds multiple packet queues with different priorities for dequeueing to the network driver. The classification (i.e. deciding which queue a packet should go into) is typically done based on Type Of Service (IPv4) or Traffic Class (IPv6) header fields but depending on qdisc implementation, might be controlled by the user as well.

Qdiscs come in two flavors, classful or classless. While classless qdiscs are not as flexible as classful ones, they also require much less customizing. Often it is enough to just attach them to an interface, without exact knowledge of what is done internally. Classful qdiscs are the exact opposite: flexible in application, they are often not even usable without insightful configuration.

As the name implies, classful qdiscs provide configurable classes to sort traffic into. In it's basic form, this is not much different than, say, the classless `pfifo_fast` which holds three queues and classifies per packet upon priority field. Though typically classes go beyond that by supporting nesting and additional characteristics like e.g. maximum traffic rate or quantum.

When it comes to controlling the classification process, filters come into play. They attach to the parent of a set of classes (i.e. either the qdisc itself or a parent class) and specify how a packet (or it's associated flow) has to look like in order to suit a given class. To overcome this simplification, it is possible to attach multiple filters to the same parent, which then consults each of them in row until the first one accepts the packet.

Before getting into detail about what filters there are and how to use them, a simple setup of a qdisc with classes is necessary:



The following commands establish the basic setup shown:

```
(1) # tc qdisc replace dev eth0 root handle 1: htb default 30
(2) # tc class add dev eth0 parent 1: classid 1:1 htb rate 95mbit
(3) # alias tclass='tc class add dev eth0 parent 1:1'
(4) # tclass classid 1:10 htb rate 1mbit ceil 20mbit prio 1
(4) # tclass classid 1:20 htb rate 90mbit ceil 95mbit prio 2
(4) # tclass classid 1:30 htb rate 1mbit ceil 95mbit prio 3
(5) # tc qdisc add dev eth0 parent 1:10 fq_codel
(5) # tc qdisc add dev eth0 parent 1:20 fq_codel
(5) # tc qdisc add dev eth0 parent 1:30 fq_codel
```

A little explanation for the unfamiliar reader:

1. Replace the root qdisc of *eth0* by an instance of `HTB`. Specifying the handle is necessary so it can be referenced in consecutive calls to `tc`. The default class for unclassified traffic is set to 30.

2. Create a single top-level class with handle 1:1 which limits the total bandwidth allowed to 95mbit/s. It is assumed that *eth0* is a 100mbit/s link, staying a little below that helps to keep the main point of enqueueing in the qdisc layer instead of the interface hardware queue or at another bottleneck in the network.

3. Define an alias for the common part of the remaining three calls in order to improve readability. This means all remaining classes are attached to the common parent class from (2).

4. Create three child classes for different uses: Class 1:10 has highest priority but is tightly limited in bandwidth - fine for interactive connections. Class 1:20 has mid priority and high guaranteed bandwidth, for high priority bulk traffic. Finally, there's the default class 1:30 with lowest priority, low guaranteed bandwidth and the ability to use the full link in case it's unused otherwise. This should be fine for uninteresting traffic not explicitly taken care of.

5. Attach a leaf qdisc to each of the child classes created in (4). Since `HTB` by default attaches `pfifo` as leaf qdisc, this step is optional. Still, the fairness between different flows provided by the classless `fq_codel` is worth the effort.

More information about the qdiscs and fine-tuning parameters can be found in *tc-htb(8)* and *tc-fq_codel(8)*.

Without any additional setup done, now all traffic leaving *eth0* is shaped to 95mbit/s and directed through class 1:30. This can be verified by looking at the `Sent` field of the class statistics printed via `tc -s class show dev eth0`: Only the root class 1:1 and it's child 1:30 should show any traffic.

Finally time to start filtering!

