Differentiated Services on Linux

Werner Almesberger Werner.Almesberger@epfl.ch, EPFL ICA
Jamal Hadi Salim hadi@nortelnetworks.com, CTL Nortel Networks
Alexey Kuznetsov kuznet@ms2.inr.ac.ru, INR Moscow

June 25, 1999

Abstract

Recent Linux kernels offer a wide variety of traffic control functions, which can be combined in a modular way. We have designed support for Differentiated Services based on the existing traffic control elements, and we have implemented new components where necessary. In this document we give a brief overview of the structure of Linux traffic control, and we describe our prototype implementation in more detail.

1 Introduction

The Differentiated Services architecture (Diffserv) lays the foundation for implementing service differentiation in the Internet in an efficient, scalable way. We assume that readers are familiar with the concepts and terminology defined in [1]. Furthermore, we assume familiarity with the packet marking as described in [2].

We have developed a design to support basic classification and DS field manipulation required by Diffserv nodes. The design enables configuration of the first PHBs that are being defined in the Diffserv WG. We have implemented a prototype of this design using the traffic control framework available in recent Linux kernels. The source code, configuration examples, and related information can be obtained from `http://icawww1.epfl.ch/linux-diffserv/`

The main focus of our work is to allow maximum flexibility for node configuration and for experiments with PHBs, while still maintaining a design that does not unnecessarily sacrifice performance.

This document is structured as follows. Section 2 gives a brief overview of traffic control functions in recent Linux kernels. Section 3 discusses where the existing model needed to be extended. Section 4 describes the new components in more detail. We conclude with examples of configuration scripts in section 5.

2 Linux Traffic Control

Figure 1 shows roughly how the kernel processes data received from the network, and how it generates new data to be sent on the network.

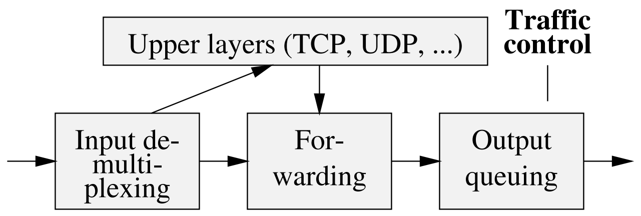

Figure 1: Processing of network data.

"Forwarding" includes the selection of the output interface, the selection of the next hop, encapsulation, etc. Once all this is done, packets are queued on the respective output interface. This is the point where traffic control comes into play. Traffic control can, among other things, decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can decide in which order packets are sent (e.g. to give priority to certain flows), it can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc.

Once traffic control has released a packet for sending, the device driver picks it up and emits it on the network.

2.1 Components

The traffic control code in the Linux kernel consists of the following major conceptual components:

- queuing disciplines
- classes (within a queuing discipline)
- filters
- policing

Each network device has a *queuing discipline* associated with it, which controls how packets enqueued on that device are treated. A very simple queuing discipline may just consist of a single queue, where all packets are stored in the order in which they have been enqueued, and which is emptied as fast as the respective device can send. See figure 2 for such a queuing discipline without externally visible internal structure.

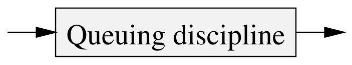

Figure 2: A simple queuing discipline without classes.

More elaborate queuing disciplines may use *filters* to distinguish among different *classes* of packets and process each class in a specific way, e.g. by giving one class priority over other classes.

Figure 3 shows an example of such a queuing discipline. Note that multiple filters may map to the same class.

Queuing disciplines and classes are intimately tied together: the presence of classes and their semantics are fundamental properties of the queuing discipline. In contrast to that, filters can be combined arbitrarily with queuing disciplines and classes as long as the queuing discipline has classes to map the packets to. But flexibility does not end there yet – classes normally do not take care of storing their packets themselves, but they use another queuing discipline to take care of that. That queuing discipline can be arbitrarily chosen from the set of available queuing disciplines, and it may well have classes, which in turn use queuing disciplines, etc. The term qdisc would be used interchangeably to mean queueing discipline in this draft.

