Linux - Advanced Networking Overview
Version 1

Saravanan Radhakrishnan
Information and Telecommunications Technology Center
Department of Electrical Engineering & Computer Science
The University of Kansas
Lawrence, KS 66045-2228

August 22, 1999

1 Disclaimer

All the text in this document is purely based on my understanding of implementation of various features. I have read some documents and I have seen the code myself, and I describe them based on my understanding. If the readers notice any description of a concept which appears contrary to their understanding of the concept, the issue can be taken up for discussion and corrections will be made to the document as necessary. I would appreciate all suggestions and comments made in an attempt to make the quality of this document better.

2 Introduction

Linux, a shareware operating system, supports a number of advanced networking features, thanks largely to the huge linux networking community. Besides the reliable TCP/UDP/IP protocol suite, a number of new features like firewalls, QoS, tunneling etc. has been added to the networking kernel. This document reviews these advanced networking features that have been implemented in the linux kernel, from a configuration, implementation and usage standpoint. Examples of usage and pointers to references have been given when appropriate.

The advanced networking features that have been dealt with in this document include the Quality of Service support in linux, which encompasses a description of the differentiated services effort, the firewall implementation using ipchains, the VPN implementation using GRE tunnels and the advanced routing implementation using netlink sockets.

The later part of this document discusses briefly some of the common tools that are available on linux. This section will discuss "zebra", a distributed routing software that is used to configure a linux box as a router, and a configuration tool to set up firewalls.

3 QoS Support in Linux

3.1 Introduction

This section discusses the QoS support that is available in the recent linux kernels. The QoS support in the kernel provides the framework for the implementation of various IP QoS technologies like integrated services [1] and differentiated services [2]. Let us begin by discussing the details of the configuration, implementation and usage of the QoS support in linux.

3.2 Configuration

The support for quality of service is available from linux kernel versions 2.1.90. However, the support is more comprehensive in the more recent kernels. This document is written with reference to the kernel version 2.2.1. This kernel also has support for differentiated services in the form of a patch, that can be downloaded from ftp://lrcftp.epfl.ch/pub/linux/diffserv/patches/ds-3.patch.gz. This patch needs to be applied in order to exercise all the QoS

features supported in linux. The latest linux kernels can be downloaded from http://www.kernelnotes.org/.

1. Apply the diff-serv patch to the linux-2.2.1 source tree.

2. Do a 'make xconfig' or 'make menuconfig' or 'make make config' in the /usr/src/linux directory.

3. Set the EXPERIMENTAL_OPTIONS to 'y'.

4. Under networking options, say 'y' to the following kernel options: Kernel/User netlink socket, Routing messages,TCP/IP networking and QoS and/or fair queueing. After turning on the QoS and/or fair queuing option, enable the CBQ, CSZ, PRIO, RED, SFQ, TEQL, TBF, GRED, DS_MARK, 'tcindex' classifier, Packer Classifier API, U32 classifiers and routing table based classifier.

5. Do a 'make dep; make clean; make bzilo'

6. Reboot the linux box using the new kernel image.

Having discussed the configuration of the QoS support in linux, let us now discuss the details involved in the implementation of these features. The location of all the kernel related files referred to in the rest of this document are specified with respect to the /usr/src/linux directory.

3.3 Implementation

3.3.1 Basic Principle

The basic principle involved in the implementation of QoS in linux is shown in Figure 1. This figure shows how the kernel processes incoming packets, and how it generates packets to be sent to the network. The input de-multiplexer examines the incoming packets to determine if the packets are destined for the local node. If so, they are sent to the higher layer for further processing. If not, it sends the packets to the forwarding block. The forwarding block, which may also received locally generated packets from the higher layer, looks up the routing table and determines the next hop for the packet. After this, it queues the packets to be transmitted on the output interface. It is at this point that the linux traffic control comes into play. Linux traffic control can

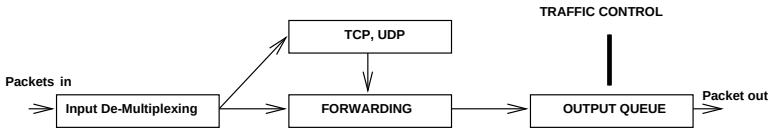

Figure 1: Linux Traffic Control

be used to build a complex combination of queuing disciplines, classes and filters that control the packets that are sent on the output interface.

From an implementation standpoint, what this means is this. When queuing disciplines are created for a device, a pointer to the queue is maintained in the device structure (in include/netdevice.h). The IP layer, after adding the necessary header information to a packet (in net/ipv4/ip_output.c) , calls the function dev_queue_xmit (in net/core/dev.c). A portion of this code is shown below.

```
q = dev->qdisc;
if (q->enqueue) {
    q->enqueue(skb, q);
    qdisc_wakeup(dev);
    return 0;
}
.
.
.
.
if (dev->hard_start_xmit(skb, dev) == 0)
.
.
.
```

This function shows that before actually sending the packet on the output interface (by doing a hard_start_xmit), the packet is enqueued in the queue maintained by the device, if one exists. Thus, an mentioned before, traffic control is implemented just before the packet is sent to the device driver.

As already mentioned, the linux traffic control mechanism provides the basic framework for the development of integrated services [1] and differentiated services [2] support in linux. This is shown in Figure 2.

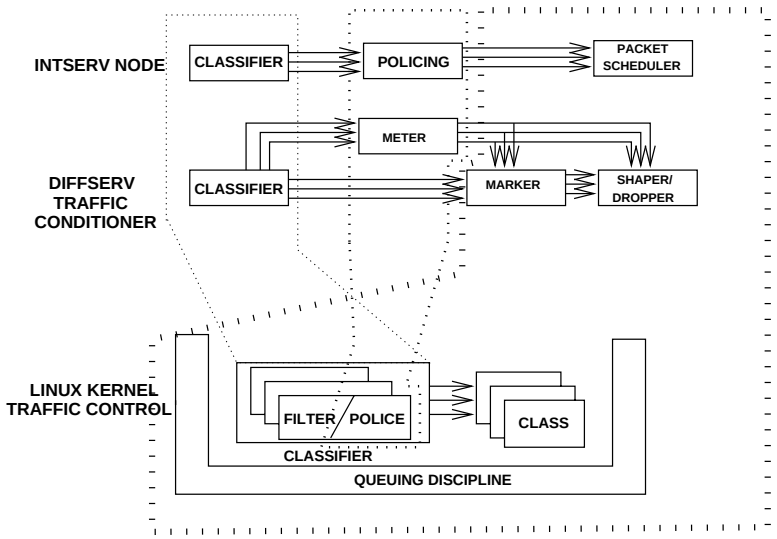

Figure 2: Framework for developing "intserv" and "diffserv"

As shown in Figure 2, the QoS support in linux consists of the following three basic building blocks, namely:

- Queueing discipline
- Classes
- Filters/Policers

Let us now discuss these basic blocks in detail.

3.4 Queuing Disciplines

This section discusses queuing disciplines, which form a basic. building block for the support of QoS in linux. It also discusses the various queuing disciplines that are supported in linux. Each network device has a queue associated with it. There are 11 types of queuing disciplines that are currently supported in linux, which includes:

- Class Based Queue (CBQ)
- Token Bucket Flow (TBF)

- Clark-Shenker-Zhang (CSZ)
- First In First Out (FIFO)
- Priority
- Traffic Equalizer (TEQL)
- Stochastic Fair Queuing (SFQ)
- Asynchronous Transfer Mode (ATM)
- Random Early Detection (RED)
- Generalized RED (GRED)
- Diff-Serv Marker (DS_MARK)

Queues are identified by a handle <major number:minor number>, where the minor number is zero for queues. Handles are used to associate classes to queuing disciplines. Classes are discussed in the next subsection.

Queuing disciplines and classes are tied to one another. The presence of classes and their semantics are fundamental properties of the queuing disciplines. In contrast, filters can be arbitrarily combined with queuing disciplines and classes, as long as the queuing disciplines have classes. Not all queuing disciplines are associated with classes. For example, the Token Bucket Flow (TBF) does not have any classes associated with it.

Figure 3 shows an example queue. In this example, there are two types of queuing disciplines, one for high priority and one for low priority. The filter selects the one for high priority, while the remaining are treated low priority. The low priority packets are served by a FIFO queue, while the high priority packets are served by a Token Bucket Flow Algorithm. The TBF queue is used to ensure that the low priority packets are not starved.

One of the main advantages of the QoS support in linux is the flexibility with which the combination of queues and classes can be set up. Each queuing discipline may have a number of classes. These classes don't store the packets themselves, but instead, use another queuing discipline for this purpose, which in turn, may have a number of classes and so on. It is this flexibility that makes the QoS support in linux unique.

When a linux kernel configured for QoS support is booted up, the function net_dev_init (in net/core/dev.c) calls the function pktsched_init (in net/sched/sch_api.c)

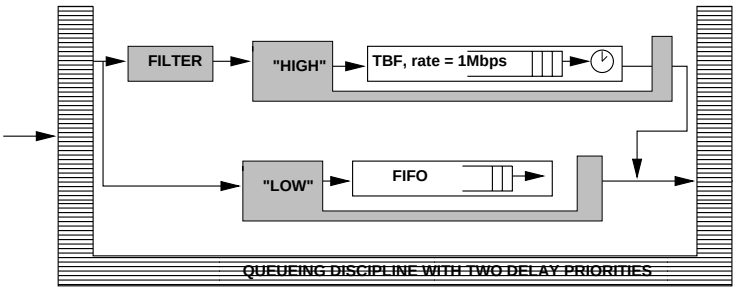

Figure 3: An Example Queue

to initialize the traffic control unit in the linux kernel. In pktsched_init, the queuing disciplines that have been compiled into the kernel are all registered and initialized. The pointers to access the the functions tc_ctl_qdisc, tc_dump_qdisc, tc_ctl_tclass and tc_dump_tclass, which are used to perform various functions on queuing disciplines and classes are also initialized in pktsched_init.

The functions that are supported on the various queuing disciplines are discussed in the following sections. These functions are defined in the Qdisc_ops structure in include/net/pkt_sched.h.

