# Table of Contents

# Linux Traffic Control(TC) Subsystem Architecture

## Introduction

The TC architecture can be best described as a Network Service infrastructure. We define a **Network Service** as:

> *The treatment of network packets, as defined by a user policy, so as to achieve a defined goal on selected packets.*

Achieving the `defined goal` is the essence of a network service.

All network services are composed of 1) a discriminator to select the packets and then 2) one or more treatments to act on the selected packets.

An example of a service is simple firewalling with *defined goals* to drop select unwanted packets and drop them or accept allowed packets. The service work flow is:

1. To select bad/good packets and
2. depending on the policy execute an action to drop or accept the packet.

A more complex service example is one that offers differentiated quality of service on interactive voice vs bulk file transfer. The flow of such a service is:

1. discriminate between voice packets and file transfer packets
2. voice is enqueued on a high priority queue and file transer traffic is queued on a low priority queue
3. the high priority queue is serviced ahead of the low priority queue, always.

The important thing to observe from the above examples is that a network service is a policy composition of several *packet processing functions*. We are going to refer to these *packet processing functions* as *Functional Blocks*. The first Functional Block in a graph always serves to discriminate traffic - this is then followed by a series of other FBs; some blocks could constitute simple processing functions like dropping packets or enqueueing packets onto a selected queue; and yet others entail more complex scheduling algorithms for preferentially dequeueing packets out of several available queues before sending out an interface.

Figure1 shows a graph of Functional Blocks that can be used to describe a network service. We will refer to such a composition as a service topology graph.

We introduce a new term, **Functional Block Type(FBT)** as:

> *A template for implementing a Functional Block based on its packet processing goals. In Figure1 A, B, C and D are examples of Functional Block Types.*

Figure1 color-encodes the nodes to represent their FB types: A, B, C, and D. At each node type, a globaly defined *node identifier* is also shown. The node identifier represents an instantiation of the node type along with its configuration attributes. We are going to refer to the node identifier as a **handle** (eg "1" for Node type A, etc).

The vertices connecting the nodes represent packet flow direction; as an example packets could flow from node A1 to either B2 or B3 or D5 depending on a processing decision made at node A1.

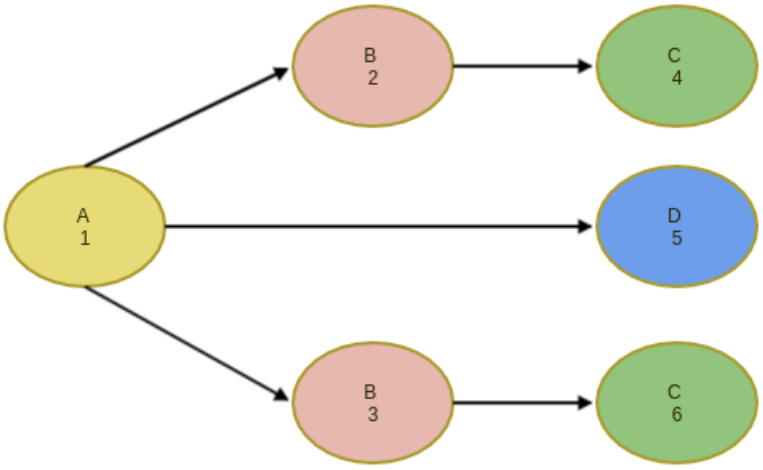


*Figure1: Generic Service Graph*

At this point we are going to introduce two new terms:

1. A **Functional Block Instance** is an instantiation of a Functional Block Type. Each of the nodes in Figure1 is an instance of its respective type and has a unique *handle*.
2. A **Service Topology** is a composition of Functional Block Instances. Figure1 is a Service Topology. Our earlier firewall example also provides a service topology which constitutes a classifier functional block instance, followed by a functional block to drop, or a functional block to accept a packet.

To provision a service policy we create Functional Block Instances and stitch them together to the desired packet flow goals. Let us construct the hypothetical service illustrated in Figure1 to illustrate how such a service topology would be provisioned.

```
1. Create FB type A handle 1 with attributes ....
2. Create FB type B handle 2 bind parent handle 1 with attributes ....
3. Create FB type B handle 3 bind parent handle 1 with attributes ....
4. Create FB type D handle 5 bind parent handle 1 with attributes ....
5. Create FB type C handle 4 bind parent handle 2 with attributes ....
6. Create FB type C handle 6 bind parent handle 3 with attributes ....
```

**Listing1: Sample Service Creation**

Step 1 creates the root of the tree as type A with id 1. Step 2 shows both a *create* and *bind* operation: a node with type B and handle id 2 is created and is then bound/attached as a branch off the node with handle id 1 (which happens to have a FBT A). A1 is the *parent* of B2.

At this point we are going to introduce another new term:

A **Parent ID** is a handle id of the node which the current specified node is branched-from. As an example from the listing above, node with handle 1 is the parent of node with handles 2, 3 and 5.

Steps 3 and 4 show node handle id 3 and 5 bind to the same root node(handle id 1). Steps 5 and 6 show binding to different parent nodes(4 to 2 and 6 to 3).

# TC Overview

The Linux Traffic Control(TC) subsystem consists of 4 FBTs. Each FBT has more than one *kind*. A *kind* is an algorithmic implementation specialization of the FBT. The Functional Block Types are:

1. **Queuing disciplines(qdisc)**. A qdisc provides templating for queue algorithms (enqueuing and dequeuing packets). Some *kinds* of qdiscs include:
    - *Pfifo* which implements a basic packet counting FIFO queueing algorithm.
    - *RED* which implements the Random Early Detection(RED) algorithm.
    - *DRR* which implements the Deficit Round Robin(DRR) algorithm.
2. **Classifiers**. Classifiers provide templates that define filtering algorithms (to discriminate/select packets). Some *kinds* of classifiers include:
    - *u32* which implements a 32-bit ternary key/mask matching algorithm.
    - *flower* which implements a multi-tuple matching algorithm.
    - *fw* which implements a metadata based matching algorithm.
3. **Actions**. Actions provide templating for arbitrary packet processing. Some *kinds* of actions include:
    - *gact* which implements amongst other things dropping and accepting of packets.
    - *mirred* which implements redirecting or mirroring packets.
    - *skbedit* which implements metadata editing on a packet.
4. **Classes**. Classes provide templating for encapsulating qdisc FBTs to allow service topology branching. On their own classes do not implement algorithms; rather class instances serve to proxy access to the internal qdiscs they carry.

These Functional Block Types are used in TC to compose Network Services.

## *TC Functional Block Location And Identity*

As discussed earlier, in order to build a service topology, each node in the topology graph needs two identifiers:

- A *handle* ID.
- Attachment point on the topology graph which we defined as a *parent* ID.

The TC architecture provides 32 bits for each of the IDs. Throughout this document we may interchangeably refer to the *handle* ID as just *handle* and *parent* ID as *parent*.

The starting point to any service topology is an *anchor* location (A tree root). The anchor location is where packets enter a topology.

(footnote?)A network port is a connectivity abstraction which may be physical or virtual. In the Linux world it is often refered to as a network device (or netdev) and in the SNMP world it is sometimes referred to as an interface - throughout this document we will use any of those terms (port, device, netdev, and interface) interchangeably to refer to this abstraction. At the moment, all anchor locations for a TC service topology start at a netdev; therefore the primary coordinate of a service topology must identify a network device.

There are several TC service topologies.

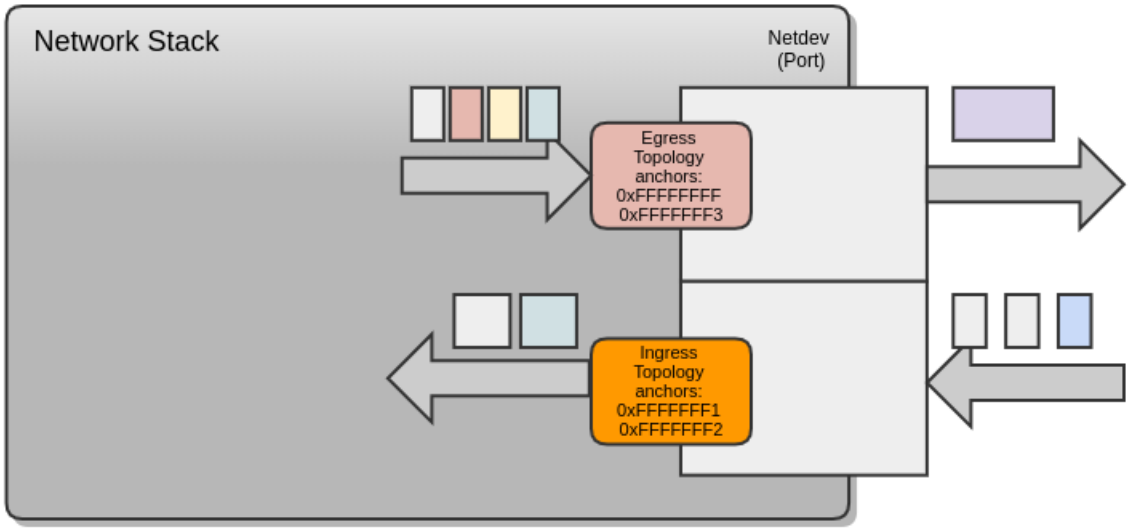


*Figure2: Qdisc Attachment Points*

As illustrated in Figure2, an egress topology is attached at the outgoing point of a netdev. An ingress topology is attached to the incoming point of a netdev.