Figure 4 shows an example of such a stack: first, there is a queuing discipline with two delay priorities. Packets which are selected by the filter go to the high-priority class, while all other packets go to the low-priority class. Whenever there are packets in the high-priority queue, they are sent before packets in the low-priority queue (e.g. the `sch_prio` queuing discipline works this way). In order to prevent high-priority traffic from starving low-priority traffic, we use a *token bucket filter* (TBF), which enforces a rate of at most 1 Mbps. Finally, the queuing of low-priority packets is done by a FIFO queuing discipline. Note that there are other ways to accomplish what we have done here, e.g. by using *class-based queuing* (CBQ).

Packets are enqueued as follows: when the `enqueue` function of a queuing discipline is called, it scans the filters until one of them indicates a match to a class identifier. It then queues the packet for the corresponding class, which usually means to invoke the `enqueue` function of the queuing discipline "owned" by that class. Packets which do not match any of the filters are typically attributed to some default class.

Typically, each class "owns" one queue, but it is in principle also possible that several classes share the same queue or even that a single queue is used by all classes of the respective queuing discipline. Note, however, that packets do not carry any explicit indication of which class they were attributed to. Queuing disciplines that change per-class information when dequeuing packets (e.g. CBQ) will therefore not work properly if the "inner" queues are shared, unless they are able either to repeat the classification or to pass the classification result from `enqueue` to `dequeue` by some other means.

Usually when enqueuing packets, the corresponding flow(s) can be policed, e.g. by discarding packets which exceed a certain rate.

3 Diffserv extensions to Linux traffic control

The traffic control framework available in recent Linux kernels [3] already offers most of the functionality required for implementing Diffserv support. We therefore closely followed the existing design and added new components only where it was deemed strictly necessary.

3.1 Overview

Figure 5 shows the general structure of the forwarding path in a Diffserv node.

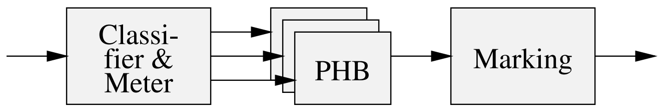

Figure 5: General Diffserv forwarding path.

Depending on the implementation, marking may also occur at different places, possibly even several times.

The classification result may be used several times in the Diffserv processing path, and it may also depend on external factors (e.g. time), so reproducing the classification result may not only be expensive, but actually impossible.

We therefore added a new field `tc_index` to the packet buffer descriptor (`struct sk_buff`), where we store the result of the initial classification. In order to avoid confusing `tc_index` with the classifier `cls_tcindex`, we will call the former `skb->tc_index` throughout this document.

`skb->tc_index` is set using the `sch_dsmark` queuing discipline, which is also responsible for initially retrieving the DSCP, and for setting the DS field in packets before they are sent on the network. `sch_dsmark` provides the framework for all other operations.

The `cls_tcindex` classifier reads all or part of `skb->tc_index` and uses this to select classes.

Finally, we need a queuing discipline to support multiple drop priorities as required for Assured Forwarding. For this, we designed GRED, a generalized RED. `sch_gred` provides a configurable number of drop priorities which are selected by the lower bits of `skb->tc_index`.

3.2 Classification and marking

The classifiers `cls_rsvp` and `cls_u32` can handle all micro-flow classification tasks. Additionally, the `ipchains` fire-

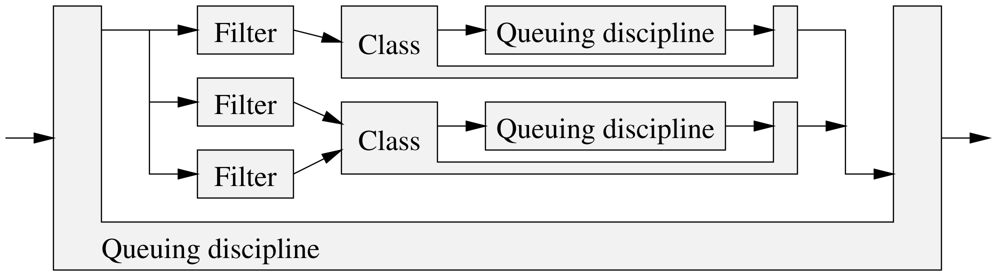

Figure 3: A simple queuing discipline with multiple classes.