3.4.1 Enqueue

The enqueue function enqueues a packet with the queueing discipline. Packets are enqueued in the following manner. As already shown in the previous section, when the IP layer calls dev_queue_xmit, the enqueue function of the queuing discipline attached to the device is called. This portion of the code in net/core/dev.c is shown below:

```
q = dev->qdisc;
if (q->enqueue) {
    q->enqueue(skb, q);
    qdisc_wakeup(dev);
    return 0;
}
```

In the enqueue function of a queuing discipline, the filters are run one by one until a match occurs. Once the match occurs, the enqueue function of the queueing discipline "owned" by that class is executed. For example, in the cbq_enqueue function (in net/sched/sch_cbq.c),

```
struct cbq_class *cl = cbq_classify(skb, sch);
int len = skb->len;

if (cl && cl->q->enqueue(skb, cl->q) == 1)
.
.
```

The function cbq_classify is used to apply the filters and determine the class to which to the packet belongs. After that, the enqueue function of the queuing discipline owned by that class is called. This queuing discipline may have its own classes, which in turn may be associated with some other queuing discipline, and so on, which makes the usage flexible, as was discussed earlier.

At this point, it is worth mentioning that when a class is created, the default queuing discipline that it owns is a Priority FIFO queue. The portion of the code (in net/sched/sch_cbq.c) that does this is shown below.

```
static int cbq_init(struct Qdisc *sch, struct rtattr *opt)
{
.
.
.

if (!(q->link.q = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops)))
    q->link.q = &noop_qdisc;
.
.
.
}
```

This can be changed by doing a graft operation, that will be discussed later in the section on classes.

3.4.2 Dequeue

The dequeue function dequeues a packet for sending. It returns the next packet that needs to be sent out on the output interface. This packet is determined by the scheduler in the queuing discipline. The scheduler can be very complicated for complex queuing disciplines like the CBQ. At the same

time, it can be very simple too, as in the case of a FIFO queue. The dequeue function for a simple FIFO queuing discipline (in net/sched/sch_fifo.c) is shown below:

```
static struct sk_buff *
pfifo_dequeue(struct Qdisc* sch)
{
    return __skb_dequeue(&sch->q);
}
```

As shown in this example, the next packet in the queue is dequeued and returned, which is desired behavior from a simple FIFO.

The dequeue function for a priority FIFO is shown next.

```
static struct sk_buff *
prio_dequeue(struct Qdisc* sch)
{
 .
 .
 .
 for (prio = 0; prio < q->bands;prio++) {
 qdisc = q->queues[prio];
 skb = qdisc->dequeue(qdisc);
 if (skb) {
  sch->q.qlen--;
  return skb;
   }
 }
 .
 .
 .
}
```

As shown in this example, whenever the prio_dequeue function is called, packets from the highest priority queue are sent first. After all the packets in the highest priority level are sent, packets from the next priority level are dequeued. This required behavior is provided by the portion of the code given above.

Having discussed the dequeue function of the queuing disciplines, let us now see the places where the dequeue function is invoked. Whenever a packet is enqueued in dev_queue_xmit, the qdisc_wakeup function (in include/net/pkt_sched.h) is invoked in an attempt to send the packet that was just enqueued. qdisc_wakeup invokes the qdisc_restart function (in net/sched/sch_generic.c), which invokes the dequeue function of the queuing discipline attached to the device. The dequeue function returns the next packet that needs to be sent out on the interface. qdisc_restart then invokes hard_start_xmit of the device to send the packet down to the device. If hard_start_xmit fails for some reason, the packet is requeued in the queuing discipline. The requeue function is discussed in a later section.

qdisc_wakeup can also be invoked from the watchdog timer handlers in the CBQ, TBF and CSZ schedulers. In the dequeue function of these queuing disciplines, when a packet is dequeued to be sent on the output interface, a watchdog timer is initiated. If for some reason, qdisc_restart does not send the packet out in time, the watchdog timer will go off and qdisc_restart is called. For example, the setting of the TBF watchdog timer in tbf_dequeue (in net/sched/sch_tbf.c) is shown below:

```
static struct sk_buff *
tbf_dequeue(struct Qdisc* sch)
{
.
.
.

if (!sch->dev->tbusy) {
.
.
    del_timer(&q->wd_timer);
    q->wd_timer.expires = jiffies + delay;
    add_timer(&q->wd_timer);
}
.
.
.
```

This example shows the way the watchdog timer is set. Yet another way of calling the dequeue function on a device is via qdisc_run_queues (in net/sched/sch_generic.c) from net_bh (in net/core/dev.c). net_bh is the bottom half handler of the networking stack in linux and is executed whenever packets are queued up for processing. In qdisc_run_queues, qdisc_restart is invoked and the rest of the actions are similar to those explained already.

Having discussed the dequeue function of a queuing discipline, let us now discuss the requeue function.

3.4.3 Requeue

The requeue function of a queuing discipline requeues a packet for transmission. After dequeueing the packet, if for some reason, the packet is not transmitted, the packet needs to be put back in the queue at the same position from where it was dequeued. The reasons for which a hard_start_xmit may fail include:

- If the device cannot establish its busy status before the start of the transmission.
- If the device is itself buggy.
- If the fastroute option is enabled.

The simplest example to demonstrate the requeue function is the pfifo_requeue (in net/sched/sch_fifo.c). The portion of the code that does the requeuing is shown below:

```
static int
pfifo_requeue(struct sk_buff *skb, struct Qdisc* sch)
{
    __skb_queue_head(&sch->q, skb);
    return 1;
}
```

For a simple FIFO queue, the requeue function should put the packet back at the head of the queue. This is what is done in the portion of the code shown above. The requeue function is different from an enqueue function in that the requeue function should put the packet back in the same place from

where it was dequeued, and it should not be reflected in the statistics that are maintained for the queue, since it was already processed by an enqueue function.

Let us now discuss the drop function of a queuing discipline.

3.4.4 Drop

The drop function is used to drop a packet from the queue. This is a very simple function which may be needed in the case of certain queuing disciplines like the RED and the GRED. These queuing disciplines will need to drop some packets under certain conditions. A portion of the GRED code (in net/sched/sch_gred.c) is shown below:

```
static int
gred_drop(struct Qdisc* sch)
{
.
.
.
    skb = __skb_dequeue_tail(&sch->q);
    if (skb) {
q= t->tab[(skb->tc_index&0xf)];
sch->stats.backlog -= skb->len;
sch->stats.drops++;
q->backlog
    -=
    skb->len;
q->other++;
kfree_skb(skb);
return
    1;
    }
.
.
}
```

As can be seen from above, the drop function simply involves dequeuing the packet from the queue and freeing the memory occupied by it.

Next, let us discuss the init function of a queuing disciplines.

3.4.5 Init

The init function of a queuing discipline is used to initialize and configure the parameters of a queuing discipline when it is created. The init function can be passed the arguments that will used to configure the queuing discipline. Each of the queuing disciplines need different sets of parameters during the process of the initial configuration. These parameters will be discussed in detail during the discussion on the usage of the traffic control features in linux. All the scheduler related files, for example, sch_cbq.c, sch_tbf.c etc. contain the scheduler data structure, that is the parameters that will used by the scheduler to determine the packet that needs to be sent next. This data structure can be very simple for certain queuing disciplines, for example, priority FIFO, or it can really very complicated for certain queuing disciplines, for example CBQ.

The scheduler data structure, for a simple FIFO (in net/sched/sch_fifo.c) is shown below:

```
struct fifo_sched_data
{
    unsigned limit;
};
```

The only parameter that a FIFO scheduler needs is the maximum length of the queue. The above structure shows the limit argument which indicates the maximum length of the queue, beyond which packets are dropped. In the fifo_init function (in net/sched/sch_fifo.c), the following portion of code demonstrates the initialization and configuration that is done in general, in the init function of a queuing discipline.

```
static int fifo_init(struct Qdisc *sch, struct rtattr *opt)
{
 struct fifo_sched_data *q = (void*)sch->data;
 if (opt == NULL) {
   q->limit = sch->dev->tx_queue_len;
   if (sch->ops == &bfifo_qdisc_ops)
       q->limit *= sch->dev->mtu;
 }
 else
 {
```

```
    struct tc_fifo_qopt *ctl = RTA_DATA(opt);
    if (opt->rta_len < RTA_LENGTH(sizeof(*ctl)))
       return -EINVAL;
    q->limit = ctl->limit;
  }
  return 0;
}
```

In the above function, if the limit argument is specified, the init function sets the limit to the specifies value. If this is not specified, it defaults the limit to the MTU of the device.

3.4.6 Reset

The reset function of a queuing discipline is used to reset the queuing discipline to its initial state. It clears all queuing disciplines, timers are stopped etc. The reset of a queuing discipline also results in a reset of the queuing discipline of the classes of this queuing discipline. As an example, let us take a look at the FIFO queuing discipline. The fifo_reset (in net/sched/sch_fifo.c) function is shown below:

```
static void
fifo_reset(struct Qdisc* sch)
{
    struct sk_buff *skb;

    while ((skb=__skb_dequeue(&sch->q)) != NULL)
kfree_skb(skb);
    sch->stats.backlog = 0;
}
```

In this example, the reset function results in the queue being drained, and the memory occupied by these packets are recovered. The backlog is set to zero. This is a simple example, which did not involve timers. A slightly more complicated example is discussed next. Let us take a look at the function tbf_reset (in net/sched/sch_tbf.c):

```
static void
tbf_reset(struct Qdisc* sch)
{
```

```
 struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;

 skb_queue_purge(&sch->q);
 sch->stats.backlog = 0;
 PSCHED_GET_TIME(q->t_c);
 q->tokens = q->buffer;
 q->ptokens = q->mtu;
 sch->flags &= ~TCQ_F_THROTTLED;
 del_timer(&q->wd_timer);
}
```

This example involves clearing of the backlog as well as resetting of the timers. This explains the operations of the reset function of a queuing discipline.