There are four IDs which are reserved on each netdev for the purpose of anchor location(root/starting point of a service graph) identification:

- An ID of 0xFFFFFFFF is reserved for use as id on the egress anchor point of a netdev as shown in Figure2. We refer to this as the anchor point of the *EGRESS* topology. This location ID is also refered to as **root** in TC nomenclature. The *EGRESS* topology constitutes qdiscs with queues.
- An ID of 0xFFFFFFF3 is reserved for use as id on the egress anchor point on the netdev for the *EGRESSCLSACT* topology. Unlike the *EGRESS* topology, the *EGRESSCLSACT* topology does not have qdiscs with queues.
- An ID of 0xFFFFFFF1 is reserved for use as id on the ingress anchor point of a netdev as shown in Figure2. We refer to this as the anchor point of the *INGRESS* topology. This location ID is also refered to as **ingress** in TC nomenclature.
- An ID of 0xFFFFFFF2 is reserved for use as id on the ingress anchor point of a netdev for the *ingress clsact* topology. This is interchangeably used with **ingress** in TC nomenclature and offers the same semantics as *INGRESS* topology - for that reason we will only be discussing the *INGRESS* topology for the rest of this document.

## *TC Service Topology*

The most fundamental TC FBT is the qdisc.

Packets flow into the ingress of a port towards the network stack or from the network stack egress towards a specific port. A TC service Topology may be provisioned in either of those directions.

Starting with a qdisc, a selection of TC FBTs can be created and linked with control tools to create a network service topology. The root of a TC Service topology tree starts at one of the anchor ports described earlier (shown in Figure2).

The simplest service graph is of one qdisc by itself illustrated by Figure3, anchored on the egress of a port with id 0xFFFFFFFF.


*Figure3: EGRESS Classless Service Topology*

As shown in Figure3 these kinds of qdiscs have queues. Examples of qdiscs that fit this description include Pfifo, RED, etc(more on this later).

More complex service topologies can be built on the egress of a port using qdiscs that are able to make use of classifiers pointing to classes. We refer to such kind of qdiscs as being *classful*. Examples of such qdiscs includes DRR, HTB, *Prio* etc. Figure4 shows an example service topology which fits this description.

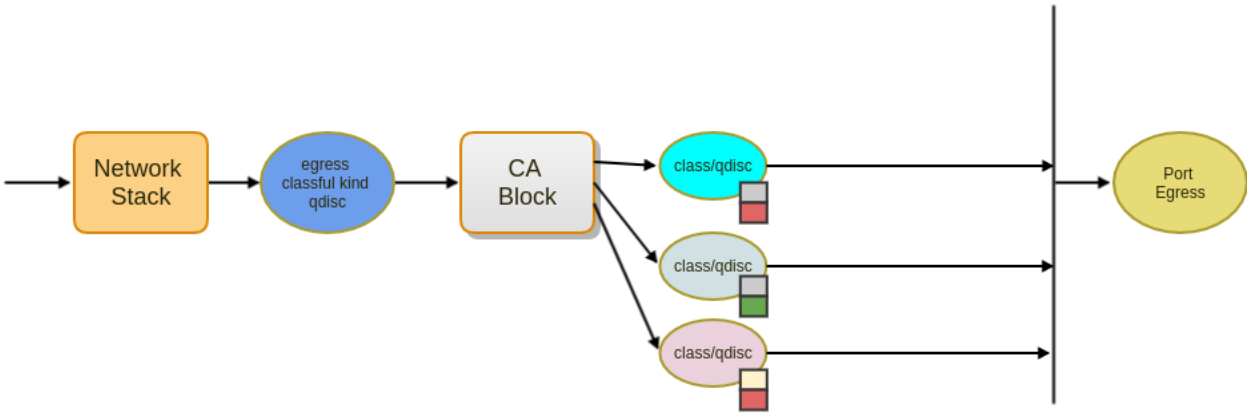


*Figure4: EGRESS Classful Service Topology*

As can be observed from Figure4 all the leaf qdiscs(encapsulated within their respective classes) on the service topology have queues.

*Classful* qdiscs can lead to other qdiscs as show in Figure5. With such a multi-level hierarchical qdisc setup, each hierarchy level requires demultiplexing to a selected class via a Classifier-Action (CA) block. It should also be noted that, like in Figure4, all leaf qdiscs have queues and qdiscs preceeding CA blocks have no embeded queues. We are going call qdiscs preceeding CA blocks as *branch* qdiscs.

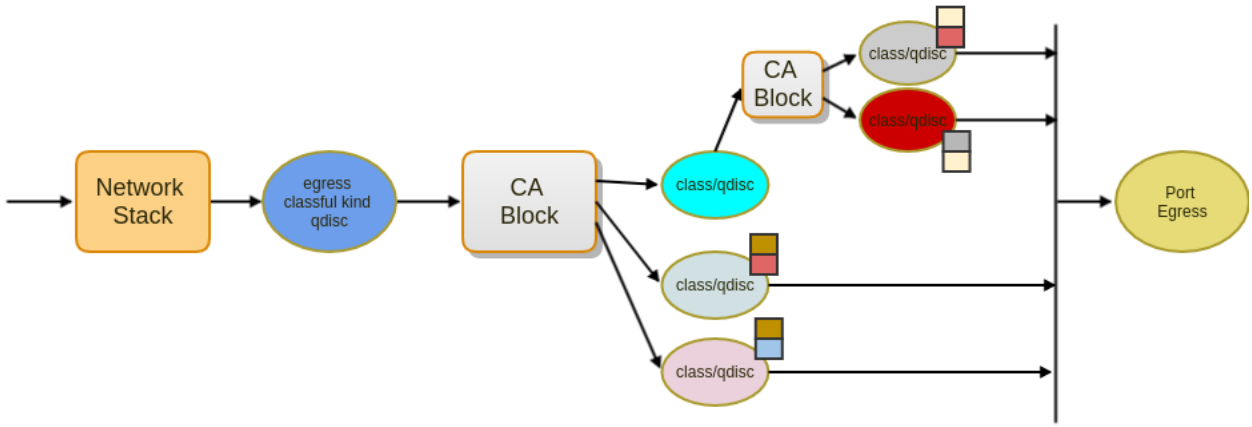


*Figure5: EGRESS Complex Classful Service Topology*

There are many use cases where an egress service can be accomplished without the need for a qdisc with queues; in such a case, the branch qdiscs act as a point of demuxing to actions. To meet these egress service requirements, a qdisc kind *clsact* can be anchored at the egress of a port (with id 0xFFFFFFF3) and would serve only to hold CA block. Figure6 shows such a setup.

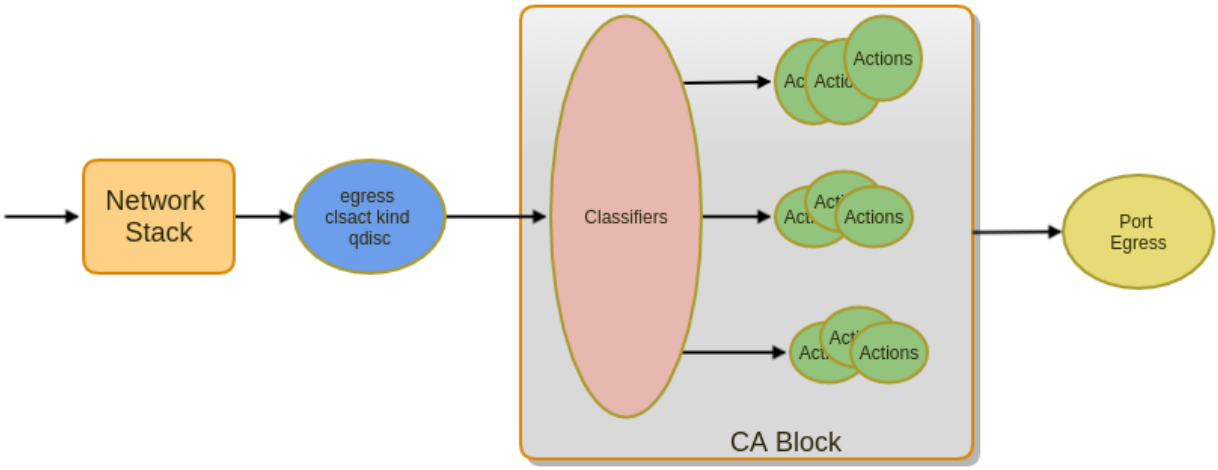


*Figure6: EGRESS Clsact Service Topology*

In Figure6 we zoom in on the CA block and show how classifiers would multiplex to a graph of actions.

Such a queue-less requirement is typically desired for an ingress service. Like the *clsact* qdisc kind, the *Ingress* qdisc kind does not own any queues. The *Ingress* qdisc kind is attached to the ingress of a port as shown in Figure7.

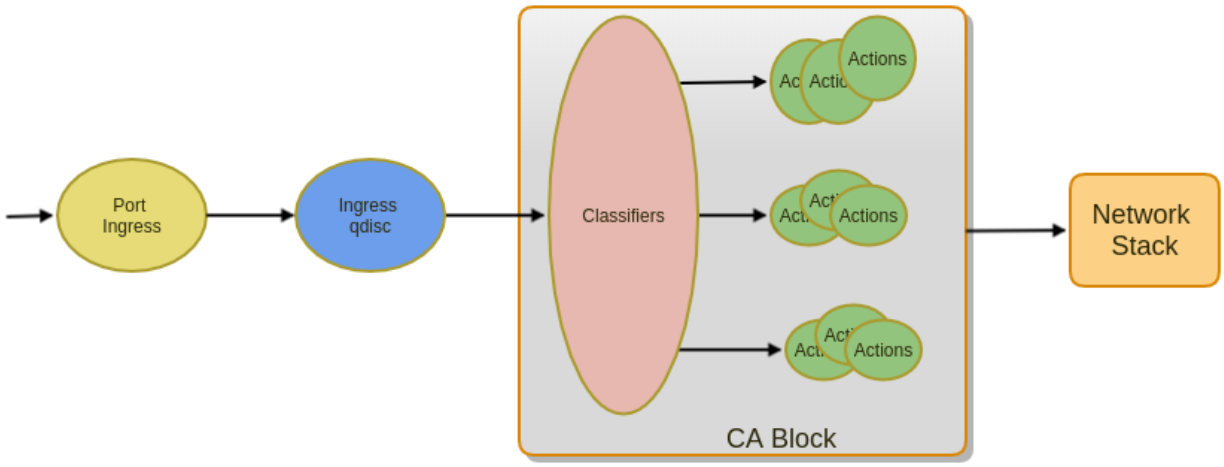


*Figure7: INGRESS Service Topology*

Figure3, Figure4, Figure5, Figure6 and Figure7 illustrate templates for possible service topologies. It should be emphasized that an instantiation of a service topology could use any of these templates and depending on parametrization of the different functional blocks will result in unique services. Infact it is possible to combine several of such topologies(ingress to egress or vice-versa) and to connect them together via the CA block - something we are going to discuss later in the document.