Let's begin with a simple one, i.e. reestablishing what `pfifo_fast` did automatically based on TOS/Priority field. Linux internally translates the header field into the priority field of struct skbuff, which `pfifo_fast` uses for classification. *tc-prio(8)* contains a table listing the priority (and ultimately, `pfifo_fast` queue index) each TOS value is being translated into. Here is a shorter version:

TOS Values	Linux Priority (Number)	Queue Index
0x0 - 0x6	Best Effort (0)	1
0x8 - 0xe	Bulk (2)	2
0x10 - 0x16	Interactive (6)	0
0x18 - 0x1e	Interactive Bulk (4)	1

Using the `basic` filter, it is possible to match packets based on that skbuff field, which has the added benefit of being IP version agnostic. Since the `HTB` setup above defaults to class ID 1:30, the Bulk priority can be ignored. The `basic` filter allows to combine matches, therefore we get along with only two filters:

```
# tc filter add dev eth0 parent 1: basic \
        match 'meta(priority eq 6)' classid 1:10
# tc filter add dev eth0 parent 1: basic \
```

```
        match 'meta(priority eq 0)' \
        or 'meta(priority eq 4)' classid 1:20
```

A detailed description of the `basic` filter and the ematch syntax it uses can be found in *tc-basic(8)* and *tc-ematch(8)*.

Obviously, this first example cries for optimization. A simple one would be to just change the default class from 1:30 to 1:20, so filters are only needed for Bulk and Interactive priorities:

```
# tc filter add dev eth0 parent 1: basic \
        match 'meta(priority eq 6)' classid 1:10
# tc filter add dev eth0 parent 1: basic \
        match 'meta(priority eq 2)' classid 1:20
```

Given that class IDs are random, choosing them wisely allows for a direct mapping. So first, recreate the qdisc and classes configuration:

```
# tc qdisc replace dev eth0 root handle 1: htb default 10
# tc class add dev eth0 parent 1: classid 1:1 htb rate 95mbit
# alias tclass='tc class add dev eth0 parent 1:1'
# tclass classid 1:16 htb rate 1mbit ceil 20mbit prio 1
# tclass classid 1:10 htb rate 90mbit ceil 95mbit prio 2
# tclass classid 1:12 htb rate 1mbit ceil 95mbit prio 3
# tc qdisc add dev eth0 parent 1:16 fq_codel
# tc qdisc add dev eth0 parent 1:10 fq_codel
# tc qdisc add dev eth0 parent 1:12 fq_codel
```

This is basically identical to above, but with changed leaf class IDs and the second priority class being the default. Using the `flow` filter with it's `map` functionality, a single filter command is enough:

```
# tc filter add dev eth0 parent 1: handle 0x1337 flow \
        map key priority baseclass 1:10
```

The `flow` filter now uses the priority value to construct a destination class ID by adding it to the value of `baseclass`. While this works for priority values of 0, 2 and 6, it will result in non-existent class ID 1:14 for Interactive Bulk traffic. In that case, the `HTB` default applies so that traffic goes into class ID 1:10 just as intended. Please note that specifying a handle is a mandatory requirement by the `flow` filter, although I didn't see where one would use that later. For more information about `flow`, see *tc-flow(8)*.

While `flow` and `basic` filters are relatively easy to apply and understand, they are as well quite limited to their intended purpose. A more flexible option is the `u32` filter, which allows to match on arbitrary parts of the packet data - yet only on that, not any meta data associated to it by the kernel (with the exception of firewall mark value). So in order to continue this little exercise with `u32`, we have to base classification directly upon the actual TOS value. An intuitive attempt might look like this:

```
# alias tcfilter='tc filter add dev eth0 parent 1:'
# tcfilter u32 match ip dsfield 0x10 0x1e classid 1:16
# tcfilter u32 match ip dsfield 0x12 0x1e classid 1:16
# tcfilter u32 match ip dsfield 0x14 0x1e classid 1:16
# tcfilter u32 match ip dsfield 0x16 0x1e classid 1:16
# tcfilter u32 match ip dsfield 0x8 0x1e classid 1:12
# tcfilter u32 match ip dsfield 0xa 0x1e classid 1:12
# tcfilter u32 match ip dsfield 0xc 0x1e classid 1:12
# tcfilter u32 match ip dsfield 0xe 0x1e classid 1:12
```

