# Advanced traffic control

The Linux kernel's network stack has network traffic control and shaping features. The **iproute2 (https://www.archlinux.org/packages/?name=iproute2)** package installs the `tc` command to control these via the command line.

The goal of this article is to show how to shape the traffic by using queueing disciplines. For instance, if you ever had to forbid downloads or torrents on a network that you admin, and not because you were against those services, but because users were "abusing" the bandwidth, then you could use queueing disciplines to allow that kind of traffic and, at the same time, be sure that one user cannot slowdown the entire network.

This is an advanced article; you are expected to have certain knowledge of network devices, iptables, etc.

## Contents



## Queuing

Queuing controls how data is **sent**; receiving data is much more reactive with fewer network-oriented controls. However, since TCP/IP packets are sent using a slow start the system starts sending the packets slow and keeps sending them faster and faster until packets start getting rejected - it is therefore possible to control how much traffic is received on a LAN by dropping packets that arrive at a router before they get forwarded. There are more relevant details, but they do not touch directly on queuing logic.

In order to be the ones fully controlling the shape of the traffic, we need to be the slowest link of the chain. That is, if the connection has a maximum download speed of 500k, if you do not limit of the output to 450k or below it is going to be the modem shaping the traffic instead of us.

Each network device has a *root* where a qdisc can be set. This root has a fq_codel qdisc by default. (more info below)

There are two kind of disciplines: classful and classless.

Classful qdiscs allow you to create classes, which work like branches on a tree. You can then set rules to filter packets into each class. Each class can itself have assigned other classful or classless qdisc.

Classless qdiscs do not allow to add more qdiscs to it.

Before starting to configure qdiscs, first we need to remove any existing qdisc from the root. This will remove any qdisc from the eth0 device:

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

## Classless Qdiscs

These are queues that do basic management of traffic by reordering, slowing or dropping packets. This qdiscs do not allow the creation of classes.

#### fifo_fast

This was the default qdisc up until systemd 217. In every network device where no custom qdisc configuration has been applied, fifo_fast is the qdisc set on the root. fifo means *First In First Out*, that is, the first packet to get in, is going to be the first to be sent. This way, no package gets special treatment.

#### Token Bucket Filter (TBF)

This qdisc allows bytes to pass, as long certain rate limit is not passed.

It works by creating a virtual bucket and then dropping tokens at certain speed, filling that bucket. Each package takes a virtual token from the bucket, and uses it to get a permission to pass. If too many packets arrive, the bucket will have no more tokens left and the remaining packets are going to wait certain time for new tokens. If the tokens do not arrive fast enough, the packets are going to be dropped. On the opposite case (too few packets sent), the tokens can be used to allow some burst (uploading spikes) to happen.

That means this qdisc is useful to slow down an interface.

Example:

Uploading can fill a modem's queue and, as result, while you are uploading a huge file, the interactivity is destroyed.

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

Note the above upload speed should be changed to your upload speed minus a small few percent (to be the slowest link of the chain). This configuration sets a TBF for the `ppp0` device, limiting the upload speed to 220k, setting a latency of 50ms for a package before being dropped, and a burst of 1540. It works by keeping the queueing on the Linux machine (where it can be shaped) instead of the modem.

#### Stochastic Fairness Queueing (SFQ)

This is a round-robin qdisc. Each conversation is set on a fifo queue, and on each round, each conversation has the possibility to send data. That is why it is called "Fairness". It is also called "Stochastic" because it does not really create a queue for each conversation, instead it uses a hashing algorithm. For the hash, there

is a chance for multiple sessions on the same bucket. To solve this, SFQ changes its hashing algorithm often to prevent that this becomes noticeable.

Example:

This configuration sets SFQ on the root on the eth0 device, configuring it to perturb (alter) its hashing algorithm every 10 seconds.

```
# tc qdisc add dev eth0 root sfq perturb 10
```

#### CoDel and Fair Queueing CoDel

Since systemd 217, fq_codel is the default. **CoDel** (Controlled Delay) is an attempt to limit buffer bloating and minimize latency in saturated network links by distinguishing good queues (that empty quickly) from bad queues that stay saturated and slow. The **fair queueing** Codel utilizes fair queues to more readily distribute available bandwidth between Codel flows. The configuration options are limited intentionally, since the algorithm is designed to work with dynamic networks, and there are some corner cases to consider that are discussed on the **bufferbloat wiki concerning Codel (http://www.bufferbloat.net/projects/codel/wiki)**, including issues on very large switches and sub megabit connections.

