tc - traffic control

Linux QoS control tool

April 24, 2009

Milan P. Stanic

mps@arvanta.net

Contents

About this document

This document should be (comprehensive) description of tc command utility from iproute2 package.

Primary motivation for this work is my wish to learn about QoS in Linux (and about QoS in general). If you find errors or big mistakes in this document that is because I don't yet understand QoS. I hope it will improve over time.

It is based on kernel 2.4 and iproute2 version 000305

It is far from to be complete and/or without errors. I am writing it for purpose of my learning only, and I am not sure will (and when) it be finished. But, I am working on it (especially when I have time :).

All of the text is taken from different documents from the net, from linux-diffserv, linux-net mailing lists and from the Linux kernel source files.

Disclaimer: Use at your own risk. I am not responsible if you make loss or damage in any sense by using information from this document.

1 What is QoS

When the kernel has several packets to send out over a network device, it has to decide which ones to send first, which ones to delay, and which ones to drop. This is the job of the packet scheduler, and several different algorithms for how to do this "fairly" have been proposed.

With Linux QoS subsystem (which is constructed of the building blocks of the kernel and user space tools like ip and tc command line utilities) it is possible to make very flexible traffic control.

2 command syntax

tc (traffic controller) is the user level program which can be used to create and associate queues with the network devices. It is used to set up various kinds of queues and associate classes with each of those queues. It is also used to set up filters by which the packets is classified.

```
Usage: tc [ OPTIONS ] OBJECT { COMMAND | help }
where OBJECT := { qdisc | class | filter }
OPTIONS := { -s[tatistics] | -d[etails] | -r[aw] }
```

Where it's expecting a number for BPS; it understands some suffixes: kbps (*1024), mbps (*1024*1024), kbit (*1024/8), and mbit (*1024*1024/8). If I'm reading the code correctly; "BPS" means Bytes Per Second; if you give a number without a suffix it assumes you want BITS per second (it divides the number you give it by 8). It also understands bps as a suffix.

Where it's expecting a time value, it seems it understands suffixes of s, sec, and secs for seconds, ms, msec, and msecs for milliseconds, and us, usec, and usecs for microseconds.

Where it wants a size parameter, it assumes non-suffixed numbers to be specified in bytes. It also understands suffixes of k and kb to mean kilobytes (*1024), m and mb to mean megabytes (*1024*1024), kbit to mean kilobit (*1024/8), and mbit to mean megabits (*1024*1024/8).

1Mbit == 128Kbps or 1 megabit is 128 kilobytes per second

bps = bits/sec (uhmm...)

kbps = bytes/sec * 1024

mbps = bytes/sec * 1024 * 1024

kbit = bits/sec * 1024

mbit = bits/sec * 1024 * 1024

In the examples Xbit and Xbps are interchangeably, when tc treats them very differently.

note: this is very confusing

note: make sure whenever you are dealing with memory related things like queue size, buffer size that their units are in bytes and when it is bandwidth and rate related parameters the units are in bits.

3 Queueing disciplines

Each network device has a queuing discipline associated with it, which controls how packets enqueued on that device are treated. It can be viewed with ip command:

```
root@dl:~# ip link show
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
link/ether 52:54:00:de:bf:19 brd ff:ff:ff:ff:ff:ff
3: tap0: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noop
link/ether fe:fd:00:00:00:00 brd ff:ff:ff:ff:ff:ff
```

Generally, queueing discipline ("qdisc") is a black box, which is able to enqueue packets and to dequeue them (when device is ready to send something) in order and at times determined by algorithm hidden in it.

By default queueing discipline is pfifo_fast which cannot be manipulated with tc. It is assigned to device when the device is started or when the other qdisc's deleted from the device. That qdiscs have 3 bands which are processed from band 0 to band 2, and when there is a packet in queue in higher priority band (lower number)

Qdisc's are:

- FIFO - simple FIFO (packet (p-FIFO) or byte (b-FIFO))
- PRIO - n-band strict priority scheduler
- TBF - token bucket filter
- CBQ - class based queue
- CSZ - Clark-Scott-Zhang
- SFQ - stochastic fair queue
- RED - random early detection
- GRED - generalized random early detection
- TEQL - traffic equalizer
- ATM - asynchronous transfer mode
- DSMARK - DSCP (Diff-Serv Code Point)marker/remarker

qdisc's are divided to two categories:

- "queues", which have no internal structure visible from outside.