3.4.7 Destroy

The destroy function of a queuing discipline is used to remove the queuing discipline. It also removes all the classes and filters associated with queuing discipline. This is implemented in the qdisc_destroy function (in net/sched/sch_generic.c). An important portion of this function is shown is below:

```
void qdisc_destroy(struct Qdisc *qdisc)
{
.
.
.
for (qp = &qdisc->dev->qdisc_list; (q=*qp) != NULL; qp = &q->next)
.
.
.
}
```

This portion of the code shows that a linear search is performed to determine the queuing discipline that needs to be destroyed, after which the memory occupied by the queue is released. As an example of the destroy function, let us take a look at the cbq_destroy function (in net/sched/sch_cbq.c):

```
static void
```

```
cbq_destroy(struct Qdisc* sch)
{
 struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
 struct cbq_class *cl;
 unsigned h;

 for (h = 0; h < 16; h++) {
  for (cl = q->classes[h]; cl;
    cl = cl->next)
    cbq_destroy_filters(cl);
 }

 for (h = 0; h < 16; h++) {
  for (cl = q->classes[h]; cl; cl = cl->next)
   if (cl != &q->link) cbq_destroy_class(cl);
 }
 qdisc_put_rtab(q->link.R_tab);
}
```

This function results in the filters and the classes associated with the queuing discipline being deleted. As was already mentioned, the destruction of a class results in the destruction of the queuing discipline owned by that class, which can be seen from the cbq_destroy_class function shown below:

```
static void cbq_destroy_class(struct cbq_class *cl)
{
    cbq_destroy_filters(cl);
    qdisc_destroy(cl->q);
    qdisc_put_rtab(cl->R_tab);
#ifdef CONFIG_NET_ESTIMATOR
    qdisc_kill_estimator(&cl->stats);
#endif
    kfree(cl);
}
```

Let us now discuss the last function associated with a queuing discipline, namely the dump function.

3.4.8 Dump

The dump function is used to dump diagnostic data associated with a queuing discipline. Each queuing discipline maintains different diagnostic data that is dumped when the dump function is invoked. There is nothing more that needs to be elaborated about this function.

This concludes the discussion on the implementation of queuing disciplines. Let us now discuss the implementation of classes in the linux kernel.

3.5 Classes

As already mentioned, queues and classes are tied to one another. Each class owns a queue, which by default is a FIFO queue. When the enqueue function of a queuing discipline is called, the queuing discipline applies the filters to determine the class to which the packet belongs. It then calls the enqueue function of the queuing discipline that is owned by this class.

There are two ways by which a class can be identified. One is via the class identifier, which is specified by the user. The other identifier, which is used within the kernel to identify a class, is referred to as the internal identifier. This ID is unique and is assigned by the queuing discipline. The class ID is a u32 data type, while the internal ID is an unsigned long integer. Most of the functions on classes use the internal ID to identify the class. However, there are a few functions (like the get and the change function, which will be discussed later) that use the class ID too.

Multiple class IDs may map to the same internal ID, however, the class ID will convey some additional information from the classifier to the queuing discipline or class.

The class ID, similar to a queuing discipline identifier, is structured in the form of a < major number:minor number>. The major number corresponds to their instance of the queuing discipline while the minor number identifies the class within that instance.

Not all queuing disciplines support classes. The ones that support classes include the CBQ, the DS_MARK, the CSZ and the p-FIFO queuing disciplines. The rest of the queuing disciplines do not support classes.

With this introduction on classes, let us now discuss the functions that can be performed on classes. These functions are defined in the Qdisc_class_ops structure in include/net/pkt_sched.h.

The following operations are permitted for the manipulation of the classes

within the various queuing disciplines that support classes. This is defined in include/net/pkt_sched.h.

3.5.1 Graft

The graft function on a class is used to attach a new queuing discipline to a class. As mentioned in the previous section, the default queuing discipline attached to a class when it is created, is a FIFO queue. To change this queuing discipline, a graft operation is performed on the class. As an example, let us take a look at the cbq_graft function in net/sched/sch_cbq.c.

```
static int cbq_graft(struct Qdisc *sch, unsigned long arg,
struct Qdisc *new, struct Qdisc **old)
{
 struct cbq_class *cl = (struct cbq_class*)arg;

 if (cl) {
  if (new == NULL) {
   if ((new = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops))==NULL)
      return -ENOBUFS;
   new->classid = cl->classid;
  }

  if ((*old = xchg(&cl->q, new)) != NULL)
   qdisc_reset(*old);

  return 0;
 }
 return -ENOENT;
}
```

As shown above, the cbq_graft function is called with the new queuing discipline. By default, if no queuing discipline is specified, a FIFO queuing discipline is attached to the class. If the new queuing discipline is specified, then the class is attached to it. That is, when a packet to be enqueued is classified to this class, the enqueue function of the queuing discipline attached to this class is called. The old queuing discipline that was attached to the class is returned in the "old" variable.

The qdisc_graft function in net/sched/sch_api.c is another example for the graft function. This function is called from the tc_ctl_qdisc function in the same file. The tc_ctl_qdisc function is invoked whenever an attempt is made to create, delete, change or get a queuing discipline.

3.5.2 Get

The get function is used to return the internal ID of a class, given its class ID. The get function increments the usage count of the class. As an example, let us take a look at the cbq_get function in net/sched/sch_cbq.c.

```
static unsigned long cbq_get(struct Qdisc *sch, u32 classid)
{
 struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
 struct cbq_class *cl = cbq_class_lookup(q, classid);

 if (cl) {
   cl->refcnt++;
   return (unsigned long)cl;
 }
 return 0;
}

static __inline__ struct cbq_class *
cbq_class_lookup(struct cbq_sched_data *q, u32 classid)
{
 struct cbq_class *cl;

 for (cl = q->classes[cbq_hash(classid)]; cl; cl = cl->next)
  if (cl->classid == classid)
    return cl;
  return NULL;
}
```

As shown in this function, the get function calls the cbq_class_lookup function to map the class ID to the internal ID. The cbq_class_lookup function searches the list of classes to determine the class with the specified class ID and return a pointer to the corresponding cbq_class structure, which contains the internal ID of the class, in addition to other information. The

get function is invoked in the tc_ctl_tclass function in net/sched/sch_api.c. The tc_ctl_tclass function is invoked whenever an attempt is made to create, delete or change a class.

3.5.3 Put

The put function is invoked when a class previously referenced using the get function is de-referenced. It decrements the usage count of the class. If the usage count reaches zero, put may remove the class itself. As an example, let us take a look at the cbq_put function in net/sched/sch_cbq.c.

```
static void cbq_put(struct Qdisc *q, unsigned long arg)
{
    struct cbq_class *cl = (struct cbq_class*)arg;

    start_bh_atomic();
    if (--cl->refcnt == 0)
cbq_destroy_class(cl);
    end_bh_atomic();
    return;
}
```

As shown above, the cbq_put function decrements the usage count of the class, and if zero, destroys the class. The put function is invoked in the tc_ctl_tclass function in net/sched/sch_api.c. The tc_ctl_tclass function is invoked whenever an attempt is made to create, delete or change a class.

3.5.4 Change

The change function on a class is used to change the properties associated with a class. However, the change function is also used to create classes at times. As an example, let us see the cbq_change function in net/sched/sch_cbq.c in detail. The cbq_change function is invoked with the queuing discipline, the class ID of the class whose properties need to be changed or added and the new properties that need to associated to the class. This can be seen below:

```
static int
cbq_change(struct Qdisc *sch, u32 classid, u32 parentid, struct rtattr **tca,
```

```
unsigned long *arg)
{
.
.
}
```

After performing some initial checks, the properties specified are associated to the class one by one. This can be seen the following portion of the code:

```
if (tb[TCA_CBQ_WRROPT-1]) {
    cbq_rmprio(q, cl);
    cbq_set_wrr(cl, RTA_DATA(tb[TCA_CBQ_WRROPT-1]));
}

if (tb[TCA_CBQ_OVL_STRATEGY-1])
    cbq_set_overlimit(cl, RTA_DATA(tb[TCA_CBQ_OVL_STRATEGY-1]));
```

This portion of the code sets the priorities and weighted round robin parameters for a class. It also sets the overlimit information. These details about the CBQ will be discussed in more detail in a later section. It should be noted that this function can be used to not only change the properties of an existing class, but it can also be used to create a new class with the properties specified. This fact is clear from the following portions of the code.

```
static int
cbq_change(struct Qdisc *sch, u32 classid, u32 parentid, struct rtattr **tca,
unsigned long *arg)
{
struct cbq_class *cl = (struct cbq_class*)*arg;
.
.
if (cl) {
  /* Change properties */
  .
  .
  return 0;
  }
```

```
/* Create a new class and assign the properties to the class */
.
.
.
}
```

The change function is invoked in the tc_ctl_tclass function in net/sched/sch_api.c. The tc_ctl_tclass function is invoked whenever an attempt is made to create, delete or change a class.

3.5.5 Delete

The delete function on a class is used to delete the class. It determines the usage of the class, by checking the reference count, and if zero, de-activates and removes the class. As an example, let us take a look at the cbq_delete function in net/sched/sch_cbq.c.

```
static int cbq_delete(struct Qdisc *sch, unsigned long arg)
{
 struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
 struct cbq_class *cl = (struct cbq_class*)arg;

 if (cl->filters || cl->children || cl == &q->link)
      return -EBUSY;

 start_bh_atomic();

 if (cl->next_alive)
      cbq_deactivate_class(cl);

 if (q->tx_borrowed == cl)
      q->tx_borrowed = q->tx_class;
 if (q->tx_class == cl) {
      q->tx_class = NULL;
      q->tx_borrowed = NULL;
 }

 cbq_unlink_class(cl);
 cbq_adjust_levels(cl->tparent);
```

```
 cl->defmap = 0;
 cbq_sync_defmap(cl);

 cbq_rmprio(q, cl);

 if (--cl->refcnt == 0)
      cbq_destroy_class(cl);

 end_bh_atomic();

 return 0;
}
```

The above function is an example of the delete function on a class. Let us now look into the walk function.