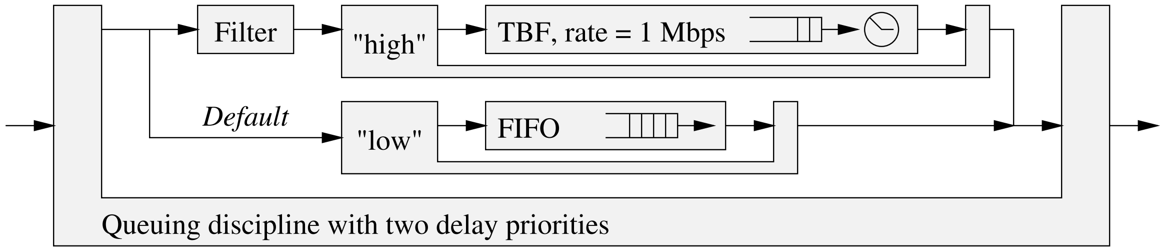

Figure 4: Combination of priority, TBF, and FIFO queuing disciplines.

wall is also capable of tagging microflows into classes. Behavior aggregate classification could also be done using `cls_u32` and `ipchains`, but since we usually already have `sch_dsmark` at the top level, we use the simpler `cls_tcindex` and retrieve the DSCP using `sch_dsmark`, which then puts it into `skb->tc_index`.

When using `sch_dsmark`, the class number returned by the classifier is stored in `skb->tc_index`. This way, the result can be re-used during later processing steps.

Nodes in multiple DS domains must also be able to distinguish packets by the inbound interface in order to translate the DSCP to the correct PHB. This can be done using the `route` classifier, in combination with the `ip rule` command interface subset.

Marking is done when a packet is dequeued from `sch_dsmark`. `sch_dsmark` uses `skb->tc_index` as an index to a table in which the outbound DSCP is stored and puts this value into the packet's DS field.

Figure 6 shows the use of `sch_dsmark` and `skb->tc_index` in a micro-flow classifier based on `cls_rsvp`. Figure 7 shows a behavior aggregate classifier using `cls_tcindex`.

3.3 Cascaded meters

Multiple meters are needed if traffic should be assigned to more than two classes, based on the bandwidth it uses. As an example, such classes could be for "low", "high", and "excess" traffic.

Our current implementation supports a limited form of cascading at the level of classifiers. We are testing a cleaner and more efficient solution at the time of writing.

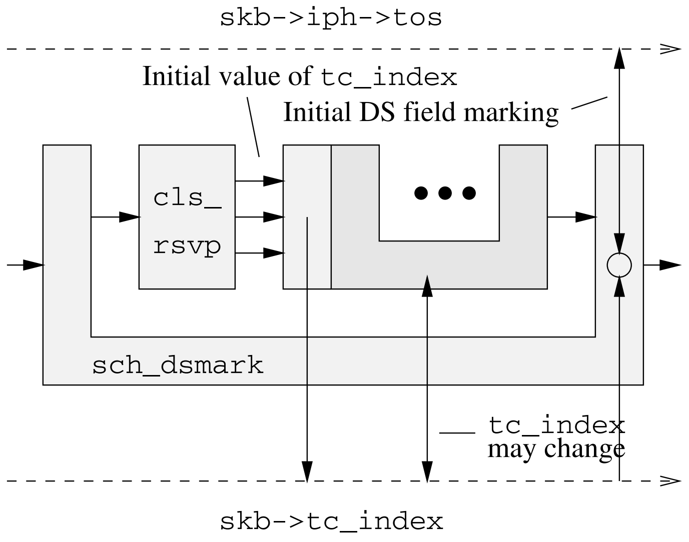

Figure 6: Micro-flow classifier.