The obvious drawback here is the amount of filters needed. And without the default class, eight more filters would be necessary. This also has performance implications: A packet with TOS value 0xe will be checked eight times in total in order to determine it's destination class. While there's not much to be done about the number of filters, at least the performance problem can be eliminated by using `u32`'s hash table support:

```
# tc filter add dev eth0 parent 1: prio 99 handle 1: u32 divisor 16
```

This creates a hash table with 16 buckets. The table size is arbitrary, but not random: Since the first bit of the TOS field is not interesting, it can be ignored and therefore the range of values to consider is just [0;15], i.e. a number of 16 different values. The next step is to populate the hash table:

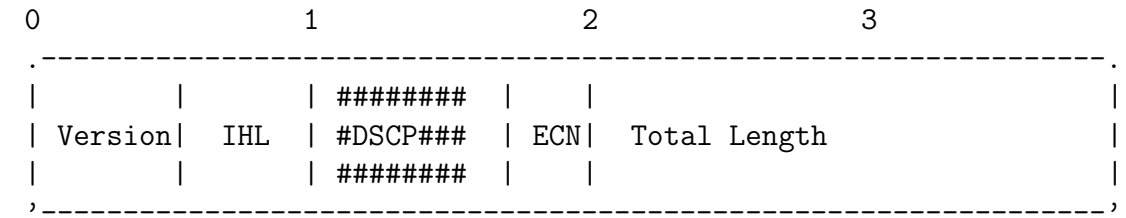
```
# alias tcfilter='tc filter add dev eth0 parent 1: prio 99'
# tcfilter u32 match u8 0 0 ht 1:0: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:1: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:2: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:3: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:4: classid 1:12
# tcfilter u32 match u8 0 0 ht 1:5: classid 1:12
# tcfilter u32 match u8 0 0 ht 1:6: classid 1:12
# tcfilter u32 match u8 0 0 ht 1:7: classid 1:12
# tcfilter u32 match u8 0 0 ht 1:8: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:9: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:a: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:b: classid 1:16
# tcfilter u32 match u8 0 0 ht 1:c: classid 1:10
# tcfilter u32 match u8 0 0 ht 1:d: classid 1:10
# tcfilter u32 match u8 0 0 ht 1:e: classid 1:10
# tcfilter u32 match u8 0 0 ht 1:f: classid 1:10
```

The parameter `ht` denotes the hash table and bucket the filter should be added to. Since the first TOS bit is ignored, it's value has to be divided by two in order to get to the bucket it maps to. E.g. a TOS value of 0x10 will therefore map to bucket 0x8. For the sake of completeness, all possible values are mapped and therefore a configurable default class is not

required. Note that the used match expression is not necessary, but mandatory. Therefore anything that matches any packet will suffice. Finally, a filter which links to the defined hash table is needed:

```
# tc filter add dev eth0 parent 1: prio 1 protocol ip u32 \
        link 1: hashkey mask 0x001e0000 match u8 0 0
```

Here again, the actual match statement is not necessary, but syntactically required. All the magic lies within the `hashkey` parameter, which defines which part of the packet should be used directly as hash key. Here's a drawing of the first four bytes of the IPv4 header, with the area selected by `hashkey mask` highlighted:



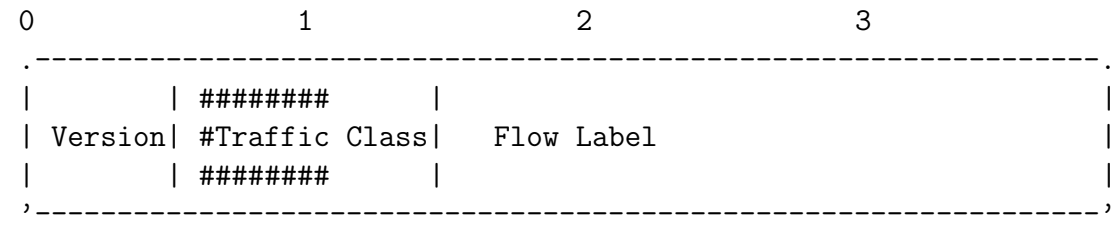
This may look confusing at first, but keep in mind that bit- as well as byte-ordering here is LSB while the mask value is written in MSB we humans use. Therefore reading the mask is done like so, starting from left:

1. Skip the first byte (which contains Version and IHL fields).
2. Skip the lowest bit of the second byte (0x1e is even).
3. Mark the four following bits (0x1e is 11110 in binary).
4. Skip the remaining three bits of the second byte as well as the remaining two bytes.