3.5.6 Walk

The walk function on a class is used to iterate over all the classes of a queuing discipline and invokes a callback function for each of the classes. This is usually used to obtain diagnostic data for all the classes of a queuing discipline. For example, let us take a look at the cbq_walk function in net/sched/sch_cbq.c.

```
static void cbq_walk(struct Qdisc *sch, struct qdisc_walker *arg)
{
 struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
 unsigned h;
.
.

 for (h = 0; h < 16; h++) {
    struct cbq_class *cl;

    for (cl = q->classes[h]; cl; cl = cl->next) {
       if (arg->count < arg->skip) {
 arg->count++;
 continue;
    }
```

```
    if (arg->fn(sch, (unsigned long)cl, arg) {
       arg->stop = 1;
       break;
    }
.
.

}
```

This portion of the code shows that the walk command iterates over all the classes of a specified queuing discipline and invokes a callback function. The walk function is called from the tc_dump_tclass function in net/sched/sch_api.c, which is invoked when a dump request is made on the class. The portion of the code in tc_dump_tclass that does this shown below:

```
static int tc_dump_tclass(struct sk_buff *skb, struct netlink_callback *cb)
{
 struct device *dev;
 struct Qdisc *q;
 .
 .
 .
 for (q=dev->qdisc_list, t=0; q; q = q->next, t++) {
   .
   arg.w.fn = qdisc_class_dump;
   .
   q->ops->cl_ops->walk(q, &arg.w);
   .
   .
   }
}
```

The callback function maps to the qdisc_class_dump function in the same file. In qdisc_class_dump, the tc_fill_tclass function is invoked, which calls the dump function on all the classes. The dump function on a class, which is discussed later is used to dump statistical information about the class.

3.5.7 Tcf_chain

The tcf_chain function on a class is used to return the anchor to the list of filters that are associated to a class. Each class is associated with a filter list which contains the list of filters that are used to identify the packets that belong to a particular class. As already mentioned, packets with different properties may map to the same class. For example, packets from two different sources may map to the same class. As a result, there may be multiple filters associated to a class. Filters are discussed in more detail in the next section. The cbq_find_tcf function in net/sched/sch_cbq.c is shown below as an example:

```
static struct tcf_proto **cbq_find_tcf(struct Qdisc *sch, unsigned long arg)
{
    struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
    struct cbq_class *cl = (struct cbq_class *)arg;

    if (cl == NULL)
       cl = &q->link;

    return &cl->filter_list;
}
```

The cbq_find_tcf function returns the pointer to the list of filters. Filters are maintained in a structure named tcf_proto in include/net/pkt_cls.h. The tcf_chain function is invoked in the tc_ctl_tfilter function in net/sched/cls_api.c. This function is called when an attempt is made to create/modify/delete/get a filter node.

3.5.8 Bind_tcf

The bind_tcf function is used to attach an instance of a filter to a class. The cbq_bind_filter function in net/sched/sch_cbq.c is an example for the bind_tcf function on a class.

```
static unsigned long cbq_bind_filter(struct Qdisc *sch, u32 classid)
{
    struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
    struct cbq_class *cl = cbq_class_lookup(q, classid);
```

```
    if (cl) {
cl->filters++;
return (unsigned long)cl;
    }
    return 0;
}
```

As shown in this function, when the cbq_bind_filter function is called, the filter count for the class is incremented. This function is almost similar to the get function on a class. The difference is that a class that is pointed to by filters cannot be deleted without deleting the filters. That is, the queuing discipline explicitly refuses requests to delete a class if the class is in use. This can be seen from the following portion of the code from the cbq_delete function in net/sched/sch_cbq.c

```
static int cbq_delete(struct Qdisc *sch, unsigned long arg)
{
.
.
 if (cl->filters || cl->children || cl == &q->link)
                return -EBUSY;
.
.
.
}
```

However, there is a bug in the dsmark queuing discipline implementation. In this case, the get and bind_tcf functions map to the dsmark_get function, which is incorrect. Thus a class being pointed to by a filter may be deleted, which is not the desired behavior.

3.5.9 Unbind_tcf

The unbind_tcf function is used to remove an instance of a filter attached to a class. The cbq_unbind_filter function in net/sched/sch_cbq.c is an example for the unbind_tcf function on a class.

```
static void cbq_unbind_filter(struct Qdisc *sch, unsigned long arg)
{
```

```
    struct cbq_class *cl = (struct cbq_class*)arg;

    cl->filters--;
}
```

This portion of the code clearly indicates that the unbind_tcf function is similar to the put function on a class. The count of the number of filters attached to the class is decremented. For a class to be deleted, this count must be zero.

3.5.10 Dump_class

The dump_class function on a class is used to dump diagnostic data about the class. There is a lot of data that is maintained about the classes and the dump function is used to observe these values. There is nothing more that needs to be elaborated on the dump_class function.

When the enqueue function of a queuing discipline is invoked, the tc_classify function in include/net/pkt_cls.h is called to determine the class to which the packet belongs. The simplest form of classification possible is the specification of the skb->priority field (in struct sk_buff in include/linux/skbuff.h). If skb->priority is specified, then no other classification is attempted. skb->priority is set to sk->priority (in include/net/sock.h) when the packet is created. The value of sk->priority can be specified with the help of the setsockopt call. The SO_PRIORITY option in the setsockopt call needs to be used for this purpose. However till linux kernel version 2.2.3, the value of sk->priority is limited between 0 and 7. Therefore this way of classifying packets does not work. It is worth mentioning at this point that skb->priority may contain other values like the TOS byte in the IP header. All these values are less than 65536, which is the smallest valid class number (as the minimum possible value for the major number of a class is 1). On selecting the class, the enqueue function of the queuing discipline owned by this class is invoked.

In the next section, we will look into the details involved in the implementation of filters.

3.6 Filters

Filters are used to classify packets based on certain properties of the packet, e.g., TOS byte in the IP header, IP addresses, port numbers etc. It is in-

voked when the enqueue function of a queuing discipline is invoked. Queuing disciplines use filters to assign the incoming packets to one of its classes.

Filters can be maintained per class or per queuing discipline based on the design of the queuing discipline. As already mentioned in the previous section, filters are maintained in filter lists. The filter list is specified as a struct tcf_proto, in include/net/pkt_cls.h.

```
struct tcf_proto
{
 /* Fast access part */
 struct tcf_proto        *next;
 void                    *root;
 int                     (*classify)(struct sk_buff*, struct tcf_proto*,
                         struct tcf_result *);
 u32                     protocol;

 /* All the rest */
 u32                     prio;
 u32                     classid;
 struct Qdisc            *q;
 void                    *data;
 struct tcf_proto_ops    *ops;
};
```

This structure is used to represent filter lists and is maintained by the classes and queuing disciplines. As an example, the cbq_class structure in net/sched/sch_cbq.c maintains the filter list by using the tcf_proto structure. The tcf_chain function on classes, described in the previous section, is used to return the anchor to a filter list, which can be used to traverse the filter list. 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. The protocol numbers are used in skb->protocol and they are defined in include/linux/if_ether.h.

Filters may also have an internal structure: it may control internal elements, which are referenced by a handle. These handles are 32-bit long, but are not divided into major and minor numbers like class IDs. Handle 0 refers to the filter itself. Like classes, filters also have an internal ID, which can

be obtained with the help of a get function. The basic structure of filters is shown in Figure 4.

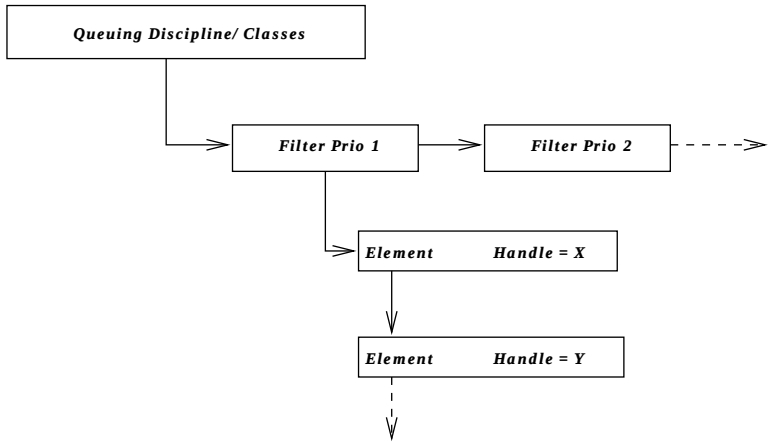

Figure 4: Structure of Filters

When the enqueue function of a queuing discipline is invoked, the tc_classify function in include/net/pkt_cls.h is invoked to classify the packet.

```
extern __inline__ int tc_classify(struct sk_buff *skb, struct tcf_proto *tp,
struct tcf_result *res)
{
 int err = 0;
 u32 protocol = skb->protocol;
 for ( ; tp; tp = tp->next) {
   if ((tp->protocol == protocol ||
     tp->protocol == __constant_htons(ETH_P_ALL))
     && (err = tp->classify(skb, tp, res)) >= >0)
        return err;
 }
 return -1;
}
```

As seen above in this function, the protocol to which the packet belongs to is determined from skb->protocol. Once this is obtained, the filters corresponding to this protocol are all applied in the order of priority. Within each filter all the internal elements are traversed in an attempt to classify

the packet. Once the packet is classified, as already mentioned, the enqueue function of the queuing discipline owned by the class is invoked. This process of obtaining a match for the packet is shown in Figure 5.

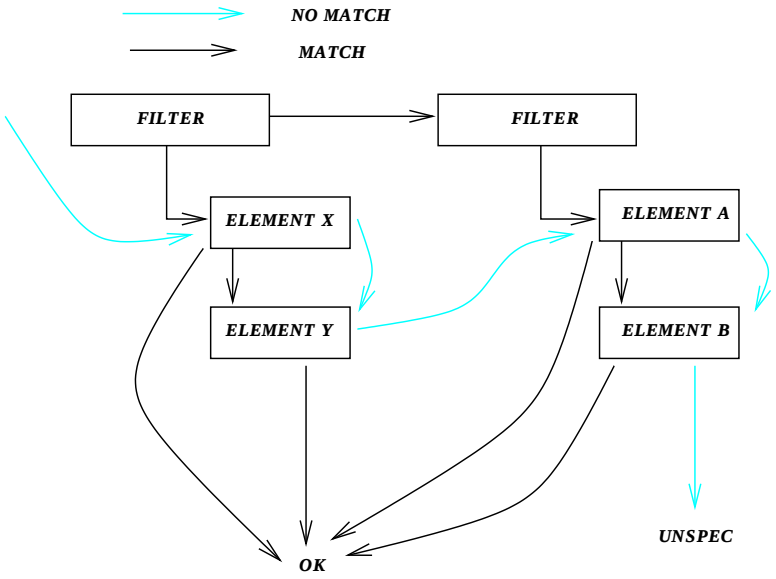

Figure 5: Matching a filter

Let us now discuss the functions that can be performed on the filters. The functions are defined in the tc_proto_ops structure in include/net/pkt_cls.h.