- "schedulers", which split all the packets to "traffic classes", using "packet classifiers". ? is qdisc's which can split packets to "traffic classes"

In turn, classes may have child qdiscs (as rule, queues) attached to them etc. etc. etc.

note: Certain qdiscs can have children and they are classfull, and others are leafs (describe it!)

classfull qdiscs: CBQ, ATM, DSMARK, CSZ and the (p-FIFO ???? or prio)

leaf qdiscs: TBF, FIFO, SFQ, RED, GRED, TEQL

note: classfull qdiscs can be also leafs

The syntax for managing queuing discipline is:

```
Usage: tc qdisc [ add | del | replace | change | get ] dev STRING
[ handle QHANDLE ] [ root | ingress | parent CLASSID ]
[ estimator INTERVAL TIME_CONSTANT ]
[ [ QDISC_KIND ] [ help | OPTIONS ] ]
tc qdisc show [ dev STRING ] [ingress]
Where:
QDISC_KIND := { [p|b]fifo | tbf | prio | cbq | red | etc. }
OPTIONS := ... try tc qdisc add <desired QDISC_KIND> help
```

add ads a qdisc to device dev

del delete qdisc from device dev

replace replace the qdisc with another

handle represents the unique handle that is assigned by the user to the queuing discipline. No two queuing disciplines can have the same handle. Qdisc handles always have minor number equal to zero.

root indicates that the queue is at the root of a link sharing hierarchy and own all bandwidth on that device. Can only have one root qdisc per device.

ingress policing on the ingress

parent represents the handle of the parent queuing discipline.

dev is network device to which we want attach qdisc

estimator is used to determine if the requirements of the queue have been satisfied. The INTERVAL and the TIME_CONSTANT are two parameters that are of very high significance to the estimator. The estimator estimate the bandwidth used by each class over the appropriate time interval, to determine whether or not each class has been receiving its link sharing bandwidth.

```
Usage: ... estimator INTERVAL TIME-CONST
INTERVAL is interval between measurements
TIME-CONST is averaging time constant
Example: ... est 1sec 8sec
```

The time constant for the estimator is a critical parameter; this time constant determines the interval over which the router attempts to enforce the link-sharing guidelines.

[1]Unfortunately, rate estimation is not a very easy task. F.e. I did not find a simple way to estimate the current peak rate and even failed to formulate the problem. So I preferred not to built an estimator into the scheduler, but run this task separately. Ideally, it should be kernel thread(s), but for now it runs from timers, which puts apparent top bounds on the number of rated flows, has minimal overhead on small, but is enough to handle controlled load service, sets of aggregates.

We measure rate over A=(1<<interval) seconds and evaluate EWMA:

avrate = avrate*(1-W) + rate*W

where W is chosen as negative power of 2: W = 2^(-ewma_log)

The resulting time constant is:

T = A/(-ln(1-W))

NOTES.

* The stored value for avbps is scaled by 2^5, so that maximal rate is ~1Gbit, avpps is scaled by 2^10.

* Minimal interval is HZ/4=250msec (it is the greatest common divisor for HZ=100 and HZ=1024 8)), maximal interval is (HZ/4)*2^EST_MAX_INTERVAL = 8sec. Shorter intervals are too expensive, longer ones can be implemented at user level painlessly.

You *have* to declare first, the CBQ qdisc, then the CBQ "parent" class, and then (optionally, I think), the CBQ "leaf " classes.

I'm not 100% sure of what I've just said. It's just how I think it works.

to stop QoS completely use the following for eth0:

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

3.1 Class Based Queue

In CBQ, every class has variables idle and avgidle and parameter maxidle used in computing the limit status for the class, and the parameter offtime used in determining how long to restrict throughput for overlimit classes.

idle: The variable idle is the difference between the desired time and the measured actual time between the most recent packet transmissions for the last two packets sent from this class. When the connection is sending more than its allocated bandwidth, then idle is negative. When the connection is sending perfectly at its alloted rate, then idle is zero.

avgidle: The variable avgidle is the average of idle, and it computed using an exponential weighted moving average (EWMA). When the avgidle is zero or lower, then the class is overlimit (the class has been exceeding its allocated bandwidth in a recent short time interval).

maxidle: The parameter maxidle gives an upper bound for avgidle. Thus maxidle limits the credit given to a class that has recently been under its allocation.