Before doing the lookup, the kernel right-shifts the masked value by the amount of zero-bits in `mask`, which implicitly also does the division by two which the hash table depends on. With this setup, every packet has to pass exactly two filters to be classified. Note that this filter is limited to IPv4 packets: Due to the related Traffic Class field being at a different offset in the packet, it would not work for IPv6. To use the same setup for IPv6 as well, a second entry-level filter is necessary:

```
# tc filter add dev eth0 parent 1: prio 2 protocol ipv6 u32 \
        link 1: hashkey mask 0x01e00000 match u8 0 0
```

For illustration purposes, here again is a drawing of the first four bytes of the IPv6 header, again with masked area highlighted:



Reading the mask value is analogous to IPv4 with the added complexity that Traffic Class spans over two bytes. Yet, for comparison there's a simple trick: IPv6 has the interesting field shifted by four bits to the left, and the new mask's value is shifted by the same amount. For further information about `u32` and what can be done with it, consult it's man page *tc-u32(8)*.

Of course, the kernel provides many more filters than just `basic`, `flow` and `u32` which have been presented above. As of now, the remaining ones are:

bpf Filtering using Berkeley Packet Filter programs. The program's return code determines the packet's destination class ID.

cgroup Filter packets based on control groups. This is only useful for packets originating from the local host, as control groups only exist in that scope.

flower An extended variant of the flow filter.

fw Matches on firewall mark values previously assigned to the packet by netfilter (or a filter action, see below for details). This allows to export the classification algorithm into netfilter, which is very convenient if appropriate rules exist on the same system in there already.

route Filter packets based on matching routing table entry. Basically equivalent to the `fw` filter above, to make use of an already existing extensive routing table setup.

rsvp, rsvp6 Implementation of the Resource Reservation Protocol in Linux, to react upon requests sent by an RSVP daemon.

tcindex Match packets based on tcindex value, which is usually set by the dsmark qdisc. This is part of an approach to support Differentiated Services in Linux, which is another topic on it's own.

Filter Actions

The tc filter framework provides the infrastructure to another extensible set of tools as well, namely tc actions. As the name suggests, they allow to do things with packets (or associated data). (The list of) Actions are part of a given filter. If it matches, each action it contains is executed in order before returning the classification result. Since the action has direct access

to the latter, it is in theory possible for an action to react upon or even change the filtering result - as long as the packet matched, of course. Yet none of the currently in-tree actions make use of this.

The Generic Actions framework originally evolved out of the filters' ability to police traffic to a given maximum bandwidth. One common use case for that is to limit ingress traffic, dropping packets which exceed the threshold. A classic setup example is like so:

```
# tc qdisc add dev eth0 handle ffff: ingress
# tc filter add dev eth0 parent ffff: u32 \
        match u32 0 0
        police rate 1mbit burst 100k
```

The ingress qdisc is not a real one, but merely a point of reference for filters to attach to which should get applied to incoming traffic. The `u32` filter added above matches on any packet and therefore limits the total incoming bandwidth to 1mbit/s, allowing bursts of up to 100kbytes. Using the new syntax, the filter command changes slightly:

```
# tc filter add dev eth0 parent ffff: u32 \
        match u32 0 0 \
        action police rate 1mbit burst 100k
```

The important detail is that this syntax allows to define multiple actions. E.g. for testing purposes, it is possible to redirect exceeding traffic to the loopback interface instead of dropping it:

```
# tc filter add dev eth0 parent ffff: u32 \
        match u32 0 0 \
        action police rate 1mbit burst 100k conform-exceed pipe \
        action mirred egress redirect dev lo
```

The added parameter `conform-exceed pipe` tells the police action to allow for further actions to handle the exceeding packet.