To formalize the 3 possible topologies discussed so far, we try to define a BNF grammar in Listing2 to describe the different constraints that need to be accounted for:

- TOPOLOGY := INGRESS | EGRESSCLSACT | EGRESS
- INGRESS := `Ingress` CA
- EGRESSCLSACT := `clsact` CA
- EGRESS := QDISC-cf | `QDISC`
- QDISC-cf := ( CA CLASS+ )
- CLASS := QDISC-cf | `QDISC`
- CA := ( `CLASSIFIER ACTION+` ) +

**Listing2: BNF Grammar**

Notes on the BNF:

- `Ingress` is a qdisc of kind "ingress"
- `clsact` is a qdisc of kind "clsact"
- QDISC-cf is a classful qdisc
- `QDISC` is a qdisc with single queue

A Service Topology **must** have a qdisc which may optionally have queues. The grammar shows three possible topologies: INGRESS, EGRESSCLSACT, EGRESS. Neither INGRESS nor EGRESSCLSACT topologies have qdiscs with queues. INGRESS, as mentioned earlier, can only have a qdisc kind called *Ingress* (which is queue-less); and EGRESSCLSACT can only have the qdisc kind called *clsact*. The EGRESS topology always has queues associated with it.

A topology may have **optional** CA(Classifier-Action) blocks following the qdisc. Both EGRESSCLSACT and INGRESS topologies always have CA blocks; on the other hand, EGRESS topology will only have CA blocks if there is a presence of *classful* qdiscs(QDISC-cf). A classful qdisc is composed of a CA block followed by TC classes. A class encapsulates a qdisc which may be classful(`QDISC-cf`) or classless (`QDISC`). A classless qdisc is a terminal/leaf in a service tree graph; whereas a class is a branching point in a service tree graph.

Classifiers are always followed by actions. Classifiers lookup either packet data or metadata and decide on action graph branch to take. The graph could constitute one or more actions that follow the classifier. An action could be a simple one such as to select a class(in the EGRESS topology) or drop a packet etc; but could also be complex and extend the service graph further by extending into new classifiers which will lead to another action sub-graph. We will discuss this further when we get to the details of actions.

The INGRESS topology is anchored at the ingress of a network port and, as described earlier, has at its root an *ingress* qdisc kind followed by a CA block. The *ingress* qdisc kind does not have any queues.

The EGRESSCLSACT topology is anchored at the egress of a network port and has at its root a *clsact* qdisc kind followed by a CA block. The *clsact* qdisc kind does not have any queues.

The EGRESS topology is also anchored at the egress of a network port (like EGRESSCLSACT) but constitutes qdiscs that have queues. Classful qdisc kinds lead to classes - allowing a complex topology to be created. Classes allow for adding branches to the topology. The branch is selected by a classifier within the topology. Classes have embedded qdiscs which may have *optional* provisioned CA blocks (if the qdisc is classful). The CA block further demultiplex to another class in the provisioned topology - which begins a new topology sub-hierarchy. This pattern can be repeated ad-infinitum providing a fractal-like object layout until a terminal non-classful qdisc is reached.

For each of the 3 different topology types, packets arrive at a qdisc at th root of the provisioned graph hierarchy and traverses the topology downstream - consulting CA blocks at each branching qdisc to decide how to proceed onto the graph.

# Document organization

In the rest of the document we will start discussing qdiscs first and then progress towards other FB types(Classes, classifiers/filters and Actions). For each FB type we will describe both the topology location semantics and object identity and how they fit within a service topology.

XXX: If we can add section numbers(how do you do it with RST?) then this is the point where we say "in section 3 we will talk about .. and in section 4 .." etc.

## Queueing Disciplines

A queueing discipline(qdisc) is a functional block abstraction centred around the concept of a queue. Two central qdisc functions are the *enqueue()* and *dequeue()* operations; each qdisc *kind* overrides these two methods with its own specific algorithm to achieve its objectives. Example:

- The PFIFO qdisc will queue packets at the tail of an associated queue whereas an active queue management algorithm like RED will compute if it should queue the packet, drop it or re-mark it.
- The ingress qdisc does not have a queue and uses the enqueue() method to invoke the CA block.
- The PFIFO qdisc will dequeue from the head of the queue whereas a shaping qdisc (such as TBF) will first compute if the consumed bandwith associated with the queue is still within the range of configured value and if it is it will dequeue a packet or refuse to when exceeded.

A qdisc is created by user control (using the iproute2 *tc* config utility) with the following parameters:

- Device/netdev e.g. "eth0" etc
- *Parent* ID - note that the parent could be a netdev anchor point (with id 0xFFFFFFFF or 0xFFFFFFF1 etc) or another qdisc's handle when branching on *classful* qdiscs.
- *Handle* ID

To summarize: a qdisc is uniquely defined with the tuples *{ifindex of netdev, parent id, handle id}* with the first two coordinates defining a topology location and the last one defining identity.

Handle IDs are typically described with a **major:minor** numbering scheme, each 16 bit. A qdisc ID always has a minor value of 0. Later on when we describe classes, we will see how the minor number is used.

Figure8 illustrates a simple (non-classful) qdisc.

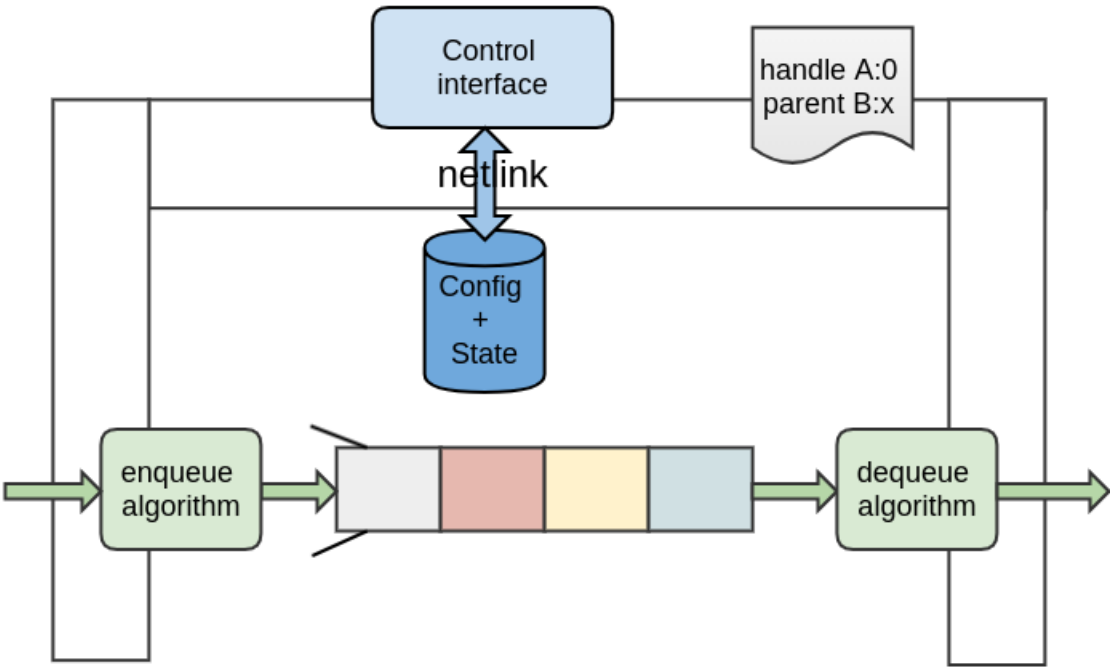


*Figure8: Basic Qdisc Setup*

As shown, the qdisc has a handle (A:0) and a parent (B:x). The qdisc could be anchored to the egress of a device directly (Parent id being FFFF:FFFF or FFFF:FFF3) or maybe in a class in the hierachy (and a parent could be of the form 1:0).

Another detail to pay attention to in Figure8 is the fact that user space has access to configuration and state of the qdisc instance via a netlink control interface(more on this later). The *tc* utility (part of iproute2), which uses the netlink interface to talk to the kernel, will be used through out this document to demonstrate access to state and config. To avoid ambiguity, going forward in this document, when we mention *tc* (in lower case) the reference is going to be explicit to the iproute2 utility; when we mention TC (upper case variant) we would be explicitly refering to the Linux Traffic Control architecture.

And last but not least, note that Figure8 highlights both the *enqueue* and *dequeue* operations for the qdisc instance. Figure8 shows presence of a queue of packets accessible to both operations. Depending on the qdisc kind, a queue of packets may not exist at all. As an example:

- the blackhole qdisc *enqueue()* method just drops any packets handed to it (and has no real queues).
- the ingress qdisc *enqueue()* is used for holding the classifier-action block and has no queue whatsoever.

For illustration, let us show an example of creating a basic egress topology service using a qdisc of kind `pfifo` to the egress of eth0. We will constrain the pfifo qdisc to have a maximum queue size to 1000 packets. Note: as discussed earlier the term *root* in *tc* nomenclature implies the anchor point FFFF:FFFF (Figure2)

```
sudo tc qdisc add dev eth0 parent root handle 1:0 pfifo limit 1000
```

**Listing3: Sample Egress Service Creation**

Essentially the configuration has instantiated a qdisc kind `pfifo` with coordinates *{eth0, root, 1:0}*. Furthermore, the instantiation has provided the *limit* attribute used by Pfifo qdisc *kind* to describe the maximum number of packets(1000) that can be queued onto the encapsulated queue.