3.4 Implementing PHBs

PHBs based only on delay priorities, e.g. Expedited Forwarding [4], can be built using CBQ [5] or the more simple `sch_prio`. (See section 5.)

Besides four delay priorities, which can again be implemented with already existing components, Assured Forwarding [6] also needs three drop priorities, which is more than the current implementation of RED supports. We therefore added a new queuing discipline which we call

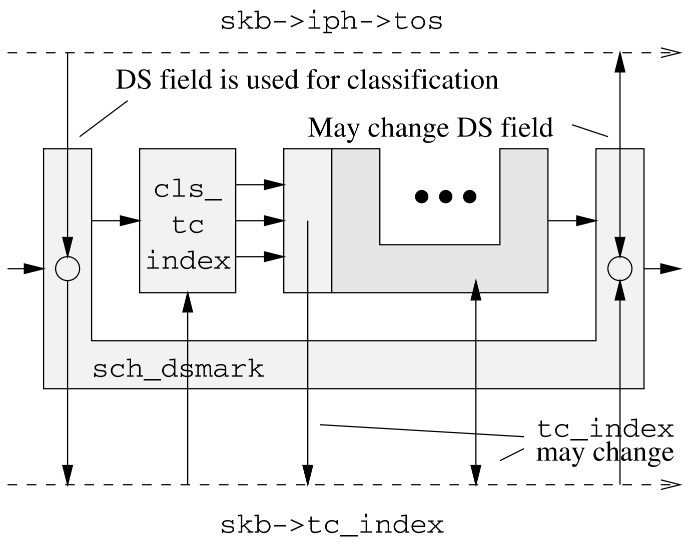

Figure 7: Behaviour aggregate classifier.

"generalized RED" (GRED). GRED uses the lower bits of `skb->tc_index` to select the drop class and hence the corresponding set of RED parameters.

3.5 Shaping

The so-called Token Bucket Filter (`sch_tbf`) can be used for shaping at edge nodes. Unfortunately, the highest rate at which `sch_tbf` can shape is limited by the system timer, which normally ticks at 100 Hz, but can be accelerated to 1 kHz or more if the processor is sufficiently powerful. Note that Linux traffic control supports more granular clocking for droppers (i.e. shapers without buffer).

CBQ can also be used to do shaping.

Higher rates can be shaped when using hardware-based solutions, such as ATM.

3.6 End systems

Diffserv-capable sources use the same functionality as edge routers, i.e. any classification and traffic conditioning can be administratively configured.

In addition to that, an application may also choose to mark packets when they are generated. For IPv4, this can be done using the `IP_TOS` socket option, which is commonly available on Unix, and of course also on Linux. Note that Linux follows the [7] convention of not allowing the lowest bit of the TOS byte to be different from zero. This restriction is compatible with use for Diffserv. Furthermore, the use of values corresponding to high precedences (i.e. DSCP 0x28 and above) is restricted. This can be avoided either by giving the application the appropriate capabilities (privileges), or by re-marking (see below).

Setting the DS field with IPv6 is currently very awkward. In the future, an improved interface is likely to be provided that unifies the IPv4 and IPv6 usage and that may contain additional improvements, e.g. selection of services instead of "raw" DS field values.

An application's choice of DS field values can always be refused or changed by traffic control (using re-marking) before a packet actually reaches the network.

4 New components

The prototype implementation of Diffserv support requires the addition of three new traffic control elements to the kernel: (1) the queuing discipline `sch_dsmark` to extract and to set the DSCP, (2) the classifier `cls_tcindex` which uses this information, and (3) the queuing discipline `sch_gred` which supports multiple drop priorities and buffer sharing.

Only the queueing discipline to extract and set the DSCP is truly specific to the differentiated services architecture. The other two elements can also be used in other contexts.

Figure 6 shows the use of `sch_dsmark` for the initial packet marking when entering a Diffserv domain. The classification and rate control metering is performed by a microflow classifier, e.g. `cls_rsvp`, in this case.