Apart from `police` and `mirred` actions, there are a few more. Here's a full list of the currently implemented ones:

bpf Apply a Berkeley Packet Filter program to the packet.

connmark Set the packet's firewall mark to that of it's connection. This works by searching the conntrack table for a matching entry. If found, the mark is restored.

csum Trigger recalculation of packet checksums. The supported protocols are: IPv4, ICMP, IGMP, TCP, UDP and UDPLite.

ipt Pass the packet to an iptables target. This allows to use iptables extensions directly instead of having to go the extra mile via setting an arbitrary firewall mark and matching on that from within netfilter.

mirred Mirror or redirect packets. This is often combined with the ifb pseudo device to share a common QoS setup between multiple interfaces or even ingress traffic.

nat Perform stateless Native Address Translation. This is certainly not complete and therefore inferior to NAT using iptables: Although the kernel module decides between TCP, UDP and ICMP traffic, it does not handle typical problematic protocols such as active FTP or SIP.

pedit Generic packet editing. This allows to alter arbitrary bytes of the packet, either by specifying an offset into the packet or by naming a packet header and field name to change. Currently, the latter is implemented only for IPv4 yet.

police Apply a bandwidth rate limiting policy. Packets exceeding it are dropped by default, but may optionally be handled differently.

simple This is rather an example than real action. All it does is print a user-defined string together with a packet counter. Useful maybe for debugging when filter statistics are not available or too complicated.

skbedit Edit associated packet data, supports changing queue mapping, priority field and firewall mark value.

vlan Add/remove a VLAN header to/from the packet. This might serve as alternative to using 802.1Q pseudo-interfaces in combination with routing rules when e.g. packets for a given destination need to be encapsulated.

Intermediate Functional Block

The Intermediate Functional Block (`ifb`) pseudo network interface acts as a QoS concentrator for multiple different sources of traffic. Packets from or to other interfaces have to be redirected to it using the `mirred` action in order to be handled, regularly routed traffic will be dropped. This way, a single stack of qdiscs, classes and filters can be shared between multiple interfaces.

Here's a simple example to feed incoming traffic from multiple interfaces through a Stochastic Fairness Queue (`sfq`):

```
(1) # modprobe ifb
(2) # ip link set ifb0 up
(3) # tc qdisc add dev ifb0 root sfq
```

The first step is to load the `ifb` kernel module (1). By default, this will create two ifb devices: *ifb0* and *ifb1*. After setting *ifb0* up in (2), the root qdisc is replaced by `sfq` in (3). Finally, one can start redirecting ingress traffic to *ifb0*, e.g. from *eth0*:

```
# tc qdisc add dev eth0 handle ffff: ingress
# tc filter add dev eth0 parent ffff: u32 \
        match u32 0 0 \
        action mirred egress redirect dev ifb0
```

The same can be done for other interfaces, just replacing *eth0* in the two commands above. One thing to keep in mind here is the asymmetrical routing this creates within the host doing the QoS: Incoming packets enter the system via *ifb0*, while corresponding replies leave directly via *eth0*. This can be observed using `tcpdump` on *ifb0*, which shows the input part of the traffic only. What's more confusing is that `tcpdump` on *eth0* shows both incoming and outgoing traffic, but the redirection is still effective - a simple prove is setting *ifb0* down, which will interrupt the communication. Obviously `tcpdump` catches the packets to dump before they enter the ingress qdisc, which is why it sees them while the kernel itself doesn't.

Conclusion

Once the steep learning curve has been mastered, the conglomerate of (classful) qdiscs, filters and actions provides a highly sophisticated and flexible infrastructure to perform QoS, which plays nicely along with routing and firewalling setups.

Further Reading

A good starting point for novice users and experienced ones diving into unknown areas is the extensive HOWTO at `http://lartc.org`. The iproute2 package ships some examples (usually in /usr/share/doc/, depending on distribution) as well as man pages for `tc` in general, qdiscs and filters. The latter have been added just recently though, so if your distribution does not ship iproute2 version 4.3.0 yet, these are not in there. Apart from that, the internet is a spring of HOWTOs and scripts people wrote - though these should be taken with a grain of salt: The complexity of the matter often leads to copying others' solutions without much validation, which allows for less optimal or even obsolete implementations to survive much longer than desired.