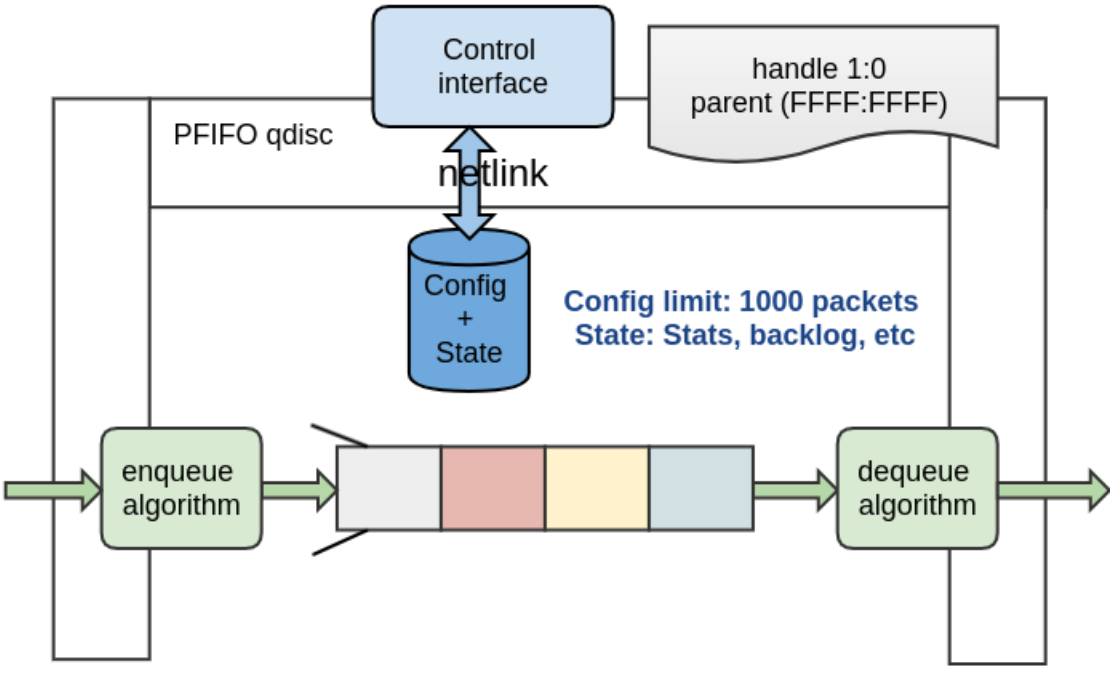

*Figure9: Pfifo Qdisc*

Figure9 visualizes the configured service topology.

For completion, lets use the netlink interface to query the configuration and state of the instantiated qdisc.

```
$  tc -s qdisc ls dev eth0
    qdisc pfifo 1: root refcnt 2 limit 1000p
     Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
     backlog 0b 0p requeues 0
```

**Listing4: Egress Service Config And State**

The results also show statistics associated with packets passing through this qdisc.

# Classful Queueing Disciplines

Egress service topologies can have hierachies as described in the the grammar in Listing2 as well as illustrated in Figure4 and Figure5. The branching feature is facilitated by qdiscs that are refered to as `classful`. A classifier adjacent to the classful qdisc is used select a class (encapsulating a qdisc).

## *Classifier Types For Selecting Classes*

There are two types of classifiers in the TC infrastructure: `built-in` and `explicit`.

1. Built-in classifiers are omni-present adjacent to classful qdiscs. These classifier kinds use packet metadata for their selection criteria. The most common metadata inspected is the *skb->priority*; however, there are qdiscs which stare at the *skb->mark* for the demux decision. Built-in classifier require no provisioning.
2. Explicit classifiers look at either or both of packet data and metadata. We will encounter a few examples of such classifiers(*u32*, *flower*, etc) in later sections. Explicit classifiers require user control provisioning.

## *Class Location And Identity*

An instantiated TC *Class* object is essentially a "wrapper" of a qdisc object. The sole purpose of a *Class* is to provide a way for the egress topology to bind qdisc objects (and thus enabling hierachical topology).

Just like qdiscs, TC classes have coordinates constituting a topology location and identity. The coordinate addressing is the same as that used by a qdisc: *{ifindex of netdev, parent id, handle ID}*. The ifindex and parentid identify a location in the service topology and the handle ID provides the identity of the class. The parentid of a class is always a qdisc and the class handle id is from the same ID space as the qdiscs - with a minor exception: Class IDs make use of the minor number. Going forward we will interchangeably refer to Class handle IDs as ClassIDs.

As mentioned a few times now, Class selection is achieved by way of a *classifier* within a topology. A classifier selects a *TC Class* (the similarity in the naming of these two TC components should give away their relationship ;->).

Figure10 shows a sample configuration.

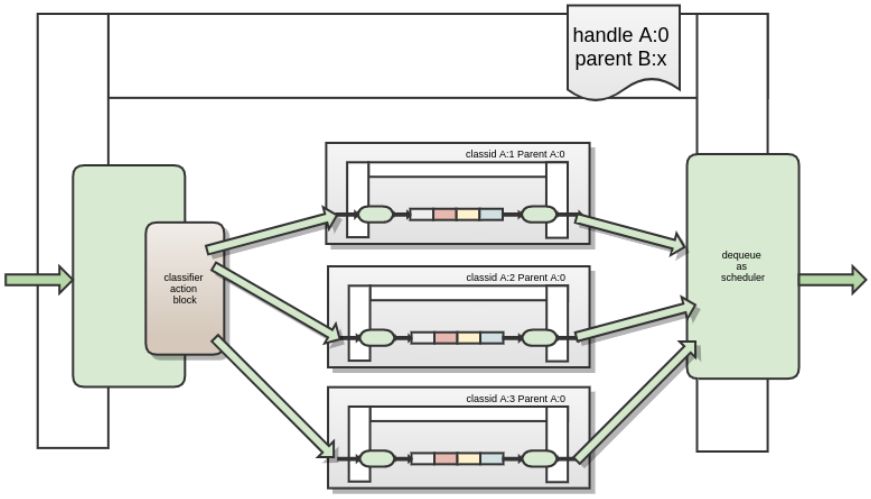


*Figure10 Classful Qdiscs*

When a packet arrives at the topmost qdisc(handle A:0) *enqueue* method, the attached classifier uses either/or both packet data and/or metadata to select a class. The selected class' encapsulated (inner) qdisc's *enqueue* method is then invoked and handed the packet data and metadata. The encapsulated (inner) qdisc will apply its own algorithm (and the pattern repeats if the inner qdisc is classful and has classes).

The *dequeue* method implementation of a classful qdisc is typically a queue scheduling algorithm. Such an algorithm decides which queue to use as its source of a packet to send out of the netdev.

For illustration of classful qdiscs, lets instantiate a *prio* qdisc and attach it to the egress of eth0.

```
$sudo tc qdisc add dev eth0 root handle 1: prio
```

**Listing5: Classful Egress Service**

Figure11 demonstrates the setup.

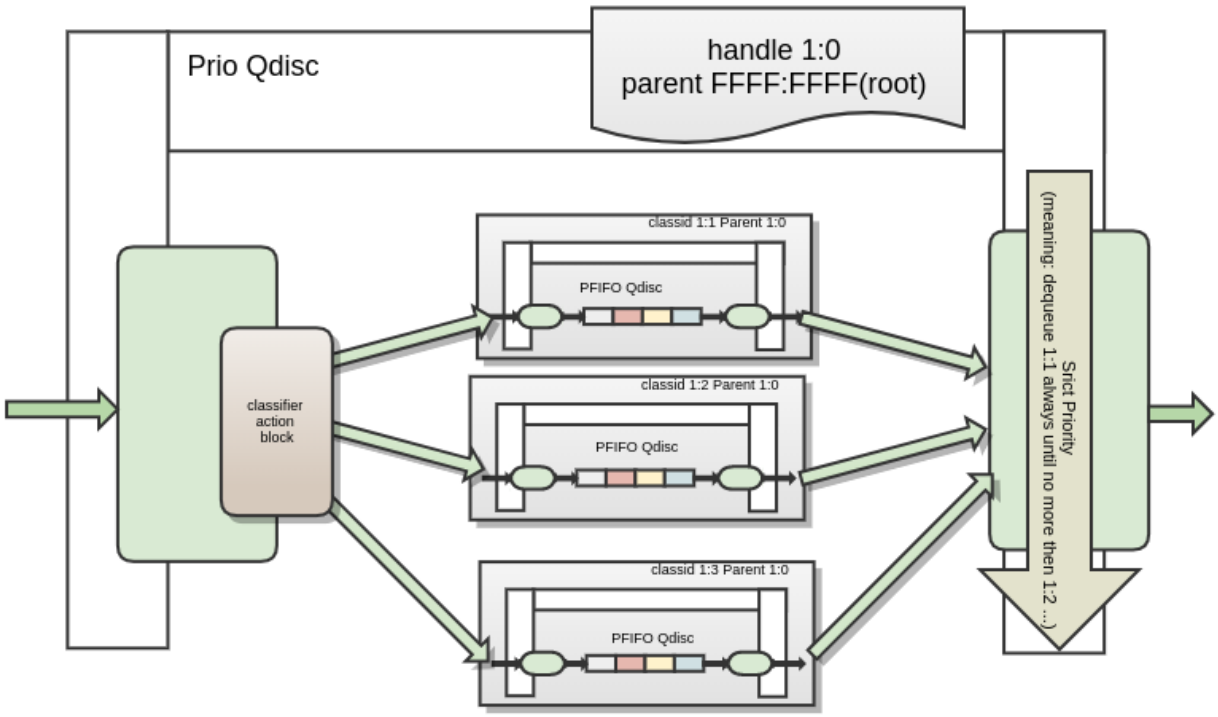


*Figure11: Prio Qdisc*

Essentially the configuration in Listing5 instantiates a qdisc of kind *Prio* with coordinates *{eth0, root, 1:0}*. When a *Prio* qdisc is instantiated, with no additional attributes specified, it creates 3 default classes - each encapsulating a *pfifo* qdisc; this can be changed by overriding the creation params; we leave this up to the reader to figure out (the reader is asked to review *Prio* qdisc documentation in the iproute2 package).