offtime: The parameter offtime gives the time interval that a overlimit must wait before sending another packet. This parameter determines the steady-state burst size for a class when the class is running over its limit.

minidle: The minidle parameter gives a (negative) lower bound for avgidle. Thus, a negative minidle lets the scheduler remember that a class has recently used more than its allocated bandwidth.

```
Usage: ... cbq bandwidth BPS avpkt BYTES [ mpu BYTES ]
[ cell BYTES ] [ ewma LOG ]
```

bandwidth represents the maximum bandwidth available to the device to which the queue is attached.

avpkt represents the average packet size. This is used in determining the transmission time which is given as Transmission Time t = average packet size / Link Bandwidth

mpu represents the minimum number of bytes that will be sent in a packet. Packets that are of size lesser than mpu are set to mpu. This is done because for ethernet-like interfaces, the minimum packet size is 64. This value is usually set to 64.

cell represents the boundaries of the bytes in the packets that are transmitted. It is used to index into an rtab table, that maintains the packet transmission times for various packet sizes.

CBQ class is automatically generated when a CBQ qdisc created. ??

note: rtab is rate table?

note: mariano: should first declare a cbq "parent" class (which uses all the bandwidth) and then declare the two "leaf" classes.

CBQ is complex qdisc and to be fully understood it is good to read Sally Floyds and Van Jacobsons paper.

3.2 Priority

Simple priority queue

```
Usage: ... prio bands NUMBER priomap P1 P2...
Where:
```

bands number of bands to add (default 3)

priomap define how the priomap looks like (default to 3-band scheduler map)

So if you define more than 3 bands, make sure to re-define the priomap

In prio as long as there is data to be dequeued in the higher priority queue, prio will favor the higher queue.

3.3 FIFO

Simple First-In-First-Out queue which provides basic store-and-forward capability. FIFO is default qdisc on most real interfaces.

```
Usage: ... [p|b]fifo [ limit NUMBER ]
```

"b" stands for bytes, while "p" stands for packets.

limit maximum length of the queue in bytes for bfifo and in packets for pfifo

This means that the maximum length of the fifo queue is measured in bytes in the first case and in number of packets in the second case.

small note: The fifo queue can be set to 0, but this still allows a single packet to be enqueued.

3.4 TBF

Token Bucket Filter is qdisc which have tokens and works like that if there is token in the bucket it possible to enqueue packet and take token. Kernel puts token in the bucket in some intervals

```
Usage: ... tbf limit BYTES burst BYTES[/BYTES] rate KBPS
[ mtu BYTES[/BYTES] ] [ peakrate KBPS ] [ latency TIME ]
```

limit is the number of bytes that can be queued

burst specifies bits per burst how much can be sent within a given unit of time to not create scheduling concerns

rate is used indirectly in qdisc's: that's at tc rate is used to calculate the transmition time required for each packet sized from mpu to mtu. Another definition: rate option is what control bandwidth. AFAIK 'bandwidth' represents the 'real' bandwidth of the device.

mtu is maximum transfer unit

peakrate max short term rate

latency max latency to queuing

Jamal: TBF is influenced by quiet a few parameters; peakrate, rate, MTU, burst size etc. It will do what you ask it to ;-> And at times it will let bursts flood the gate i.e you might end up sending at wire speed. What are your parameters like?

3.5 RED

Random Early Detection discard packet even when there is space in the queue. As the queue length increases drop probability also increases. This approach enables sender to be notified that there is likelihood of congestion before it is actually appeared.

```
Usage: ... red limit BYTES min BYTES max BYTES avpkt BYTES burst PACKETS
           probability PROBABILITY bandwidth KBPS [ ecn ]
```

limit actual physical size of the queue

min minimum threshold in Kilobytes

max maximum threshold in Kilobytes.

avpkt is average packet size

burst is burstiness (from Jamal: used to compute time constant) ???

probability should be random drop probability

bandwidth should be the real bandwidth of the interface

ecn ? explicit congestion notification (flag or what)

Always make sure that min < max < limit

3.6 GRED

Generalized RED is used in DiffServ implementation and it has virtual queue (VQ) within physical queue. Currently, the number of virtual queues is limited to 16.

GRED is configured in two steps. First the generic parameters are configured to select the number of virtual queues DPs and whether to turn on the RIO-like buffer sharing scheme. Also at this point, a default virtual queue is selected.