3.6.1 Classify

The classify function on a filter is used to match a packet to a class based on certain properties of the packet. The result of any classification can be one of four return values from the policing function tcf_police in net/sched/police.c, namely TC_POLICE_UNSPEC, TC_POLICE_OK, TC_POLICE_RECLASSIFY or TC_POLICE_SHOT. The policer is discussed in more detail in the next section. At this point, it is enough to understand that the tc_classify function returns TC_POLICE_UNSPEC when no matching filter is found for the packet. If not, the classifier fills in the tcf_result structure (defined in include/net/pkt_cls.h) and returns it. The tcf_result structure contains the internal ID as well as the class ID of the class to which the packet belongs. As an example, let us look into the route_classify function in net/sched/cls_route.c.

```
static int route_classify(struct sk_buff *skb, struct tcf_proto *tp,
                          struct tcf_result *res)
{
 struct dst_entry *dst = skb->dst;

 if (dst) {
  u32 clid = dst->tclassid;

  if (clid && (TC_H_MAJ(clid) == 0 ||
    !(TC_H_MAJ(clid^tp->q->handle)))) {
        res->classid = clid;
        res->class = 0;
        return 0;
    }
  }
  return -1;
}
```

The route classifier classifies packets based on the destination IP address. In the route_classify function, the destination is associated with a dst_entry structure (in include/net/dst.h). The class ID of the class is stored in the tclassid field of the dst_entry structure. If the destination to which the packet needs to be sent is determined, the corresponding class ID is returned in the tcf_result structure. In general, the filters are also associated with policers, to determine if a flow is in profile. Having discussed the classify function, let us now discuss the init function of a filter.

3.6.2 Init

The init function on a filter is used to initialize the parameters for a filter. As an example, let us take a look at the tcindex_init function in net/sched/cls_tcindex.c, which is used to initialize the parameters for a tcindex classifier.

```
static int tcindex_init(struct tcf_proto *tp)
{
  struct tcindex_data *p;

  p = kmalloc(sizeof(struct tcindex_data),GFP_KERNEL);
  if (!p) {
```

```
    return -ENOMEM;
  }
  tp->root = p;
  memset(p->h,0,sizeof(p->h));
  p->mask = 0xffff;
  p->shift = 0;
  p->fall_through = 1;
  return 0;
}
```

The tcindex_data structure, the definition of which is available in the same file, is initialized in the tcindex_init function. The mask and the shift, which are used in combination to determine a handle are set to 0xffff and 0 respectively. The details of the tcindex classifier are discussed in more detail later. The init function is invoked from the tc_ctl_tfilter function in net/sched/cls_api.c, which is called whenever a filter is added, deleted or changed.

3.6.3 Destroy

The destroy function on a filter is used to remove a filter. The cbq_destroy function which was discussed previously, also results in the destroy function on the filters being called. If the filter or any of its elements are registered with classes, the destroy function on a filter calls the unbind_tcf function to de-register from these classes. The unbind_tcf function was discussed in the previous section on classes. It also removes any policer that had been attached to filter. Let us look at the tcindex_destroy function in net/sched/cls_tcindex.c as an example.

```
static void tcindex_destroy(struct tcf_proto *tp)
{
 struct tcindex_data *p = PRIV(tp);
 struct tcindex_filter *f;
 int i;

 for (i = 0; i < HASH_SIZE; i++)
  while (p->h[i]) {
  unsigned long cl;
```

```
  f = p->h[i];
  p->h[i] = f->next;
  cl = xchg(&f->res.class,0);
  if (cl) tp->q->ops->cl_ops->unbind_tcf(tp->q,cl);
#ifdef CONFIG_NET_CLS_POLICE
  tcf_police_release(f->police);
#endif
  kfree(f);
  }
 kfree(p);
 tp->root = NULL;
}
```

This function shows that the destroy function on a filter unbinds itself from all the classes to which it was bound (using bind_tcf function on the class) and removes the policer that was attached to it. The destroy function takes the tcf_proto structure as an input to determine the filter that needs to be deleted and to determine the classes and queues to which it is attached. It then frees the memory that is occupied by the filter. For more complicated filters like the u32, the destroy function is a lot more complicated than this. The destroy function is invoked from the tc_ctl_tfilter function in net/sched/cls_api.c.

3.6.4 Get

As already mentioned, every filter has an internal ID corresponding to the handle. This mapping can be obtained with the help of the get function on the filter. This is similar to the get function on a class. The tcf_result structure, which is used in the case of classes, is also used in the case of filters. As an example, let us look at the tcindex_get function in net/sched/cls_tcindex.c.

```
static unsigned long tcindex_get(struct tcf_proto *tp, u32 handle)
{
 DPRINTK("tcindex_get(tp %p,handle 0x%08x)\n",tp,handle);
 return (unsigned long) lookup(PRIV(tp),handle);
}

static struct tcindex_filter *lookup(struct tcindex_data *p,__u16 key)
```

```
{
 struct tcindex_filter *f;

 for (f = p->h[HASH(key)]; f; f = f->next)
  if (f->key == key) break;
 return f;
}
```

This tcindex_get function takes as input the 32-bit handle and a pointer to the tcf_proto structure, which contains the the information about the filter. It then calls the lookup function, which walks through the filter list to determine the filter with the specified handle. The lookup function returns the corresponding tcindex_filter structure, which contains the tcf_result structure as a member. The get function on a filter is invoked from the tc_ctl_tfilter function in net/sched/cls_api.c. If the tc_ctl_tfilter function is invoked to delete a filter, the get function returns the internal ID, which can then be used to call the destroy function on a filter.

3.6.5 Put

The put function on a class is used to de-reference a filter that was previously referenced using the get function. But in general, the put function is never invoked. A look at the classifier files (files in net/sched/ starting with the prefix cls) will indicate this. Thus, this function need not be discussed any further.

3.6.6 Change

The change function on a filter is used to change the properties of a filter. This is similar to the change function on classes and queuing disciplines. The configuration parameters are passed using a mechanism that is similar to the way the parameters are passed for classes and queuing disciplines. When the change function is invoked on a filter, if new elements are added to the filter, or if a new filter is added to a class, the unbind_tcf function is called to remove the binding between the class and the filter, which is then followed by the bind_tcf function on the class to bind the filter with new properties to the class. If any policer is attached to the filter, then its properties are also modified. As an example, let us take a look at the tcindex_change function in net/sched/cls_tcindex.c.

```
static int tcindex_change(struct tcf_proto *tp,u32 handle,struct rtattr **tca,
unsigned long *arg)
{
.
.
if (rtattr_parse(tb,TCA_TCINDEX_MAX,RTA_DATA(opt),RTA_PAYLOAD(opt)) < 0)
   return -EINVAL;
if (tb[TCA_TCINDEX_MASK-1]) {
.
.
.
if (tb[TCA_TCINDEX_CLASSID-1]) {
  unsigned long cl = xchg(&f->res.class,0);

  if (cl)
   tp->q->ops->cl_ops->unbind_tcf(tp->q,cl);
  f->res.class = tp->q->ops->cl_ops->bind_tcf(tp->q, f->res.classid);
.
.
  if (tb[TCA_TCINDEX_POLICE-1]) {
   struct tcf_police *police = tcf_police_locate(tb[TCA_TCINDEX_POLICE-1]);
   tcf_police_release(xchg(&f->police,police));
  }
.
.

}
```

This function shows how a change function on a filter is processed. First, the properties associated with the filter are changed. This is followed by unbinding and binding the filter to the classes to which it is attached. Finally, the policer attached to the filter is also modified. The change function on a filter is invoked from the tc_ctl_tfilter function in net/sched/cls_api.c.

3.6.7 Delete

The delete function on a filter is used to delete a particular element of the filter. As was discussed previously, to delete the entire filter, the destroy

function on the filter is invoked. As in the case of the destroy function, the delete function results in an unbind_tcf function being called on the class to which the element is attached. The policer attached to the element is also removed. As an example, let us take a look at the tcindex_delete function in net/sched/cls_tcindex.c.

```
static int tcindex_delete(struct tcf_proto *tp, unsigned long arg)
{
.
.
 for (walk = p->h+HASH(f->key); *walk && *walk != f;
  walk = &(*walk)->next);
 if (!*walk) return -ENOENT;
 *walk = f->next;
 cl = xchg(&f->res.class,0);
 if (cl) tp->q->ops->cl_ops->unbind_tcf(tp->q,cl);
.
 tcf_police_release(f->police);
.
.
}
```

This function searches for the internal element in a filter by walking through the filter, and after determining this, makes the class 0 (the xchg function does this) and calls the unbind_tcf function to detach itself from the class. After this, it also releases the policer, if any, that is attached to the element. The distinction between the delete on an element and a destroy on a filter is made in the tc_ctl_tfilter function. The following portion of code from tc_ctl_tfilter will make this evident:

```
fh = tp->ops->get(tp, t->tcm_handle);

if (fh == 0) {
 if (n->nlmsg_type == RTM_DELTFILTER && t->tcm_handle == 0) {
  *back = tp->next;
  tp->ops->destroy(tp);
  kfree(tp);
  err = 0;
  goto errout;
```

```
}
.
.

switch (n->nlmsg_type) {
.
.

 case RTM_DELTFILTER:
      err = tp->ops->delete(tp, fh);
      goto errout;

}
.
.
```

The get function is used to obtain the internal ID for the filter. As mentioned earlier in this section, a filter has a handle of zero, while the internal elements are identified by the handles. If the internal ID returned by the get function is zero, and if the DELETE option is specified, then it can be concluded that a filter needs to be destroyed. If the handle returned is not zero, then it can be concluded that a particular element of the filter needs to be deleted.