A *Prio* qdisc by default uses a built-in classifier - which uses the skb->priority metadata to select a class to which it enqueues to. To understand this let us query the configuration of the qdisc via netlink(using *tc*):

```
$sudo tc qdisc ls dev eth0
  qdisc prio 1: root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

**Listing6: Classful Egress Service Config View**

The keywords "bands" (set to 3) and "priomap" define the scheduling approach taken. Bands are essentially queues - we see 3. Priority metadata to class selection is programmed in the "priomap".

Lets display the classes associated with this qdisc using *tc*.

```
$sudo tc class ls dev eth0
 class prio 1:1 parent 1:
 class prio 1:2 parent 1:
 class prio 1:3 parent 1:
```

**Listing7: Classful Egress Service Class View**

A *Prio* qdisc *dequeue* method implements a *strict priority* scheduler. Queue 0 - which is referred to as "band 0" in *Prio* qdisc documentation is mapped to classid 1:1 in Figure11; band 1 is mapped to classid 1:2 and band 2 is mapped to classid 1:3. Classid 1:1 is considered the highest priority queue - always dequeued from when it has packets. Only when classid 1:1 has no more packets will the other classes will be asked for packets by the scheduler using a strict priority, classid 1:2 next and so on.

***Non-Homogenous qdisc Topology***

The TC Egress topology allows for different qdisc kinds to be placed in the different graph nodes. It is therefore possible to build complex network services (XXX: Assured Forwarding or Expedited Forwarding Service etc as examples?).

To demonstrate the feature we build upon the already instantiated *prio* qdisc from Figure11 and replace its existing class default pfifo qdiscs in each of the 3 classes with other qdiscs.

```
$ sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf limit 2000 rate 6mbit burst 15k
$ sudo tc qdisc add dev eth0 parent 1:2 handle 20: pfifo limit 500
$ sudo tc qdisc add dev eth0 parent 1:3 handle 30: sfq limit 255
```

**Listing8: Non-Homegenous Egress Service Creation**

The above configuration replaces the qdisc in class 1:1 with a *TBF* qdisc; packets coming out of class 1:1 will be shaped to 6 Mbps. Class 1:2 is replaced with a *pfifo* qdisc with a maximum queue size of 500 packets(replacing the default 1000) and class 1:3 qdisc with Stochastic Fair Queueing(*SFQ*) limited to 255 packets.

To better appreaciate the egress topology, for the rest of this document we are going to switch to a slightly different visualization so we can focus more on illustrating the hierarchical abstraction. Figure12 uses this new illustration approach to visualize the setup derived from Figure11 that is described above.

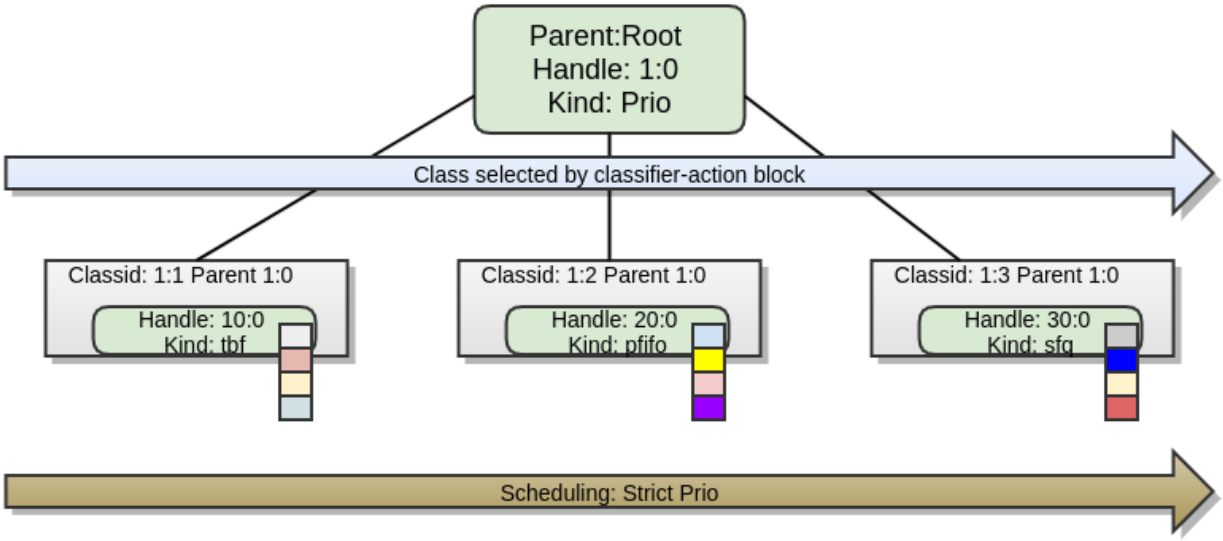


*Figure12: Alternate Prio Qdisc*

Lets query the qdisc setup we created for state and config:

```
$sudo tc -s qdisc ls dev eth0
 qdisc prio 1: 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)
  backlog 0b 0p requeues 0
 qdisc sfq 30: parent 1:3 limit 255p quantum 1514b depth 127 divisor 1024
  Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
  backlog 0b 0p requeues 0
 qdisc tbf 10: parent 1:1 rate 6000Kbit burst 15Kb lat 4294.9s
  Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
  backlog 0b 0p requeues 0
 qdisc pfifo 20: parent 1:2 limit 500p
  Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
  backlog 0b 0p requeues 0
```

**Listing9: Non-Homegenous Egress Service Config And State**

Observe the config of the different qdiscs in Listing9 - and note that the kernel will provide some good defaults for the different attributes which we did not specify for each of the qdisc creations we provisioned in Listing8 (example *sfq* creates 1024 hash buckets etc). Each of these attributes can, of course, be user specified.

Lets query the classes:

```
$sudo tc -s class ls dev eth0
class prio 1:1 parent 1: leaf 10:
 Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
class prio 1:2 parent 1: leaf 20:
 Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
```

```
 backlog 0b 0p requeues 0
class prio 1:3 parent 1: leaf 30:
 Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
class tbf 10:1 parent 10:
```

**Listing10: Non-Homegenous Egress Service Class Config And State**

It should be noted that the TBF qdisc also spawns one class: classid 10:1 (just like the *Prio* qdisc did). This is a consequence of how the kernel sets TBF - although it is still a single class it provides opportunity to replace the embedded qdisc with some other qdisc kind (example, we could replace the *pfifo* with a *Prio* qdisc).

#### *Selecting Classes Using Built-In Classifier*

As was discussed earlier, classifiers can be built-in or explicit. In this section we discuss built-in classifiers. There are two approaches to built-in classifiers: 1) direct mapping and 2) qos mapping.

Each qdisc kind may approach the built-in classifier differently in terms of which metadata to use or in terms of which to implement or when implementing both how to prioritize one approach over the other. We will describe the *prio* qdisc approach since it encompasses both built-in approaches using the `skb->priority` metadata.

As mentioned earlier, the *Prio* qdisc has a built-in classifier which looks at the priority metadata field(`skb->priority`) and selects one of the classes/queues based on the metadata value. The packet's priority metadata value could be set by different parts of the kernel before an egress prio qdisc sees it; for example: the Vlan code transfers the 802.1p value to it; the IPV4 routing code copies the TOS fields to this same field (XXX: Is it TOS still?); a user could (running a server/client) set this value with a socketsetopt() call; and a CA block at any topology (ingress etc) could set it via the skbedit action.

When the *Prio* qdisc *enqueue()* method is invoked, it uses the priority metadata field value to index into the priomap array shown in the configuration earlier (*1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1*). We will call this the `qos mapping` approach for priority classification. There are 16 possible priority values(0-15) in the above default priomap. When a value of zero is seen on the packet, the prio entry at priomap[0] (value = 1) is found. The value 1 maps to class 1:2. When prio is 2, index 2(priomap[2]) is looked up and the resulting value of 2 is mapped to 1:3, etc. When a priority goes outside the acceptable range(0-15 in the above priomap), the index 0 is used to select the class (classid 1:2).

There is another way the priority metadata is used to classify; to directly map the metadata to a classid. We will call this the `direct mapping` approach. Example, a socket user or skbedit action could set the value to 0x00010003 which is equivalent to 1:3. When the *Prio* qdisc *enqueue()* is provided such a value it selects class 1:3 directly. Almost all the classful qdiscs (ATM, CBQ, DRR etc) support this direct mapping classification method.

## *Dequeueing From Qdiscs*

As mentioned earlier, the *dequeue()* method is used by different qdisc kinds to implement algorithms that suits that qdisc kinds's algorithm end goal. In the case of a classful qdisc, the associated dequeue algorithm typically defines scheduling of the enqueued packets; so as to make decisions when those packets depart the qdisc (if they do at all).

For the sake of simplicity in explaining we are going to describe the workflow for dequeuing as something initiated by the kernel. The kernel starts by invoking the topology root qdisc *dequeue()* method. The root in turn invokes its downstream class/qdisc dequeue() methods. The downstream qdisc dequeue(), depending on its algorithm, may or may not return a packet even if one was available on its queue. As an example, a

shaping qdisc like TBF will not return a packet when its dequeue method is invoked if the shaping rate is exceeded.

We are going to use the *prio* qdisc to illustrate classful dequeueing.

The *prio* qdisc implements a strict `work conserving priority scheduling algorithm`. The highest priority queue(classid 1:1) is serviced until it is empty, then classid 1:2 is dequeued from until it is empty(as long as no new packet shows up on classid 1:1, in which case dequeuing from classid 1:1 takes precedence); and last classid 1:3 is serviced (unless classid 1:2 or 1:1 have packets in which case servicing those queues takes precedence). As can be observed this means that in the default setup, classid 1:1 can starve out classid 1:2 and 1:3, and likewise classid 1:2 can starve 1:3. *This is the design intent of the Prio scheduler*.

In the default *prio* setup we have shown so far in Figure11, all the classes encapsulate *pfifo* qdiscs. The user could choose to change the pfifo qdisc to alleviate this effect. Infact the example in Figure12 does just this by replacing the default *pfifo* qdisc with *TBF* which rate limits dequeueing of classid 1:1 to 6Mbps. When the classid 1:1 dequeue() is invoked by parent qdisc(1:0) dequeue() it will always return a packet - unless the rate of 6Mbps is exceeded; otherwise the behavior is as if that queue is empty, in which case the Prio qdisc scheduling gives a chance to classid 1:2 to send something.

## More Complex Egress Topologies

Let us build a topology with two levels of hierarchies to demonstrate something more complex:

```
$ sudo tc qdisc add dev eth0 parent root handle 1: drr
$ sudo tc class add dev eth0 parent 1: classid 1:1 drr
$ sudo tc class add dev eth0 parent 1: classid 1:2 drr
$ sudo tc class add dev eth0 parent 1: classid 1:3 drr
$ sudo tc qdisc add dev eth0 parent 1:2 handle 20: prio
```

**Listing11: More Complex Egress Topology Creation**

At the top of the hierarchy is a Deficit Round Robin (*DRR*) qdisc with three classes. DRR class 1:2 extends the topology based on the *prio* qdisc. The provisioned setup is visualized in Figure13.

Lets see the qdisc config.