This classifier determines the initial TC index which is then stored in `skb->tc_index`. Afterwards, further processing is performed by an inner queuing discipline. Note that this queuing discipline may read and even change `skb->tc_index`.

When a packet leaves `sch_dsmark`, `skb->tc_index` is examined and the DS field of the packet is set accordingly.

Figure 7 shows the use of `sch_dsmark` and `cls_tcindex` in a node which works on a behavior aggregate, i.e. on packets with the DS field already set. The procedure is quite similar to the previous scenario, with the exception that `cls_tcindex` takes over the role of `cls_rsvp` and that the DS field of the incoming packet is copied to `tc_index` before invoking the classifier.

Note that the value of the outbound DS field can be affected at three locations: (1) in `sch_dsmark`, when classifying based on `skb->tc_index`, which contains the original value of the DS field; (2) by changing `skb->tc_index` in an inner queuing discipline; and (3) in `sch_dsmark`, when mapping the final value of `skb->tc_index` back to a new value of the DS field.

4.1 `sch_dsmark`

As illustrated in figure 8, the `sch_dsmark` queuing discipline performs three actions based on the scripting invocation:

- If `set_tc_index` is set, it retrieves the content of the DS field and stores it in `skb->tc_index`.
- It invokes a classifier and stores the class ID returned in `skb->tc_index`. If the classifier finds no match, the value of `default_index` is used instead. If `default_index` is not set, the value of `skb->tc_index`

is not changed. Note that this can yield undefined behaviour if neither `set_tc_index` nor `default_index` is set.

- After sending the packet through its inner queuing discipline, it uses the resulting value of `skb->tc_index` as an index into a table of (mask,value) pairs. The original value of the DS field is then replaced using the following formula:
  `ds_field = (ds_field & mask) | value`

Table 4.1 lists the parameters that can be configured in the `dsmark` queuing discipline. The upper part of the table shows parameters of the queuing discipline itself. The lower part shows parameters of each class.

`indices` is the size of the table of (mask,value) pairs.

4.2 `cls_tcindex`

As shown in figure 9, the `cls_tcindex` classifier uses `skb->tc_index` to select classes. It first calculates the lookup key using the algorithm
`key = (skb->tc_index >> shift) & mask`
Then it looks for an entry with this handle. If an entry is found, it may call a meter (if configured), and it will return the class IDs of the corresponding class.

If no entry is found, the result depends on whether `fall_through` is set. If set, a class ID is constructed from the lookup key. Otherwise, it returns a “not found” indication and the search continues with the next classifier. We call construction of the class ID an “algorithmic mapping”. This can be used to avoid setting up a large number of classifier elements if there is a sufficiently simple relation between values of `skb->tc_index` and class IDs. An example of this trick is used in the AF scripts on the web site.

The size of the lookup table can be set using the `hash` option. `cls_tcindex` automatically uses perfect hashing if the range of possible choices does not exceed the size of the lookup table. If the `hash` option is omitted, an implementation-dependent default value is chosen.

Table 4.2 shows the parameters that can be configured in the `tcindex` classifier. The upper part of the table shows parameters of the classifier itself. The lower part shows parameters of each element.

Note that the keyword used by `tc` (the command-line tool used to manually configure traffic control elements) does not always correspond to the variable internally used by `cls_tcindex`.

4.3 `sch_gred`

Figure 10 shows how `sch_gred` uses `skb->tc_index` for the selection of the right virtual queue (VQ) within a physical queue. What makes `sch_gred` different from other Multi-RED implementations is the fact that it is decoupled from any one specific block along the packet’s path such as a

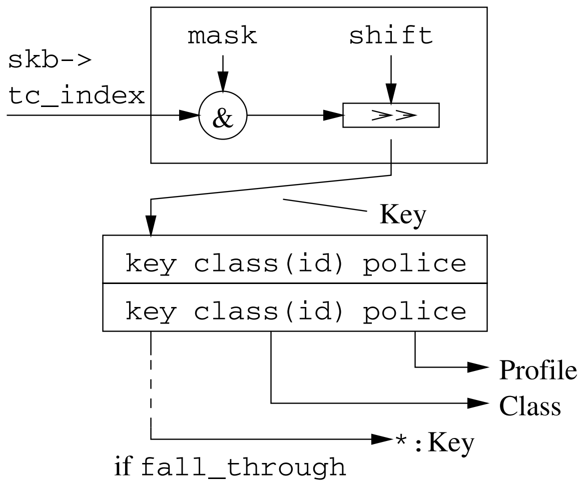

Figure 9: The `tcindex` classifier.