3.6.8 Walk

The walk function on a filter is used to iterate over all the elements of a filter and invokes a callback function for each of the elements. This is usually used to obtain diagnostic data for all the elements of a filter. For example, let us take a look at the tcindex_walk function in net/sched/cls_tcindex.c.

```
static void tcindex_walk(struct tcf_proto *tp, struct tcf_walker *walker)
{
 struct tcindex_data *p = PRIV(tp);
 struct tcindex_filter *f;
 int i;
```

```
 for (i = 0; i < HASH_SIZE; i++)
  for (f = p->h[i]; f; f = f->next) {
   if (walker->count >= walker->skip)
    if (walker->fn(tp,(unsigned long) f,walker) < 0) {
      walker->stop = 1;
      return;
    }
    walker->count++;
  }
}
```

This portion of the code shows that the walk command iterates over all the filters of a specified filter and invokes a callback function. The walk function is called from the tc_dump_tfilter function in net/sched/cls_api.c, which is invoked when a dump request is made on all the filters. The portion of the code in tc_dump_tfilter that does this shown below:

```
static int tc_dump_tfilter(struct sk_buff *skb, struct netlink_callback *cb)
{
struct tcf_proto *tp, **chain;
.
for (tp=*chain, t=0; tp; tp = tp->next, t++) {
.
.
 if (tp->ops->walk == NULL) continue;
 tp->ops->walk(tp, &arg.w);
}
.
.
```

The callback function maps to the tcf_node_dump function in the same file. In tcf_node_dump, the tcf_fill_node function is invoked, which calls the dump function on all the filters. The dump function on a filter, which is discussed later is used to dump statistical information about the filter.

3.6.9 Dump

The dump function on a filter is used to dump diagnostic data about the filter and one or more of its elements. There is a lot of data that is maintained

about the filters and the dump function is used to obtain these values. There is nothing more that needs to be elaborated on the dump function on a filter.

Having discussed about the various operations that can be performed on filters, let us now discuss the different types of filters supported. Filters can be classified into generic filters and specific filters based on the scope of the packets their instances can classify.

Generic filters need only one instance of the filter per queuing discipline to classify packets for all classes. The route classifier is an example of a generic classifier. The route_classify function in net/sched/cls_api.c takes the class ID from the packet, where it was stored by another entity in the protocol stack. As far as the route classifier is concerned, this entity is the routing functionality in net/ipv4/route.c. The rt_set_nexthop function in this file, which is used to set the next hop for a particular destination address, also sets the class ID for the packets sent to this destination. This can be set from the user space using the ip tool, which will be discussed later in this document. Generic filters are explained in Figure 6.

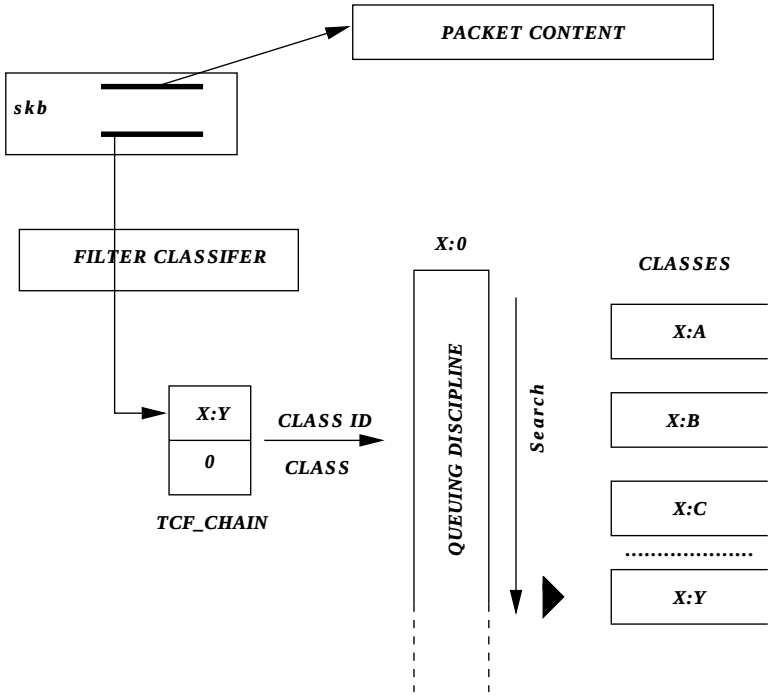

Figure 6: Generic Filters

Specific filters need one or more instances of a filter or its internal elements per class to identify packets belonging to this class. Multiple instances of a filter (or its elements) on the same filter list (which may potentially map to the same class) are distinguished based on the internal IDs. Since the specific filters have at-least one instance of the filter per class, they can store the internal ID of the class in the tcf_proto structure, thereby ensuring a fast lookup of the class. It is here that the specific filters score over the generic filters. In the case of generic filters, the tcf_result is returned with the class field (that is, the internal class ID) set to zero. The queuing discipline is responsible for doing another lookup to determine the class ID (as shown in Figure 6). Specific filters are described in Figure 7.

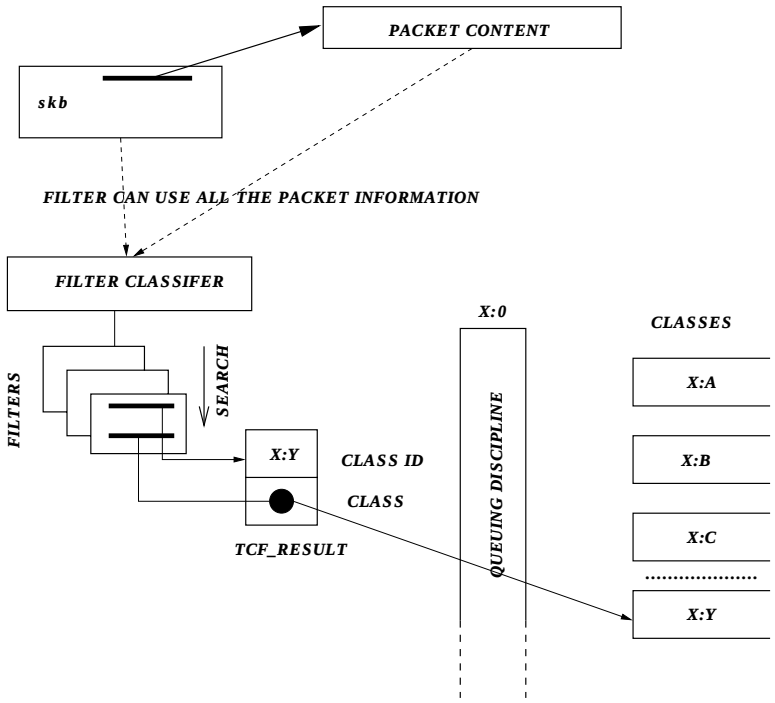

Figure 7: Specific Filters

This concludes the details involved in the implementation of filters. Before discussing the user level tools available to make use of the kernel features, let us take a brief look at the interface between the kernel and the user space.

3.7 Interface between the kernel and user space

The interface between the kernel and the user space is achieved using netlink sockets. Netlink sockets are described in more detail in later sections. For now, let us look at the sequence of steps involved in executing a command given in the user space.

The interfaces between the kernel traffic elements and the user space programs are defined in include/linux/pkt_cls.h and include/linux/pkt_sched.h. The pkt_sched.h file will specify the parameters that are of significance in each type of queue. rtnetlink is used to exchange traffic control objects between the user level and the kernel level. This is specified in the net/core/rtnetlink.c and linux/include/rtnetlink.h. rtnetlink is based on netlink. The netlink socket uses the sockaddr_nl address structure. This is the structure that is used by the user level code to communicate with the kernel. The code for the netlink is in net/netlink/.

```
struct sockaddr_nl
{
        sa_family_t     nl_family;      /* AF_NETLINK   */
        unsigned short  nl_pad;         /* zero         */
        __u32           nl_pid;         /* process pid  */
        __u32           nl_groups;      /* multicast groups mask */
};

struct nlmsghdr
{
        __u32           nlmsg_len;      /* Length of message including header
*/
        __u16           nlmsg_type;     /* Message content */
        __u16           nlmsg_flags;    /* Additional flags */
        __u32           nlmsg_seq;      /* Sequence number */
        __u32           nlmsg_pid;      /* Sending process PID */
};
```

The message format used to transmit traffic control messages to the kernel is shown in Figure 8. The messages are stored at byte boundaries. The length of the message includes the message header as well.

If traffic control is enabled in the linux kernel, at boot time, the __initfunc function in net/core/dev.c is called. This in turn invokes the pktsched_init

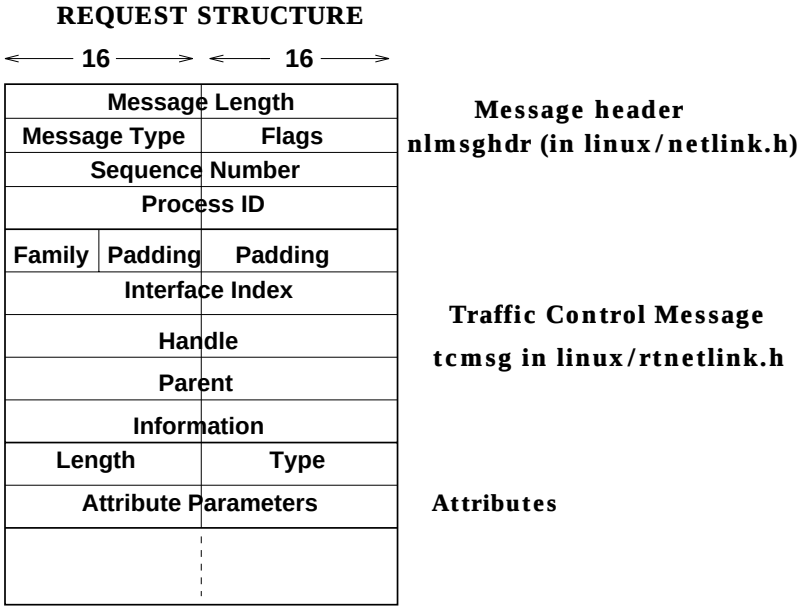

Figure 8: tc to kernel - Message formats

function in net/sched/sch_api.c. This function initiates various declarations and bindings. One of the most important initializations that is done here is shown below:

```
struct rtnetlink_link *link_p;

if (link_p) {
 link_p[RTM_NEWQDISC-RTM_BASE].doit = tc_ctl_qdisc;
 link_p[RTM_DELQDISC-RTM_BASE].doit = tc_ctl_qdisc;
 link_p[RTM_GETQDISC-RTM_BASE].doit = tc_ctl_qdisc;
 link_p[RTM_GETQDISC-RTM_BASE].dumpit = tc_dump_qdisc;
 link_p[RTM_NEWTCLASS-RTM_BASE].doit = tc_ctl_tclass;
 link_p[RTM_DELTCLASS-RTM_BASE].doit = tc_ctl_tclass;
 link_p[RTM_GETTCLASS-RTM_BASE].doit = tc_ctl_tclass;
 link_p[RTM_GETTCLASS-RTM_BASE].dumpit = tc_dump_tclass;
}
```

This fills in the pointers to the various functions that need to be called based on actions (i.e., add, delete, change etc) and entities (i.e., queuing disciplines, classes and filters) specified at the user level. This function also

registers the various queuing disciplines that are supported during the kernel configuration.