```
$ sudo tc qdisc ls dev eth0
  qdisc drr 1: root refcnt 2
  qdisc prio 20: parent 1:2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

**Listing12: More Complex Egress Topology Config**

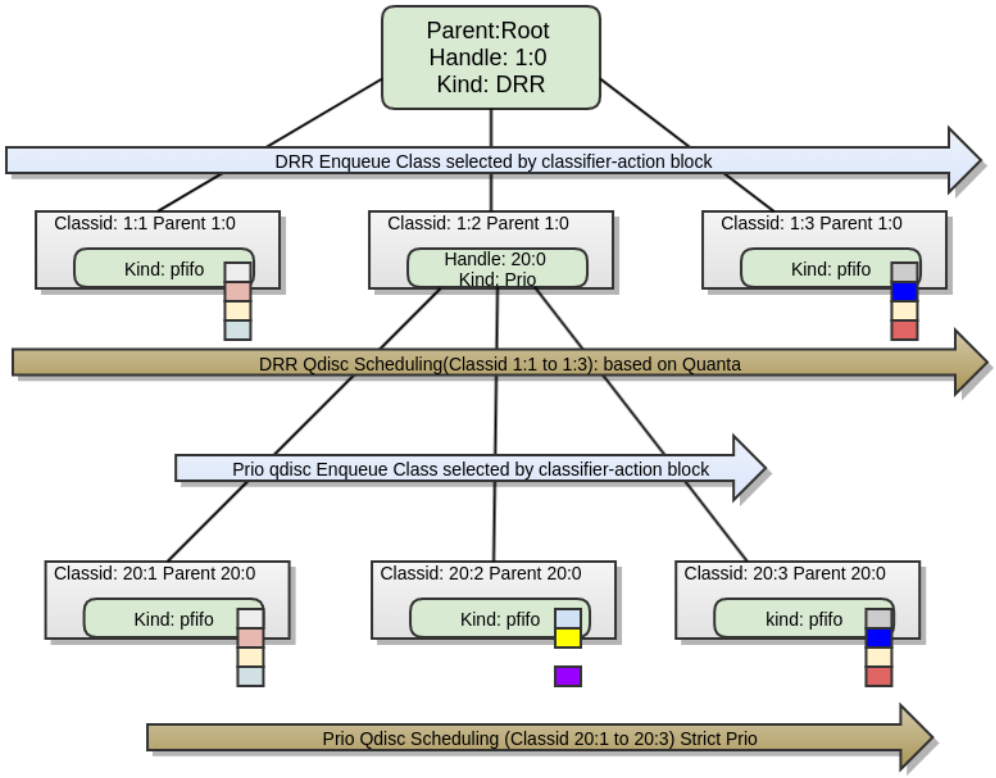


*Figure13: More Complex Hierarchy*

Lets see the class configuration.

```
$ sudo tc class ls dev eth0
class drr 1:1 root quantum 1514b
class drr 1:2 root leaf 20: quantum 1514b
class drr 1:3 root quantum 1514b
class prio 20:1 parent 20:
class prio 20:2 parent 20:
class prio 20:3 parent 20:
```

**Listing13: More Complex Egress Class Topology Config**

DRR by default uses the built-in priority metadata classifier to select classes on which packets are to be enqueued to. On the dequeue phase, DRR gives an opportunity for each queue to supply upto a quantum of bytes (1514 in the examples in Listing13.

## *Enqueuing On Complex Egress Topologies*

Let us demonstrate how a packet would traverse the topology depicted in Figure13 for the purpose of enqueueing.

Let us assume that the packet arrives at the root *DRR* qdisc with `skb->priority` equal to 0x00010002. The sequence of events is as follows:

1. First the root(*DRR*) enqueue() method is invoked.
2. The `enqueue()` method consults built-in priority classifier - it uses the `skb->priority` as input to `direct mapping` classification. Class 1:2 matches. This class happens to encapsulate a *prio* qdisc with id 20:0.
3. The *Prio* qdisc 20:0 `enqueue()` method is invoked and consults its built-in classifier
    - `direct mapping` classification will fail to find a match because qdisc 20:0 does not have a child class with id 1:2;
    - `qos mapping` classification is consulted next and will fail since the value 0x00010002 is above the maximum priority value of 15. The *prio* qdisc in this case will assume a priority value of zero as the default. `qos mapping` classification is used indexing into the priomap array index 0 to map to class 20:2.
4. The packet is enqueued on the queue encapsulated within class 20:2.

## *Dequeuing From Complex Egress Topologies*

Let us demonstrate how a packet would be dequeued from the topology depicted in Figure13.

1. The root *DRR* `dequeue()` method is invoked
2. The root `dequeue()` method invokes classes 1:1 to 1:3 in that order. For each of the classes:
    - Invoke encapsulated `dequeue()` methods(*pfifo* for class 1:1 and 1:3 and *prio* for class 1:2).
        - Each of the child classes is allowed to provide up to a quanta size (initally shown as 1514 bytes in the config above). The class is requested to provide data for as long as the sum of the bytes for that request round does not exceeded the available quanta.

- If the class is not able to provide the full quanta amount of bytes, then whatever the difference is gets added to its quota for the next time. Example if a class was only able to provide 514 bytes when it had a quanta of 1514 bytes, the next time it will be allowed to send upto 2514 bytes.
- If the class had more data than the quanta allows, then it will miss its opportunity to transmit but will get a better opportunity the next time. As an example, lets say the class had only 514 bytes quanta available and had a 1000 bytes packet to send, it will not be allowed to send at that moment but will get its quanta incremented by 1514 bytes for the next round (for a total of 2028 bytes).

Note: Class 1:2 being hierarchical is treated by *DRR* the same as the other 2 classes i.e. it will be allowed to send 1514 bytes. Given *prio* is in charge of class 1:2 then that class is subject to *prio* algorithmn when it gets requested to dequeue a packet. It is up to the *prio* qdisc to pick which of its associated classes to dequeue from: and as we have seen it will dequeue from class 20:1 always unless class 20:1 has no more packets etc. *DRR* is totally oblivious to this - all it is requesting from the prio qdisc is to be provided upto a quanta amount of bytes.

## Classifying Classes Using Explicit Classifiers

As already mentioned classifiers are one of the key functional block types in the TC architecture. Their role is to look at input packet data and/or metadata and decide what treatment a packet gets. So far we have looked at the built-in classifier which operates on the *skb->priority* metadata. We are now going to introduce the second kind of classifiers that we are going to refer to as `explicit classifiers`. Explicit classifiers are user provisioned. There are many kinds of explicit classifiers (just like there are many kinds of qdiscs) all of which are user plug-able into the service topology. The underlying architectural decision to allow different algorithms is based on the fact that **it is impossible to have one classifier algorithm that is optimal for all kinds of packet meta/data classification**. Some examples:

- A match on a string using the latest string search algorithm is most likely not the most fit algorithm for IPv6 header classification.
- Even for the same type of field/tuple classification(eg looking up destination IP addresses) some algorithms are better suited for specific environments than others; example small systems with few rules care about RAM cost for policy storage compared to a large system where lookup speed is more important than cost of storage, etc.

Given that classifier algorithms are a constant source of industry advancements and academic curiosity, the idea that a new classifier could be authored and easily plugged-in makes the TC architecture very flexible.

There are many kinds of classifiers in the TC subsystem. Here are a few that work on packet data :

- *u32* which matches based on 32-bit ternary key/mask chunks on arbitrary packet offsets. U32 is also capable of looking up rules based on *skb->mark* metadata in addition to the packet data.
- *flower* which matches based on many pre-defined header tuples (src/dst ipv4/6, ip protocol etc) to match on packets. On the egress side it makes use of the packet flow information already constructed by the network stack to speed up its lookups.
- *bpf* matches via arbitrary EBPF programs logic on packets. It is up to the EBPF program author to decide on the algorithm.

And here are a few samples that are metadata based:

- *cgroup* which matches on control groups(cgroups) metadata assignment on packets.
- *fw* which matches on *skb->mark* metadata
- *route* which matches based on *route realm* metadata added by the routing subsystem

User control can create an instance of a classifier referred to as a *filter*. Filters of different classifier kinds can be grouped together into a list known as a *chain*. To resolve ambiguity in a chain, filters are ordered by ascending preference; i.e when more than one filter rule matches a packet, the highest priority filter is chosen as the least ambigiuos.