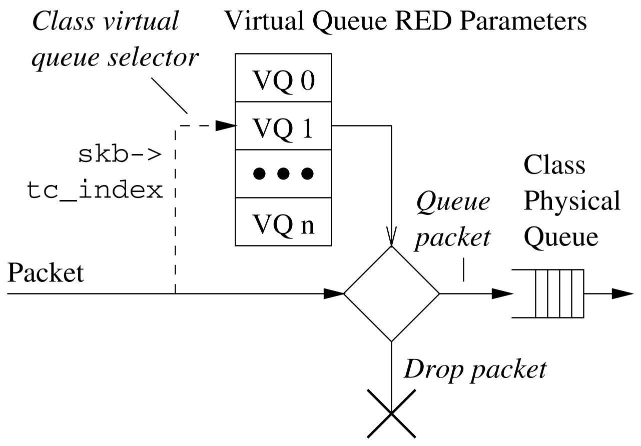

Figure 10: Generic RED and the use of `skb->tc_index`

header classifier or meter. For example, CISCO’s DWRED [8] is tied to mapping VQ selection based on the precedence bits classification. On the other hand, RIO [9] is tied to the IN/OUT metering levels for the selection of the VQ. In the case of GRED, any classifier, meter, etc. along the data path can affect the selection of the VQ by setting the appropriate value of `skb->tc_index`.

GRED also differs from the two mentioned multiple RED mechanisms in that it is not limited to a specific number of VQ. The number of VQs is configurable for each physical class queue. GRED does not assume certain drop precedences (or priorities). It depends on the configuration parameters passed on by the user. In essence, DWRED and RIO are special cases of GRED.

Currently, the number of virtual queues is limited to 16 (the least significant 4 bits of `skb->tc_index`). There is a one to one mapping between the values of `skb->tc_index` and the virtual queue number in a class. Buffer sharing is

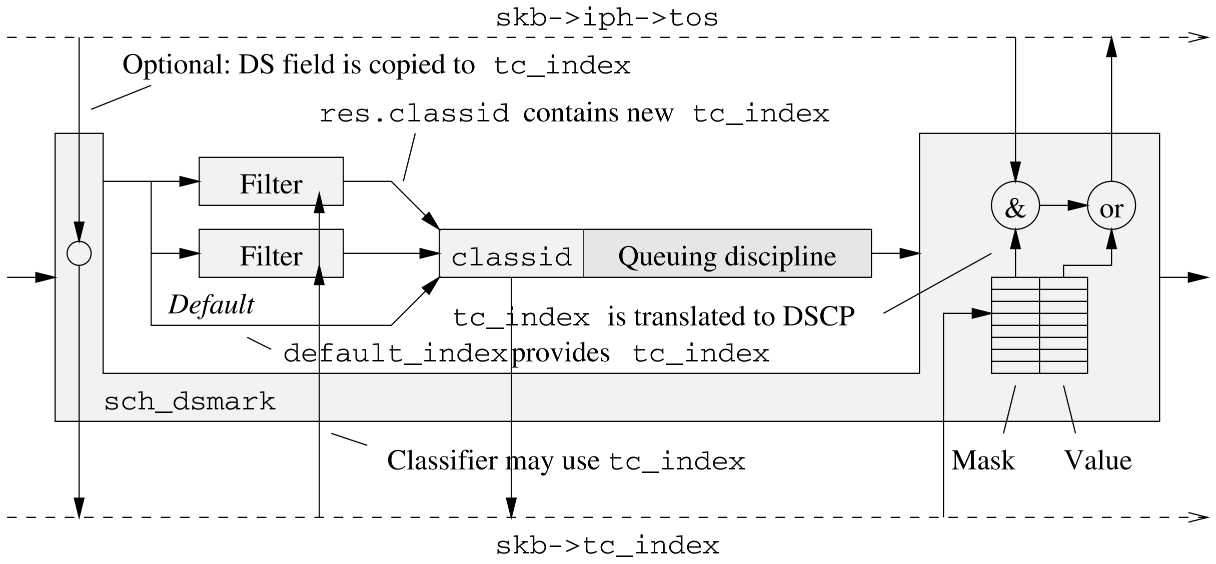
skb->iph->tos
Optional: DS field is copied to tc_index
res.classid contains new tc_index
Filter
Filter
Default
classid
Queuing discipline
tc_index is translated to DSCP
default_index provides tc_index
sch_dsmark
&
or
Classifier may use tc_index
Mask
Value
skb->tc_index