Additional information is available via the **`tc-codel(8)` (https://jlk.fjfi.cvut.cz/arch/manpages/man/tc-codel.8)** and **`tc-fq_codel(8)` (https://jlk.fjfi.cvut.cz/arch/manpages/man/tc-fq_codel.8)**.

**Warning:** Make sure your ethernet driver supports Byte Queue Limits before using CoDel. **Here is a list of drivers supported as of kernel 3.6 (http://www.bufferbloat.net/projects/bloat/wiki/BQL_enabled_drivers)**

### Classful Qdiscs

Classful qdiscs are very useful if you have different kinds of traffic which should have differing treatment. A classful qdisc allows you to have branches. The branches are called classes.

Setting a classful qdisc requires that you name each class. To name a class,the `classid` parameter is used . The `parent` parameter, as the name indicates, points to the parent of the class.

All the names should be set as `x:y` where `x` is the name of the root, and `y` is the name of the class. Normally, the root is called `1:` and its children are things like `1:10`

#### Hierarchical Token Bucket (HTB)

HTB 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. Here is an example with comments explaining what each line does:

```
# This line sets a HTB qdisc on the root of eth0, and it specifies that the class 1:30 is used by default
. It sets the name of the root as 1:, for future references.
tc qdisc add dev eth0 root handle 1: htb default 30

# This creates a class called 1:1, which is direct descendant of root (the parent is 1:), this class gets
assigned also an HTB qdisc, and then it sets a max rate of 6mbits, with a burst of 15k
tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k
```

```
# The previous class has this branches:

# Class 1:10, which has a rate of 5mbit
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5mbit burst 15k

# Class 1:20, which has a rate of 3mbit
tc class add dev eth0 parent 1:1 classid 1:20 htb rate 3mbit ceil 6mbit burst 15k

# Class 1:30, which has a rate of 1kbit. This one is the default class.
tc class add dev eth0 parent 1:1 classid 1:30 htb rate 1kbit ceil 6mbit burst 15k

# Martin Devera, author of HTB, then recommends SFQ for beneath these classes:
tc qdisc add dev eth0 parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev eth0 parent 1:20 handle 20: sfq perturb 10
tc qdisc add dev eth0 parent 1:30 handle 30: sfq perturb 10
```

# Filters

Once a classful qdisc is set on root (which may contain classes with more classful qdiscs), it is necessary to use filters to indicate which package should be processed by which class.

On a classless-only environment, filters are not necessary.

You can filter packets by using tc, or a combination of tc + iptables.

## Using tc only

Here is an example explaining a filter:

```
# This command adds a filter to the qdisc 1: of dev eth0, set the
# priority of the filter to 1, matches packets with a
# destination port 22, and make the class 1:10 process the
# packets that match.
tc filter add dev eth0 protocol ip parent 1: prio 1 u32 match ip dport 22 0xffff flowid 1:10

# This filter is attached to the qdisc 1: of dev eth0, has a
# priority of 2, and matches the ip address 4.3.2.1 exactly, and
# matches packets with a source port of 80, then makes class
# 1:11 process the packets that match
tc filter add dev eth0 parent 1: protocol ip prio 2 u32 match ip src 4.3.2.1/32 match ip sport 80 0xffff
flowid 1:11
```

## Using tc + iptables

iptables has a method called fwmark that can be used to mark packets across interfaces.

First, this makes packets marked with 6, to be processed by the 1:30 class

```
# tc filter add dev eth0 protocol ip parent 1: prio 1 handle 6 fw flowid 1:30
```

This sets that mark 6, using iptables

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

You can then use iptables normally to match packets and then mark them with fwmark.