The second step is used to set parameters for individual virtual queues.

```
Usage: ... gred DP drop-probability limit BYTES min BYTES max BYTES
avpkt BYTES burst PACKETS probability PROBABILITY bandwidth KBPS
[prio value]
OR ... gred setup DPs <num of DPs> default <default DP> [grio]
```

setup identifies that this is a generic setup for GRED

DPs is the number of virtual queues

default specifies default virtual queue

grio turns on the RIO-like buffering scheme

limit defines the virtual queue "physical" limit in bytes

min defines the minimum threshold value in bytes

max defines the maximum threshold value in bytes

avpkt is the average packet size in bytes

bandwidth is the wire-speed of the interface

burst is the number of average-sized packets allowed to burst

probability defines the drop probability in the range (0...)

DP identifies the virtual queue assigned to these parameters

drop-probability ?

prio identifies the virtual queue priority if grio was set in general parameters

3.7 SFQ

Stochastic Fair Queue as it's name implies. It processes queues in round-robin order.

```
Usage: ... sfq [ perturb SECS ] [ quantum BYTES ]
```

perturb is no of seconds after them hashing function will be changed to minimize hash collision to small time interval (the perturb interval).

quantum is DRR (Deficit Round Robin) round quantum like in CBQ.

3.8 ATM

Used to re-direct flows from the default path to ATM VCs. Each flow can have its own ATM VC, but multiple flows can also share the same VC.

Werner: ATM qdisc is different. It takes packets from some traffic stream (no matter what interface or such), and sends it over specific (and typically dedicated) ATM connections.

Werner: Then there's the case of qdiscs that don't really queue data, e.g. sch_dsmark or sch_atm.

3.9 Dsmark

Diff-serv marker isn't really a queuing discipline. It marks packet according to specified rule. It is configured as qdisc first and after that as class (if it is used for classification)

```
Usage: dsmark indices INDICES [ default_index DEFAULT_INDEX ] [ set_tc_index ]
```

indices is the size of the table of (mask,value) pairs. See bellow. (maybe mask ^ value)

default_index is used if the classifier finds no match

set_tc_index if set retrieves the content of the DS field and stores it in skb->tc_index

When invoked to create class it's parameter are:

```
Usage: ... dsmark [ mask MASK ] [ value VALUE ]
```

mask mask on DSCP (default 0xff)

value value to or with (default 0)

Outgoing DSCP = (Incoming DSCP AND mask) OR value

Where Incoming DSCP is the DSCP value of the original incoming packet, and Outgoing DSCP is the DSCP that the packet will be assigned as it leaves the queue.

3.10 INGRESS

if present, the ingress qdisc is invoked for each packet arriving on the respective interface

ingress is a qdisc that only classifies but doesn't queue

the usual classifiers, classifier combinations, and policing functions can be used

the classification result is stored in skb->tc_index, a la sch_dsmark

if the classification returns a "drop" result (TC_POLICE_SHOT), the packet is discarded. Otherwise, it is accepted.

Since there is no queue for implicit rate limiting (via PRIO, TBF, CBQ, etc.), rate limiting must be done explicitly via policing. This is still done exactly like policing on egress.

4 classes

mps: should I explain what is class and their intimacy with qdisc? Yes? Classes are main component of the QoS. (stupid explanation)

The syntax for creating a class is shown below:

```
tc class [ add | del | change | get ] dev STRING
[ classid CLASSID ] [ root | parent CLASSID ]
[ [ QDISC_KIND ] [ help | OPTIONS ] ]
tc class show [ dev STRING ] [ root | parent CLASSID ]
```

Where: QDISC_KIND := { prio | cbq | etc. }

OPTIONS := ... try tc class add <desired QDISC_KIND> help

The QDISC_KIND can be one of the queuing disciplines that support classes. The interpretation of the fields:

classid represents the handle that is assigned to the class by the user. It consists of a major number and a minor number, which have been discussed already.

root indicates that the class represents the root class in the link sharing hierarchy.

parent indicates the handle of the parent of the queuing discipline.