Figure 8: The `dsmark` queuing discipline.

Variable name / `tc` keyword	Value	Default
`indices`	2^n	none
`default_index`	0. . .`indices`-1	absent
`set_tc_index`	none (flag)	absent
`mask`	0. . .0xff	0xff
`value`	0. . .0xff	0

Table 1: Configuration parameters of `sch_dsmark`.

achieved using one of two ways (selectable via configuration):

- Simple setting of physical queue limits. It is up to the individual configuring the virtual queues parameters to decide which one gets preferential treatment. Sharing and preferential treatment amongst virtual queues is based on parameter settings such as the per-virtual queue physical limit, threshold values and drop probabilities. This is the default setting.
- A similar average queue trick as that is used in [9]. This is selected by the operator `grio` during the setup. Each VQ within a class is assigned a priority at configuration time. Priorities range from 1 to 16 at the moment, with 1 being the highest. The computation of the average queue value (for a VQ) involves first summing to the current stored average queue value all the the other average queue values of the VQs which are more important than it. This way a relatively higher priority (lower priority value) gets preferential treatment because its average queue is always the lowest; the relatively lower priority will still continue to send when the higher ones are not dominating the buffer space. A user can still configure the per-virtual-Queue physical queue limits, threshold values, and drop probabilities as in the (first) case when the `grio` option is not defined.

The second scheme is slightly slower than the first one (a few more per-packet computations).

GRED is configured in two steps. First (see also the upper part of table 4.3) the generic parameters are configured to select the number of virtual queues `DPs` and whether to turn on the RIO-like buffer sharing scheme (`grio`). Also at this point, a default virtual queue is selected so that packets with out of range values of `skb->tc_index` or misconfigured priorities in the case of `grio` buffer-sharing setup are directed to it. Normally, the default virtual queue is the one with the highest likelihood of having a packet discarded. The operator `setup` identifies that this is a generic setup for GRED.

The second step is to set parameters for individual virtual queues. (See also the lower part of table 4.3). These parameters are equivalent to the traditional RED parameters. In addition, each RED configuration identifies which virtual queue the parameters belong to as well as the priority if the `grio` technique is selected. The mandatory parameters are:

- `limit` defines the virtual queue “physical” limit in

Variable	`tc` keyword	Value	Default
`hash`	`hash`	1...0x10000	implementation-dependent
`mask`	`mask`	0...0xffff	0xffff
`shift`	`shift`	0...15	0
`fall_through`	`fall_through`/ `pass_on`	flag	`fall_through`
`res`	`classid`	*major*:*minor*	none
`police`	`police`	Profile	none

Table 2: Configuration parameters of `cls_tcindex`.