A netlink socket is created from the user level application to the kernel, in order to send configuration messages, which will be interpreted and executed by the kernel. When the user issues a specific action on a specific entity, a sendto is done on the netlink socket. This results in the netlink_sendmsg function in net/netlink/af_netlink.c being invoked. The rtnetlink_rcv_msg function in net/core/rtnetlink.c receives the messages sent from the user space. In this function, the message header is examined (nlmsghdr) to determine the type of the message. The message type could be RTM_NEWQDISC, RTM_DELQDISC etc. Based on the message type, the corresponding function in rtnetlink_link is invoked (either doit or dumpit, which point to appropriate function). These are the steps involved in executing a command that is specified at the user level.

3.8 Usage - tc

'tc' (traffic controller) is the user level program that can be used to create and associate queues with the output devices. It is used to set up various kinds of queues and associate classes with each of those queues. It can also be used to set up filters based on the routing table, u32 classifiers, tcindex classifiers and RSVP classifiers. As already mentioned, it uses netlink sockets as a mechanism to communicate with the kernel networking functions.

The usage for tc is :

```
tc [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { qdisc | class | filter }
       OPTIONS := { -s[statistics] | -d[details] | -r[raw] }
```

The Object could be a queuing discipline, class or a filter. Let us discuss the general usage of queuing disciplines,classes and filters.

3.9 Queuing Disciplines

The syntax for creating a queuing discipline is:

```
tc qdisc [ add | del | replace | change | get ] dev STRING
       [ handle QHANDLE ] [ root | parent CLASSID ]
       [ estimator INTERVAL TIME_CONSTANT ]
```

```
       [ [ QDISC_KIND ] [ help | OPTIONS ] ]

       tc qdisc show [ dev STRING ]
Where:
QDISC_KIND := { [p|b]fifo | tbf | prio | cbq | red | etc. }
```

The interpretation for the fields:

- handle represents the unique handle that is assigned by the user to the queuing discipline. No two queuing disciplines can have the same handle.
- root indicates that the queue is at the root of a link sharing hierarchy.
- parent represents the handle of the parent queuing discipline.
- 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 way these parameters are set is described in [3].

3.10 Classes

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. }
```

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.

3.11 Filters

The syntax for creating a filter is shown below:

```
tc filter [ add | del | change | get ] dev STRING
       [ prio PRIO ] [ protocol PROTO ]
       [ root | classid CLASSID ] [ handle FILTERID ]
       [ [ FILTER_TYPE ] [ help | OPTIONS ] ]

       tc filter show [ dev STRING ] [ root | parent CLASSID ]
Where:
FILTER_TYPE := { rsvp | u32 | fw | route | etc. }
FILTERID := ... format depends on classifier
```

The interpretation of the fields:

- prio 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.

Having discussed the general syntax for creating, deleting and changing queuing discipline, classes and filters, let us now take a look at the various queuing disciplines that currently supported in linux.

3.12 Class Based Queue

This section discusses Class Based Queues in detail. The terms commonly used in the CBQ context and the user-level syntax to set up these queues are discussed in this section.

Let us first define some basic terms in CBQ. 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.

1. 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.

2. avgidle: The variable avgidle is the average of idle, and it computed using an exponential weigted 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).

3. 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.

4. 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.

5. 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.

There are three types of classes, namely leaf classes (such as a video class) that have directly assigned connections; nonleaf classes used for link-sharing; and the root class that represents the entire output link.

The syntax to create a CBQ is shown below:

```
tc qdisc [ add | del | replace | change | get ] dev STRING  \
cbq bandwidth BPS [ avpkt BYTES ] [ mpu BYTES ] [ cell BYTES ] [ ewma LOG ]
```

The interpretation of the fields:

- 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

$$TransmissionTimet = \frac{averagepacketsize}{LinkBandwidth}$$

- 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.

For e.g.
tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit allot 1514 cell 8 avpkt 1000 mpu 64

In the above example, a class based queue is created and attached to device eth0. The handle for the queue is 1: (that is, 1:0), where 1 represents the major number and 0 represents the minor number. The bandwidth available on the outgoing link is 10 Mbit. allot is a parameter that is used by the link sharing scheduler. A cell value of 8 indicates that the packet transmission time will be measured in terms of 8 bytes.

Let us now discuss the syntax for creating a class for a CBQ.

```
tc qdisc [add | del | replace | change] 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 ]
```

The interpretation of the fields:

- bandwidth represents the maximum bandwidth that is available to the queuing discipline owned by this class.

- rate represents the bandwidth that is allocated to this class. The kernel does not use this directly. It uses pre-calculated rate translation tables.
- 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.
- isolated indicates that the class will not share bandwidth with any of non-descendant classes
- allot, cell, mpu, estimator and ewma have already been explained.
- weight should be made proportional to the rate.
- The spilt 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.
- prio represents the priority that is assigned to this class.
- This again, is concerned with classification. It is intended to make fallback classification. When a packet does not match any classifier, this fallback classification is used. This is done in the following manner. The TOS byte in the incoming packets or the SO_PRIORITY in the locally generated packets is used as a logical priority. If a class is ready to serve a logical priority, the defmap option is used. If a packet matches a classifier, this logical priority is not used.

For e.g.
tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20 avpkt 1000 split 1:0 defmap c0

In this example, a CBQ class with handle 1:2 is created. Its parent is identified by the handle 1:1. The priority assigned to it is 3, the average packet size is 1000 bytes. The split node is 1:0, which represents the root of the link sharing structure. The defmap is c0, that is, packets with this TOS (for incoming packets) or SO_PRIORITY (for locally generated packets) that DO NOT classify under any class are considered to belong to the class with handle 1:2.

3.12.1 Route Classifiers

Route classifiers classify the packets based on the the routing table. This involves the use of another tool, namely the IP tool (IPT). This has been described comprehensively in [2]. The syntax for creating a route classifier:

```
tc filter [add | del | change | get] dev STRING
[parent PARENTID] [protocol PROTO]
[prio PRIORITY]  route

Where:
PROTO = {ip | icmp | etc.}
```

All the fields above have already been explained. The keyword route indicates that it is a route classifier. All the packets at the specified device will be classified based on the routing table. A set of rules are specified that indicate how the packets need to be treated. This is discussed in [2].

For e.g.
To install a route classifier:

tc filter add dev eth0 parent 1:0 protocol ip prio 100 route

In the above example, a route classifier is attached to the root of a CBQ tree. The classifier will classify IP packets (as indicated by the protocol field) and the priority assigned to it 100.

To specify rules to the filter:
ip route add 129.237.125.150 via 129.237.125.146 dev eth0 flow 1:2

In this example, a route is added to 129.237.125.150 via 129.237.125.146 and all such traffic will be considered as belonging to the class whose handle is 1:2.

3.12.2 u32 Classifiers

To classify based on individual application requirements, a more powerful classification scheme (than route classification) is needed. u32 classifiers are used to do this. Classification can be done based on the destination IP address, destination TCP/UDP port, source IP address, source TCP/UDP port, TOS byte and protocol. A more detailed description of this classifier will be done soon.

4 Example - 1 : CBQ with route classifiers

Let us consider a very simple scenario. Lets set up a CBQ for an ethernet device on testbed14 (129.237.125.149). Lets assume that the average packet size is 1000, a cell value of 8 bytes and a maximum burst size of 20 bytes. Lets have two types of classes, one for traffic to the machine 129.237.125.146 (testbed11) and for the traffic to machine 129.237.125.148 (testbed13). The traffic to testbed11 is given a higher priority over the traffic to testbed 13. However, testbed11 requires a bandwidth of only 3 Mbps while testbed13 requires a bandwidth of 7 Mbps.

This example explains classification based on the routing table. The environment set up is shown in Figure 9.

To set up this environment in testbed14.