4.1 CBQ

This algorithm classifies the waiting packets into a tree-like hierarchy of classes; the leaves of this tree are in turn scheduled by separate algorithms (called "disciplines" in this context).

```
Usage: ... cbq bandwidth BPS rate BPS maxburst PKTS [ avpkt BYTES ]
[ minburst PKTS ] [ bounded ] [ isolated ]
[ allot BYTES ] [ mpu BYTES ] [ weight RATE ]
[ prio NUMBER ] [ cell BYTES ] [ ewma LOG ]
[ estimator INTERVAL TIME_CONSTANT ]
[ split CLASSID ] [ defmap MASK/CHANGE ]
```

bandwidth represents the maximum bandwidth that is available to the queuing discipline owned by this class. It is only used as helper value to compute min/max idle values from maxburst and avpkt.

rate represents the bandwidth that is allocated to this class. rate should be set to the desired bandwidth (you want) to allocate to a given traffic class. The kernel does not use this directly. It uses pre-calculated rate translation tables. It is used to compute overlimit status of class.

maxburst represents the number of bytes that will be sent in the longest possible burst.

avpkt represents the average number of bytes in a packet belonging to this class.

minburst represents the number of bytes that will be sent in the shortest possible burst.

bounded indicates that the class cannot borrow unused bandwidth from its ancestors. If this is not specified, then the class can borrow unused bandwidth from the parent (default off).

isolated indicates that the class will not share bandwidth with any of non-descendant classes

allot allot is MTU + MAC header

mpu is explained at page 7

weight should be made proportional to the rate.(explain CBQ is implemented using Weighted Round Robin algorithm)

prio represents the priority that is assigned to this class. priority of value 0 is highest (most important) and value 7 is lowest.

cell represents the boundaries of the bytes in the packets that are transmitted. It is used to index into an rtab table, that maintains the packet transmission times for various packet sizes.

ewma is explained at page 6

estimator is explained at page 6

split field is used for fast access. This is normally the root of the CBQ tree. It can be set to any node in the hierarchy thereby enabling the use of a simple and fast classifier, which is configured only for a limited set of keys to point to this node. Only classes with split node set to this node will be matched. The type of service (TOS in the IP header) and sk->priority is not used for this purpose.

defmap say that best effort traffic, not classfied by another means will fall to this class. defmap is bitmap of logical priorities served by this class

A note about CBQ class setup:

cbq class has fifo qdisc attached by default

You *have* to declare first, the CBQ qdisc, then the CBQ "parent" class, and then (optionally, I think), the CBQ "leaf " classes. I'm not 100% sure of what I've just said. It's just how I think it works.

5 filters (or classifier)

Filters are used to classify (map) packets based on certain properties of the packet e.g. TOS byte in the IP header, IP addresses, port numbers etc to certain classes. Queuing disciplines uses filters to assign incoming packets to one of its classes. Filters can be maintained per class or per queuing disciplines based on the design of the queuing discipline. Filters are maintained in filter lists. Filter lists are ordered by priority, in ascending order. Also, the entries are keyed by the protocol for which they apply, e.g., IP, UDP etc. Filters for the same protocol on the same filter list must have different priority values.

Filter vary in the scope

Filters have meters associated with them (TB+rate estimator)

```
Usage: tc filter [ add | del | change | get ] dev STRING
[ pref PRIO ] [ protocol PROTO ]
[ estimator INTERVAL TIME_CONSTANT ]
[ root | classid CLASSID ] [ handle FILTERID ]
[ [ FILTER_TYPE ] [ help | OPTIONS ] ]
```

or

```
tc filter show [ dev STRING ] [ root | parent CLASSID ]
```

Where:

```
FILTER_TYPE := { rsvp | u32 | fw | route | etc. }
FILTERID := ... format depends on classifier, see there
OPTIONS := ... try tc filter add <desired FILTER_KIND> help
```

The interpretation of the fields:

pref represents the priority that is assigned to the filter.

protocol is used by the filter to identify packets belonging only to that protocol. As already mentioned, no two filters can have the same priority and protocol field.

root indicates that the filter is at the root of the link sharing hierarchy.

classid represents the handle of the class to which the filter is applied.

handle represents the handle by which the filter is identified uniquely. The format of the filter is different for different classifiers.

estimator is explained at page 6

5.1 filter rsvp

Use RSVP protocol for classification

```
Usage: ... rsvp ipproto PROTOCOL session DST[/PORT | GPI ]
[ sender SRC[/PORT | GPI ]
[ classid CLASSID ] [ police POLICE_SPEC ]
[ tunnelid ID ] [ tunnel ID skip NUMBER ]
```

```
Where:
GPI := { flowlabel NUMBER | spi/ah SPI | spi/esp SPI |
u{8|16|32} NUMBER mask MASK at OFFSET}
POLICE_SPEC := ... look at TBF
FILTERID := X:Y
```