# Example of ingress traffic shaping with SNAT

Qdiscs on ingress traffic provide only policing with no shaping. In order to shape ingress, the IFB (Intermediate Functional Block) device has to be used. However, another problem arises if SNAT or MASQUERADE is in use, as all incoming traffic has the same destination address. The Qdisc intercepts the incoming traffic on the external interface before reverse NAT translation so it can only see the router's IP as destination of the packets.

The following solution is implemented on OpenWRT and can be applied to Archlinux: First the outgoing packets are marked with MARK and the corresponding connections (and related connections) with CONNMARK. On the incoming packets an ingress u32 filter redirects the traffic to IFB (action mirred), and also retrieves the mark of the packet from CONNTRACK (action connmark) thus providing information as to which IP behind the NAT initiated the traffic).

This function is integrated in kernel since **linux (https://www.archlinux.org/packages/?name=linux)**-3.19 and in **iproute2 (https://www.archlinux.org/packages/?name=iproute2)** since 4.1.

The following is a small script with only 2 HTB classes on ingress to demonstrate it. Traffic defaults to class 3:30. Outgoing traffic from 192.168.1.50 (behind NAT) to the Internet is marked with "3" and thus incoming packets from the Internet going to 192.168.1.50 are marked also with "3" and are classified on 3:33.

```
#!/bin/sh -x

# Maximum allowed downlink. Set to 90% of the achievable downlink in kbits/s
DOWNLINK=1800

# Interface facing the Internet
EXTDEV=enp0s3

# Load IFB, all other modules all loaded automatically
modprobe ifb
ip link set dev ifb0 down

# Clear old queuing disciplines (qdisc) on the interfaces and the MANGLE table
tc qdisc del dev $EXTDEV root    2> /dev/null > /dev/null
tc qdisc del dev $EXTDEV ingress 2> /dev/null > /dev/null
tc qdisc del dev ifb0 root       2> /dev/null > /dev/null
tc qdisc del dev ifb0 ingress    2> /dev/null > /dev/null
iptables -t mangle -F
iptables -t mangle -X QOS

# appending "stop" (without quotes) after the name of the script stops here.
if [ "$1" = "stop" ]
then
        echo "Shaping removed on $EXTDEV."
        exit
fi

ip link set dev ifb0 up

# HTB classes on IFB with rate limiting
tc qdisc add dev ifb0 root handle 3: htb default 30
tc class add dev ifb0 parent 3: classid 3:3 htb rate ${DOWNLINK}kbit
tc class add dev ifb0 parent 3:3 classid 3:30 htb rate 400kbit ceil ${DOWNLINK}kbit
tc class add dev ifb0 parent 3:3 classid 3:33 htb rate 1400kbit ceil ${DOWNLINK}kbit

# Packets marked with "3" on IFB flow through class 3:33
tc filter add dev ifb0 parent 3:0 protocol ip handle 3 fw flowid 3:33

# Outgoing traffic from 192.168.1.50 is marked with "3"
iptables -t mangle -N QOS
iptables -t mangle -A FORWARD -o $EXTDEV -j QOS
iptables -t mangle -A OUTPUT -o $EXTDEV -j QOS
```

```
iptables -t mangle -A QOS -j CONNMARK --restore-mark
iptables -t mangle -A QOS -s 192.168.1.50 -m mark --mark 0 -j MARK --set-mark 3
iptables -t mangle -A QOS -j CONNMARK --save-mark

# Forward all ingress traffic on internet interface to the IFB device
tc qdisc add dev $EXTDEV ingress handle ffff:
tc filter add dev $EXTDEV parent ffff: protocol ip \
        u32 match u32 0 0 \
        action connmark \
        action mirred egress redirect dev ifb0 \
        flowid ffff:1


exit 0
```

# See also

- **Linux Advanced Routing & Traffic Control (http://lartc.org)**
- **Wikipedia page for the tc command**

Retrieved from "https://wiki.archlinux.org/index.php?title=Advanced_traffic_control&oldid=505353"

- This page was last edited on 31 December 2017, at 06:27.
- Content is available under GNU Free Documentation License 1.3 or later unless otherwise noted.