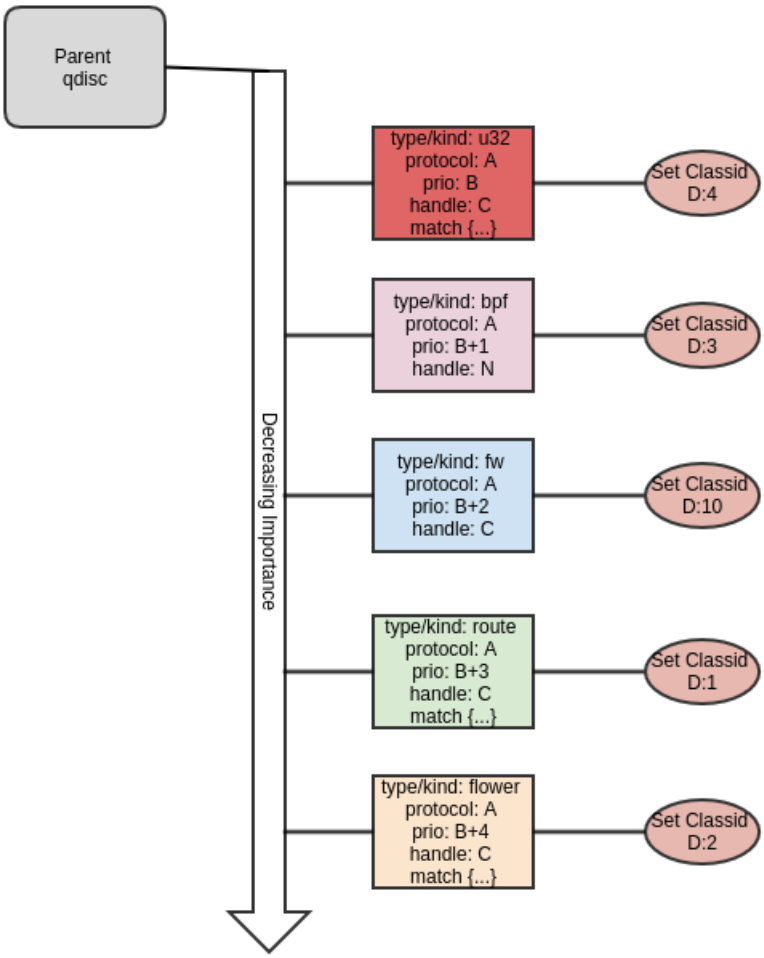


*Figure14: TC Filter Chain*

Figure14 shows a sample of a filter chain.

Filter entries in a chain are *keyed by ethernet protocol* (IPv4, IPV6, ARP, etc). Further there's a loose rule which is important to mention: *Filters with the same protocol on the same chain must have different priorities* with some very speacial exceptions which we will discuss later(u32 filters in a table XXX which may have the same priority).

## *Filter Topology Location*

As was described in section XXXX, the qdisc *enqueue()* method is the point in which classifiers are invoked for the purpose of selecting classes. Filter chains must therefore be installed in the service topology at locations where classful qdiscs reside. Figure14 illustrates an example where a filter chain is attached to a qdisc.

The basic coordinate to consider for a filter follows the same semantic for location as a qdisc/class i.e *{netdev, parent ID}* and as mentioned earlier, location is further refined by *{protocol, priority}*.

### *Filter Identity*

Most of the time it is sufficient to specify the coordinates: *{netdev, parent ID, protocol, priority}* to uniqueuely identify a filter with some exceptions which we will discuss.

Figure14 is a little deceiving - it gives the impression that the lookup is a linked list of filter entries. Infact, depending on the classifier, a filter on a chain can have a reference to a table or even multiple hash tables (the u32 classifier for example is one such classifier). To uniqueuely identify a filter, the TC subsystem uses a 32-bit filter *handle ID*. Note, this is similar to the handle ID that identifies a class or a qdisc. There are two differences compared to qdisc or class handles; in that filter handle semantics are per classifier-type specific:

- filter handles IDs are tied to the classifier kind semantics. In some cases a filter handle could map to a table index; in others a filter handle maps to a hash key not different from a TCAM key, etc. Given the handle is filter kind specific, infact it is possible to have the same filter handle ID amongst filters in a chain (of different classifier kinds). And it may even possible to have the same exact filter id for many different priorities of the same classifier kind when the handle semantic is unneeded.
- filter handles IDs are sometimes required to be specified by the user control (eg the fw classifier requires it) and in other cases are issued by the kernel but could be optionally provided by user control (eg u32 and flower classifiers behave this way).

XXX: Refer to netdev01 and netdev11 papers which describe u32 to some extent (and when the doc grows to be very large we can go into the details).

The summary is: the user may need to understand the classifier kind to make sense of the filter handle.

Let us show an example by installing a *flower* filter on the *DRR* qdisc of Figure13:

```
$ sudo tc filter add dev eth0 parent 1:0 protocol ip priority 10 \
flower dst_mac 52:54:00:3d:c7:6d \
classid 1:1
```

**Listing14: Filter Creation**

The filter instantiated from classifier kind *flower* uses the *flower* classification algorithm to match a destination mac address of *52:54:00:3d:c7:6d*. The filter location (and identity in this case) in the tuples *{netdev, parent ID, protocol, priority}* are set as *{eth0, 1:0, ip, 10}*. The *flower* classifier does not require to set the filter *handle* identification and will provide one for us.

The kernel makes some default choices when a user creates filters; when *priority* is not specified the kernel will choose a very low priorty. Additionally, if the *parent* is not specified egress anchor point (0xFFFFFFFF) is assumed.

Lets display the kernel's view of what we installed:

```
$ sudo tc filter ls dev eth0
filter parent 1: protocol ip pref 10 flower
filter parent 1: protocol ip pref 10 flower handle 0x1 classid 1:1
 dst_mac 52:54:00:3d:c7:6d
 eth_type ipv4
 not_in_hw
```

**Listing15: Filter Configuration**

Observe that the kernel provisioned the handle of 0x1 as the filter id. For flower classifier the handle is not that important for completion, however, we note the filter location and identity above is: *{netdev, parent ID, protocol, priority, handle}* being set as *{eth0, 1:0, ip, 10, 0x1}*.

***Filter Chains***

Figure14 shows a generic example of a filter chain. It shows multiple filters derived from different classifier kinds(bpf, flower, u32, fw, route). Each filter rule matches certain meta/data headers and sets the 32 bit classid (*tc* syntax "classid X:Y") to select a DRR class to enqueue the packet to.

Let us add a second lower priority filter using the u32 classifier to match all ICMP packets on the example setup from Figure13 in Listing16.

```
$ sudo tc filter add dev eth0 parent 1:0 protocol ip priority 11 \
u32 match ip protocol 1 0xff \
classid 1:2
```

**Listing16: Filter Chain Addition**

When the filter matches, we select classid 1:2 as the target class to enqueue on. We visualize this in Figure15:

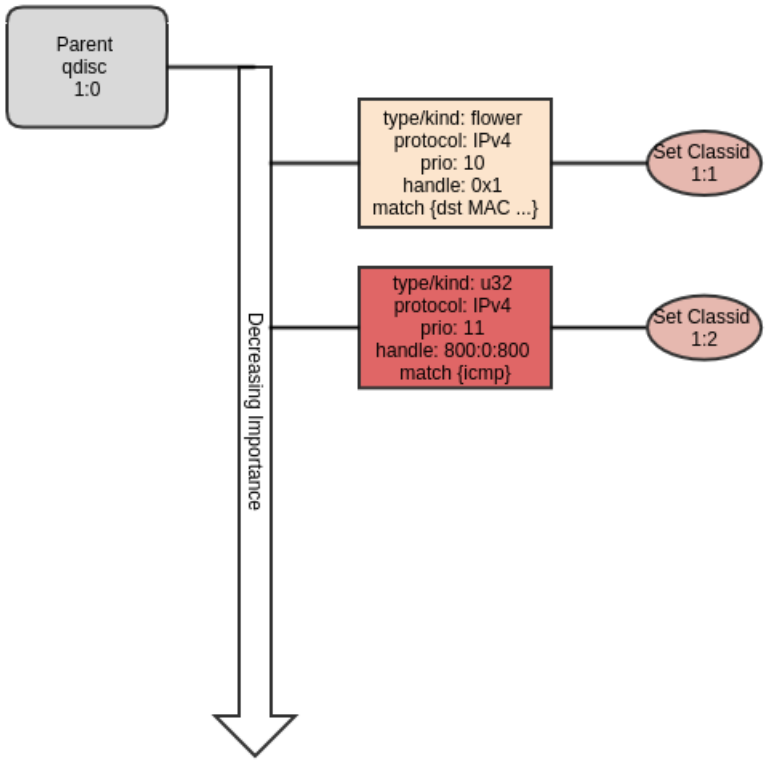


*Figure15: Simple TC Filter Chain At Parent 1:0 Location*

So what happens to a packet arriving on the DRR qdisc at the root anchor?

- The *enqueue()* method consults the built-in priority classifier - it uses the `skb->priority` as input to `direct mapping` classification. If this metadata value matches a known class then the packet is enqueueud to the correct class. Else:
- if it is an IPV4 packet then:
    - if the packet carries a destination MAC address of *52:54:00:3d:c7:6d* it will be match the flower rule and will be enqueued to class 1:1 via that class' *pfifo* enqueue(); else the next priority filter in the chain (u32 rule below) is consulted.
    - if it is an ICMP packet, the u32 rules matches and class 1:2 *enqueue()* is invoked and it ends up in the *prio* qdisc 20:0 using the built-in classifier as mentioned earlier in XXX.
    - else it failed to match - the DRR scheduler drops the packet.
- else it is not an IPv4 packet, and the *DRR* scheduler drops the packet.

Lets dump the filter chain configuration:

```
$ sudo tc filter ls dev eth0
filter parent 1: protocol ip pref 10 flower
classid 65537
filter parent 1: protocol ip pref 10 flower handle 0x1 classid 1:1
 dst_mac 52:54:00:3d:c7:6d
 eth_type ipv4
 not_in_hw

filter parent 1: protocol ip pref 11 u32
filter parent 1: protocol ip pref 11 u32 fh 800: ht divisor 1
filter parent 1: protocol ip pref 11 u32 fh 800::800 order 2048 key ht 800 bkt 0 \
 flowid 1:2 not_in_hw
  match 00010000/00ff0000 at 8
```

**Listing17: Filter Chain Config And State**

As can be observed, the two filters are ordered according to their provisioned priorities. The u32 filter specifies a filter handle of 800::800( or 0x80000800); as mentioned earlier this is very specific to the *u32* way of describing the 32 bit handle id. We will discuss the semantics of u32 filtering later. Also note that *u32* tc output shows a term "flowid"; we will interchangeably use this term with "classid" throughout the document.