Comparing to general packet classification problem, RSVP needs only several relatively simple rules:

(dst, protocol) are always specified, so that we are able to hash them.

ipproto is one of the IP protocol (TCP, UDP and maybe other)

session is destination (address?) with or without port, or gpi (Generalized Port Identifier)

src may be exact, or may be wildcard, so that we can keep a hash table plus one wildcard entry.

source port (or flow label) is important only if src is given.

police specification is explained on the page 18, and it should be, but tc gives (with help command) reference to TBF?

rsvp filter is used to distinguish an application session (dst port dst ip address). In an DiffServ edge router it can be used to mark packets of specific applications in order to be classified in the appropriate PHB.

Alexey: IMPLEMENTATION.

We use a two level hash table: The top level is keyed by destination address and protocol ID, every bucket contains a list of "rsvp sessions", identified by destination address, protocol and DPI(="Destination Port ID"): triple (key, mask, offset).

Every bucket has a smaller hash table keyed by source address (cf. RSVP flowspec) and one wildcard entry for wildcard reservations. Every bucket is again a list of "RSVP flows", selected by source address and SPI(="Source Port ID" here rather than "security parameter index"): triple (key, mask, offset).

All the packets with IPv6 extension headers (but AH and ESP) and all fragmented packets go to the best-effort traffic class.

Two "port id"'s seems to be redundant, rfc2207 requires only one "Generalized Port Identifier". So that for classic ah, esp (and udp,tcp) both *pi should coincide or one of them should be wildcard.

At first sight, this redundancy is just a waste of CPU resources. But DPI and SPI add the possibility to assign different priorities to GPIs. Look also at note 4 about tunnels below.

One complication is the case of tunneled packets. We implement it as following: if the first lookup matches a special session with "tunnelhdr" value not zero, flowid doesn't contain the true flow ID, but the tunnel ID (1...255). In this case, we pull tunnelhdr bytes and restart lookup with tunnel ID added to the list of keys. Simple and stupid 8)8) It's enough for PIMREG and IPIP.

Two GPIs make it possible to parse even GRE packets. F.e. DPI can select ETH_P_IP (and necessary flags to make tunnelhdr correct) in GRE protocol field and SPI matches GRE key. Is it not nice? 8)8)

Well, as result, despite its simplicity, we get a pretty powerful classification engine.

Panagiotis Stathopoulos: Well an rsvp filter is used to distinguish an application session (dst port dst ip address). In an DiffServ egde router it can be used to mark packets of specific applications in order to be classified in the appropriate PHB.

note: I have to read more about RSVP

5.2 filter u32

Anything in the header can be used for classification

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.

```
Usage: ... u32 [ match SELECTOR ... ] [ link HTID ] [ classid CLASSID ]
[ police POLICE_SPEC ] [ offset OFFSET_SPEC ]
[ ht HTID ] [ hashkey HASHKEY_SPEC ]
[ sample SAMPLE ]
or u32 divisor DIVISOR
Where: SELECTOR := SAMPLE SAMPLE ...
SAMPLE := { ip | ip6 | udp | tcp | icmp | u{32|16|8} } SAMPLE_ARGS FILTERID := X:Y:Z
```

match 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.

link

classid

police

offset

ht is hash table

hashkey is the key to hash table

sample is protocol such as IP or higher layer protocol such as UDP, TCP or ICMP. sample can be 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.

police specification is explained on the page 18

The syntax here is match ip <item> <value> <mask>

So match ip protocol 6 0xff matches protocol 6, TCP. (See /etc/protocols) match ip dport 0x17 0xffff is TELNET (/etc/services). Note that the number is hexadecimal, not decimal.

note: (mps) ht - hash table HTID Hash Table ID is fh - filter handle in filter show

The filters are packed to hash tables of key nodes with a set of 32bit key/mask pairs at every node. Nodes reference next level hash tables etc.

It seems that it represents the best middle point between speed and manageability both by human and by machine.

It is especially useful for link sharing combined with QoS; pure RSVP doesn't need such a general approach and can use much simpler (and faster) schemes.