```
# Attaching the Qdisc to the eth0 device.  The maximum available bandwidth is
# 10Mbit.
tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit cell 8 avpkt 1000 \
mpu 64

# Adding the root class to the queuing discipline. The root has 10 Mbit
# completely.
tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit rate 10Mbit \
allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20 avpkt 1000

# Traffic to testbed11. The priority is 3 and the allocation is 3 Mbit.
```

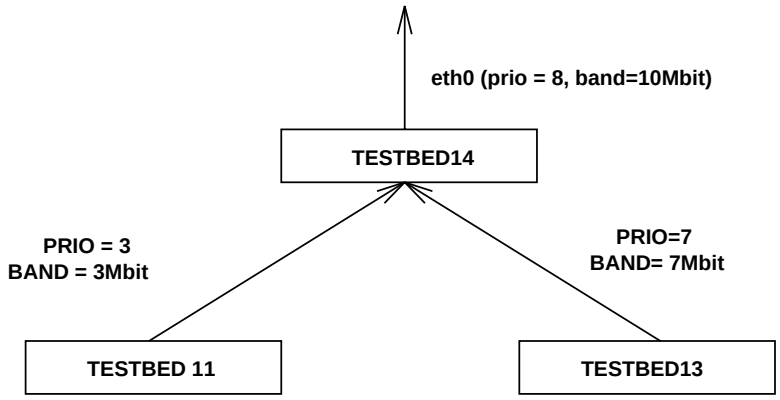

Figure 9: Linux QoS Support - Example 1

```
tc class add dev eth0 parent 1:1 classid 1:2 cbq bandwidth 10Mbit rate 3Mbit \
allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20 avpkt 1000 split 1:0

# Traffic to testbed13. The priority is 7 and the allocation is 7 Mbit.
tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 10Mbit rate 8Mbit \
allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20 avpkt 1000 split 1:0

# Installing the route classifier on the root of the tree.
tc filter add dev eth0 parent 1:0 protocol ip prio 100 route

# Assigning the route and the rules for testbed11
ip route add 129.237.125.146 via 129.237.125.149 flow 1:2

# Assigning the route and the rules for testbed13
ip route add 129.237.125.148 via 129.237.125.149 flow 1:3
```

Note that none of the classes are specified as bounded which means that the classes can borrow bandwidth from the parent.

5 Example - 2 : CBQ with u32 classifier

Let us now consider a slightly more complicated scenario. Lets set up a CBQ for an ethernet device on testbed14 (129.237.125.149). Lets again assume that the average packet size is 1000, a cell value of 8 bytes and a maximum

burst size of 20 bytes. Let us again have two types of classes, one for traffic to the machine 129.237.125.146 (testbed11) and the other for the traffic to machine 129.237.125.148 (testbed13). testbed11 requires a bandwidth of 3 Mbps and a priority of 3 while testbed13 requires a bandwidth of 7 Mbps and a priority of 7.

Now let us assume that the two classes in turn have two more classes each. The class setup for traffic to testbed11 is in turn classified into two classes, one with a priority of 1, a bandwidth of 2 Mbits and using port 6010 in testbed11, and the other with a priority of 2 and a bandwidth of 1 Mbit and using port 6011 on testbed11.

Similarly, the traffic to testbed13 is classified into two classes, one with a priority 1, a bandwidth of 2 Mbits and using port 6021 in testbed13, and the other with a priority of 2 and a bandwidth of 1 Mbit and using port 6031 on testbed13.

Also, let us assume that the testbed14 enforces the bandwidth sharing in a very strict sense. That is, none of the children are allowed to borrow unused bandwidth.

This example explains classification based on u32 classifiers. The set up is show in Figure 10.

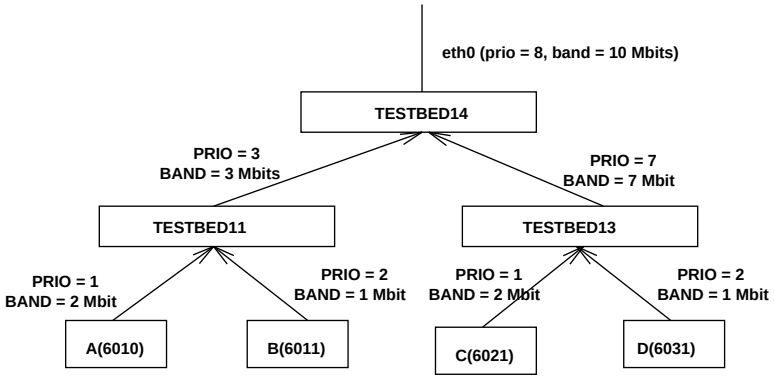

Figure 10: Linux QoS Support - Example 2

To set up this environment in testbed14

```
# Attaching the Qdisc to the eth0 device.  The maximum available bandwidth is
# 10Mbit.
tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit allot 1514 cell 8
```

```
avpkt 1000 mpu 64

# Adding the root class to the queuing discipline. The root has 10 Mbit
# completely.
tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit rate
10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20 avpkt 1000


# Traffic to testbed11. The priority is 3 and the allocation is 3 Mbit. Note
# that it is set to bounded because of the strict link sharing rules.
tc class add dev eth0 parent 1:1 classid 1:2 cbq bandwidth 10Mbit rate 3Mbit
allot 1514 cell 8 weight 300Kbit prio 3 maxburst 20 avpkt 1000 bounded

# Traffic to testbed13. The priority is 7 and the allocation is 7 Mbit.  Note
# that it is set to bounded because of the strict link sharing rules.
tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 10Mbit rate 7Mbit
allot 1514 cell 8 weight 700Kbit prio 7 maxburst 20 avpkt 1000 bounded

# Attaching another CBQ to the traffic to testbed11. This is meant for further
# classification.
tc qdisc add dev eth0 parent 1:2 handle 2: cbq bandwidth 3Mbit allot 1514
cell 8 avpkt 1000 mpu 64

# Attaching another CBQ to the traffic to testbed13. This is meant for further
# classification.
tc qdisc add dev eth0 parent 1:3 handle 3: cbq bandwidth 7Mbit allot 1514
cell 8 avpkt 1000 mpu 64

# Setting up the classes for the traffic to testbed11 appropriately.
tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 3Mbit rate 3Mbit
allot 1514 cell 8 weight 300Kbit prio 3 maxburst 20 avpkt 1000
tc class add dev eth0 parent 2:1 classid 2:2 cbq bandwidth 3Mbit rate 2Mbit
allot 1514 cell 8 weight 200Kbit prio 1 maxburst 20 avpkt 1000
tc class add dev eth0 parent 2:1 classid 2:3 cbq bandwidth 3Mbit rate 1Mbit
allot 1514 cell 8 weight 100Kbit prio 2 maxburst 20 avpkt 1000

# Setting up the classes for the traffic to testbed13 appropriately.
tc class add dev eth0 parent 3:0 classid 3:1 cbq bandwidth 7Mbit rate 7Mbit
```

```
allot 1514 cell 8 weight 700Kbit prio 7 maxburst 20 avpkt 1000
tc class add dev eth0 parent 3:1 classid 3:2 cbq bandwidth 7Mbit rate 5Mbit
allot 1514 cell 8 weight 500Kbit prio 1 maxburst 20 avpkt 1000
tc class add dev eth0 parent 3:1 classid 3:3 cbq bandwidth 7Mbit rate 2Mbit
allot 1514 cell 8 weight 200Kbit prio 2 maxburst 20 avpkt 1000


# Installing a route classifier on device eth0.
tc filter add dev eth0 parent 1:0 protocol ip prio 100 route

# Setting up the routes to testbed11 and testbed13 and the rules for this
# traffic.
ip route add 129.237.125.146 via 129.237.125.149 flow 1:2
ip route add 129.237.125.148 via 129.237.125.149 flow 1:3

# Installing a u32 classifier for the traffic to testbed11
tc filter add dev eth0 parent 2:0 prio 3 protocol ip u32

# Creating  hash tables for classification of the packets to testbed11
tc filter add dev eth0 parent 2:0 prio 3 handle 1: u32 divisor 256
tc filter add dev eth0 parent 2:0 prio 3 handle 2: u32 divisor 256

# Configuring the 6th slot of the hash table 1 to select packets with
# destination set to testbed11 and the port set to 6010 (0x177a) and direct
# these to class 2:2 (which was just set up).
tc filter add dev eth0 parent 2:0 prio 3 u32 ht 1:6: match ip dst
129.237.125.146 match tcp dst 0x177a 0xffff flowid 2:2

# Configuring the 6th slot of the hash table 2 to select packets with
# destination set to testbed11 and the port set to 6011 (0x177b) and direct
# these to class 2:3 (which was just set up).
tc filter add dev eth0 parent 2:0 prio 3 u32 ht 2:6: match ip dst
129.237.125.146 match tcp dst 0x177b  0xffff flowid 2:3

# Lookup hash table and it it is not fragmented, use the protocol as the hash
# key.
tc filter add dev eth0 parent 2:0 prio 3 handle ::1 u32 ht 800:: match ip
nofrag offset mask 0x0F00 shift 6 hashkey mask 0x00ff0000 at 8 link 1:
```

```
tc filter add dev eth0 parent 2:0 prio 3 handle ::1 u32 ht 800:: match ip
nofrag offset mask 0x0F00 shift 6 hashkey mask 0x00ff0000 at 8 link 2:

# Installing a u32 classfier for the traffic to testbed13
tc filter add dev eth0 parent 3:0 prio 7 protocol ip u32

# Creating  hash tables for classification of the packets to testbed13
tc filter add dev eth0 parent 3:0 prio 7 handle 3: u32 divisor 256
tc filter add dev eth0 parent 3:0 prio 7 handle 4: u32 divisor 256

# Configuring the 6th slot of the hash table 3 to select packets with
# destination set to testbed13 and the port set to 6021 (0x1785) and direct
# these to class 3:2 (which was just set up).
tc filter add dev eth0 parent 3:0 prio 7 u32 ht 3:6: match ip dst
129.237.125.148 match tcp dst 0x1785 0xffff flowid 3:2

# Configuring the 6th slot of the hash table 4 to select packets with
# destination set to testbed13 and the port set to 6031 (0x178f) and direct
# these to class 3:3 (which was just set up).
tc filter add dev eth0 parent 3:0 prio 7 u32 ht 4:6: match ip dst
129.237.125.148 match tcp dst 0x178f 0xffff flowid 3:3


# Lookup hash table and it it is not fragmented, use the protocol as the hash
# key.
tc filter add dev eth0 parent 3:0 prio 7 handle ::1 u32 ht 800:: match ip
nofrag offset mask 0x0F00 shift 6 hashkey mask 0x00ff0000 at 8 link 3:
tc filter add dev eth0 parent 3:0 prio 7 handle ::1 u32 ht 800:: match ip
nofrag offset mask 0x0F00 shift 6 hashkey mask 0x00ff0000 at 8 link 4:
```

6 References

1. RFC 1633, Integrated Services
2. RFC 2475, Differentiated Services
3. Linux Traffic Control, Werner Almesberger

4. IP tool README, Alexey Kuznetsov, Jason Keimig