Lets say it turns out to be a bad idea to drop packets that failed to match; we add another rule to "match everything else" and send it to class 1:3. For fun we are use a new simple classifier kind called *matchall* (any of the other classifier kind could have been used, but *matchall* exists merely to match everything, so seems ideal to use).

XXX: Listing17 seems to show a bug in flower display? classid 65537..

```
$ sudo tc filter add dev eth0 parent 1:0 protocol ip priority 12 \
  matchall \
  classid 1:3
```

**Listing18: Filter Chain Default Rule Config**

And dumping the filter chain again:

```
$ sudo tc filter ls dev eth0
filter parent 1: protocol ip pref 10 flower
classid 65537
filter parent 1: protocol ip pref 10 flower handle 0x1 classid 1:1
  dst_mac 52:54:00:3d:c7:6d
  eth_type ipv4
  not_in_hw

filter parent 1: protocol ip pref 11 u32
filter parent 1: protocol ip pref 11 u32 fh 800: ht divisor 1
filter parent 1: protocol ip pref 11 u32 fh 800::800 order 2048 key ht 800 bkt 0 \
 flowid 1:2 not_in_hw
  match 00010000/00ff0000 at 8

filter parent 1: protocol ip pref 12 matchall
filter parent 1: protocol ip pref 12 matchall handle 0x1 flowid 1:3
  not_in_hw
```

**Listing19: Filter Chain Config And State**

And to visualize this new configuration from Listing18 it will look as shown in Figure16:

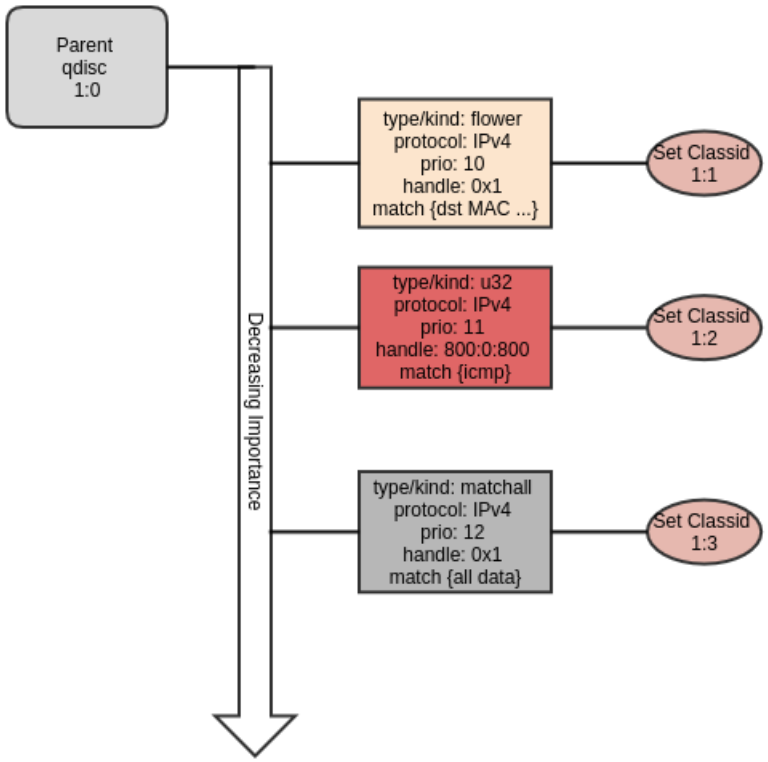


*Figure16: Simple 3 Rule TC Filter Chain At Parent Location 1:0*

With the new setup, any packet not matching the flower or u32 rules will be matched by the matchall rule at priority 12 - and therefore no packets will get dropped.

### *Topology Class selection*

Although we have thus far shown filter chains being adjacent to classful qdiscs selecting classes adjacent to them in the topology, infact a filter can select a class that is on a totaly different topology branch. As an example, in Listing20 we extend the setup in Figure16 and add a *u32* rule to the filter chain at location 1:0 to match src subnet 10.0.0.0/24 and select it to be enqueued on classid 20:3 (i.e not one of the adjacent 1:1-1:3 classes) which is in a downstream branch location in the topoloy.

```
$ sudo tc filter add dev eth0 parent 1:0 protocol ip priority 16 \
  u32 \
  match ip src 10.0.0.0/24 \
  classid 20:3
```

**Listing20: New Filter Creation**

Lets display the filter chain config at that level:

```
$ sudo tc filter ls dev eth0 parent 1:0
  filter pref 10 flower
  classid 65537
  filter protocol ip pref 10 flower handle 0x1 classid 1:1
    dst_mac 52:54:00:3d:c7:6d
    eth_type ipv4
    not_in_hw

  filter protocol ip pref 11 u32
  filter protocol ip pref 11 u32 fh 800: ht divisor 1
  filter protocol ip pref 11 u32 fh 800::800 order 2048 key ht 800 bkt 0 flowid 1:2 \
   not_in_hw
    match 00010000/00ff0000 at 8

  filter protocol ip pref 12 matchall
  filter protocol ip pref 12 matchall handle 0x1 flowid 1:3
    not_in_hw

  filter protocol ip pref 16 u32
  filter protocol ip pref 16 u32 fh 801: ht divisor 1
  filter protocol ip pref 16 u32 fh 801::800 order 2048 key ht 801 bkt 0 flowid 20:3 \
   not_in_hw
    match 0a000000/ffffff00 at 12
```

**Listing21: Filter Chain Config And State**

### *Filter Chains On Multiple Topology Locations*

Figure15 and Figure16 show examples of a filter chain located at parent location 1:0. But as mentioned earlier we can install these rules at different locations as prescribed by the filter rule coordinates.

As an example, on Listing22 we install a filter rule on the *prio* qdisc with handle id 20:0 (inside class 1:2) in the topology of Figure13.

```
$ sudo tc filter add dev eth0 parent 20: protocol ip priority 10 \
  u32 \
  match ip src 192.168.8.0/24 \
  classid 20:1
```

**Listing22: New Filter Chain Creation**

The filter rule selects class 20:1 for IPv4 packets matching IP source subnet 192.168.8.0/24

Dumping the new filter chain config:

```
$ sudo tc filter ls dev virbr0 parent 20:
  filter protocol ip pref 10 u32
  filter protocol ip pref 10 u32 fh 800: ht divisor 1
  filter protocol ip pref 10 u32 fh 800::800 order 2048 key ht 800 bkt 0 \
   flowid 20:1
    match c0a80800/ffffff00 at 12
```

**Listing23: New Filter Chain Config**

XXX: May need to describe how a packet gets treated here..

XXX: May need a diagram?

***Basic Actions***

So far we have seen that when a filter matches we can execute a "select a queue/class" action ("classid X:Y").

The TC Classifier-action block provides a lot more actions and an architecture that allows composition of a complex graph of actions when a packet is matched. As an introduction we will demonstrate a few basic actions.

Lets add an action on parent 20:0 to count packets coming from source IP 192.168.8.1 and then enqueue them on classid 20:2 in Listing24:

```
$ sudo tc filter add dev eth0 parent 20: protocol ip priority 9 \
  u32 \
  match ip src 192.168.8.1/32 \
  classid 20:1 \
  action ok
```

**Listing24: New Filter Chain Extension**

The action "ok" accepts all packets and accounts for the packets and their bytes. Notice how we stratetigally placed this rule before the rule which matches the whole subnet by setting a lower priority.

Lets add another action on parent 20:0 to "drop" packets coming from source IP 192.168.8.32/32

```
$ sudo tc filter add dev eth0 parent 20: protocol ip priority 8 \
  u32 \
  match ip src 192.168.8.32 \
```

```
  classid 20:1 \
  action drop
```

**Listing25: New Filter Chain Extension**

And now lets dump the filter chain config at parent 20:

```
$ sudo tc filter ls dev virbr0 parent 20:filter protocol ip pref 9 u32

  filter protocol ip pref 9 u32 fh 803::801 order 2049 key ht 803 bkt 0 \
   terminal flowid 20:1
    match c0a80820/ffffffff at 12
       action order 33: gact action drop
        random type none pass val 0
        index 3 ref 1 bind 1

  filter protocol ip pref 9 u32 fh 803: ht divisor 1
  filter protocol ip pref 9 u32 fh 803::800 order 2048 key ht 803 bkt 0 \
   flowid 20:2
    match c0a80801/ffffffff at 12
       action order 1: gact action pass
        random type none pass val 0
        index 2 ref 1 bind 1

  filter protocol ip pref 10 u32
  filter protocol ip pref 10 u32 fh 800: ht divisor 1
  filter protocol ip pref 10 u32 fh 800::800 order 2048 key ht 800 bkt 0 \
   flowid 20:1
    match c0a80800/ffffff00 at 12
```

**Listing26: New Filter Chain Config And State**

XXX: Build up this as an arguement to talk about classifier-action blocks and queue-less qdiscs..

# Classifier Action Blocks

More advanced stuff: Multi-chains, blocks etc. XXX: Describe classifiers - location/identity address of classifiers.. XXX: Describe relationship to Actions XXX: Describe relationship to chains/blocks

## *Actions*

XXX: We go into details of Action topology etc...

## *Multiple Filter Tables*

Talk about using u32 and then using multiple filter chains

## *Shared Classifier-Action Blocks*

XXX

## Hardware Offloading

XXX: Talk about how we offload qdiscs, filters and actions

## Appendix: Kernel + User Code Overview

We go into code details

### *Qdisc Ops*

XXX: Go into kernel level qdisc_ops structure XXX: Go into the netlink messages to configure the qdiscs

### *Class Ops*

XXX: Go into kernel level class_ops structure XXX: Go into the netlink messages to configure the classes

### *Filter Ops*

XXX: Go into kernel level filter_ops structure XXX: Go into the netlink messages to configure the filters

### *Action Ops*

XXX: Go into kernel level action_ops structure XXX: Go into the netlink messages to configure the actions