5.3 filter fw

Classifier mapping ipchains' fwmark to traffic class

```
Usage: ... fw [ classid CLASSID ] [ police POLICE_SPEC ]
POLICE_SPEC := ... look at TBF
CLASSID := X:Y
```

classid is class handle

police specification is explained on the page 18, and it should be, but tc gives (with help command) reference to TBF?

5.4 filter route

Use routing table decisions for classification

```
Usage: ... route [ from REALM | fromif TAG ] [ to REALM ]
[ flowid CLASSID ] [ police POLICE_SPEC ]
POLICE_SPEC := ... look at TBF
CLASSID := X:Y
```

from REALM is realm in ip route table

fromif TAG is interface tag

to REALM is (again) ip route table realm

flowid CLASSID is class to which packet (if passed) is

police specification is explained on the page 18, and it should be, but tc gives (with help command) reference to TBF?

1. For now we assume that route tags < 256. It allows to use direct table lookups, instead of hash tables.
2. For now we assume that "from TAG" and "fromdev DEV" statements are mutually exclusive.
3. "to TAG from ANY" has higher priority, than "to ANY from XXX"

5.5 tcindex

Use tc_index internal tag in skb to select classes.

```
Usage: ... tcindex [ hash SIZE ] [ mask MASK ] [ shift SHIFT ] [ pass_on | fall_through ] [
```

hash is the size of the lookup table

mask is the bit mask (this explanation is worthless)

shift the mask right by SHIFT number

pass_on defines that this packet will pass

fall_through

classid is the class to which filter is attached

police specification is explained on the page 18

note: key = (skb->tc_index >> shift) & mask

6 police

The purpose of policing is to ensure that traffic does not exceed certain bounds. For simplicity, we will assume a broad definition of policing and consider it to comprise all kinds of traffic control actions that depend in some way on the traffic volume.

We consider four types of policing mechanisms:

- policing decisions by filters
- refusal to enqueue a packet
- dropping of a packet from an "inner" queueing discipline
- dropping of packet when enqueuing a new one

```
Usage: ... police rate BPS burst BYTES[/BYTES] [ mtu BYTES[/BYTES] ]
[ peakrate BPS ] [ avrate BPS ] [ ACTION ]

Where: ACTION := reclassify | drop | continue
```

rate is the long-term rate attached to the meter

peakrate this is the peakrate a flow is allowed to burst in the short-term. Basically this upper-bounds the rate.

mtu a packet exceeding this size will be dropped. The default value is 2KB. This is fine with ethernet whose MTU is 1.5KB but will not be fine with Gigabit ethernet exploiting Jumbo frames for example. It also will not be valid for the lo device whose MTU is defined by amongst other things how much RAM you have. You must set this value if you have exceptions to the rule.

ACTION exceed/non-exceed: This allows to define what actions should be exercised when a flow either exceeds its allocated or doesn't. they are:

pass (?)

reclassify used by CBQ to go to BE (Best Effort, ask Jamal?)

drop simply drops packet

continue - lookup the next filter rule with lower priority

note: "drop" is only recognized by the following qdiscs: atm, cbq, dsmark, and (ingress - really?). In particular, prio ignores it.

References

[1] A. N. Kuznetsov, docs from iproute2

[2] Werner Almesberger, Linux Network Traffic Control – Implementation Overview

[3] Jamal Hadi Salim, IP Quality of Service on Linux `http://????`

Saravanan Radhakrishnan, Linux - Advanced Networking Overview
`http://qos.ittc.ukans.edu/howto/howto.html`

[4] Almesberger, Jamal Hadi Salim, Alexey Kuznetsov - Differentiated Services on Linux

[5] linux-diffserv mailing list `linux-diffserv@lrc.di.epfl.ch`

[6] Sally Floyd, Van Jacobson - Link-sharing and Resource Management Models for Packet Networks

[7] Sally Floyd, Van Jacobson - Random Early Detection Gateways for Congestion Avoidance

[8] Related Cisco documents from `http://www.cisco.com/`

[9] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, Daniel Zapalla - RSVP: A New Resource ReSerVation Protocol

[10] Related RFC's

[11] and many others

Appendix

note: flowid is sometimes class handle sometimes something else

mariano - good setup for me: If you remove the router and then the modem line becomes ppp0 (instead of eth0), you should declare that ppp0 has "bandwidth 30K". Then, the classes should use "bandwidth 30K rate 20K" and "bandwidth 30K rate 10K"