Variable	`tc` keyword	Value	Default
`DPs`	`DPs`	1...16	none
`def`	`default`	1...`DPs`	none
`grio`	`grio`	none (flag)	absent
`limit`	`limit`	bytes	none
`qth_min`	`min`	bytes	none
`qth_max`	`max`	bytes	none
n/a	`avpkt`	bytes	none
n/a	`bandwidth`	rate	10 Mbps
n/a	`burst`	packets	none
n/a	`probability`	[0 ... 1)	0.02
`DP`	`DP`	1...`DPs`	0
`prio`	`prio`	1...`DPs`	none

Table 3: Configuration parameters of `sch_gred`.

bytes.

- `min` defines the minimum threshold value in bytes.
- `max` defines the maximum threshold value in bytes.
- `avpkt` is the average packet size in bytes.
- `bandwidth` is the wire-speed of the interface.
- `burst` is the number of average-sized packets allowed to burst. The Linux RED implementation attempts to compute an optimal W value for the user based on the `avpkt`, minimum threshold and allowed burst size. This is based on the equation:

$$\text{burst} + 1 - \frac{\text{qmin}}{\text{avpkt}} < \frac{1-(1-\text{W})^{\text{burst}}}{\text{W}}$$

as described in [10].
- `probability` defines the drop probability in the range [0 ...).
- `DP` identifies the virtual queue assigned to these parameters.
- `prio` identifies the virtual queue priority if `grio` was set in the general parameters.

5 Building sample configurations

Communication and configuration of the kernel code or modules is achieved by a user level program tc written by Alexey. The interaction is shown in figure 5.

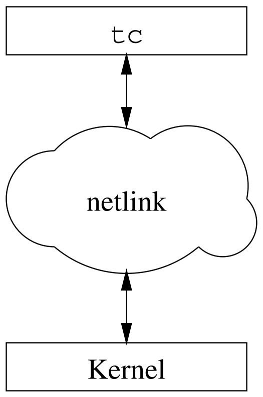

Figure 11: User space to kernel communication using `tc`

Given the flexibility of the code, there are many ways to reach the same end goal. Depending on the requirement, one could script the same PHB using a different combinations of qdiscs; e.g. one could build a core EF capable router using either CBQ to rate limit it and prioritise its traffic or instead use the PRIO qdisc with a Token Bucket attached to rate limit it. It is hoped that users of Linux Diffserv will be able to script their own flavored configurations. The examples below (as well as those on the Linux Diffserv web site) are simplistic, in the sense that they only assume one interface per node. One should easily be able to extend them for more than one interface

The normal recipe for creating a configuration script is:

- attach `sch_dsmark` to the output interface
- define the structure of the queuing discipline(s) inside `sch_dsmark`
- number the classes and decide on a numbering scheme to use for `skb->tc_index` (the latter may be trivial if `skb->tc_index` is only used within `sch_dsmark`.)
- identify which packets go to which classes and configure the classifier(s) of `sch_dsmark` accordingly

The script lines in the next subsections are numbered for clarity of the accompanying description below.

For clarity, we did not include handling of historical DS field values in our scripts.

5.1 Edge device: Packet re-marking

```
1. tc qdisc add dev eth0 handle 1:0 root dsmark indices 64
2. tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
3. tc class change dev eth0 classid 1:2 dsmark mask 0x3 value 0x68
4. tc class change dev eth0 classid 1:3 dsmark mask 0x3 value 0x48
5. tc filter add dev eth0 parent 1:0 protocol ip prio 4 handle 1: u32
       divisor 1
6. tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 2: u32
       divisor 1
7. tc filter add dev eth0 parent 1:0 prio 4 u32
       match ip dst 10.0.0.0/24
       police rate 1Mbit burst 2K continue
       flowid 1:1
8. tc filter add dev eth0 parent 1:0 prio 5 u32
       match ip dst 10.0.0.0/24
       flowid 1:2
9. tc filter add dev eth0 parent 1:0 prio 4 u32
       match ip dst 10.1.0.0/16
       match ip src 192.1.0.0/16
       match ip protocol 6 0xff
       match ip dport 0x17 0xffff
       flowid 1:3
```

The first line attaches a dsmarker to the interface eth0 on the root node. The second line instructs the dsmarker to remark the DSCP of classid 1:2 by first masking out bits 6 and 7 then ORing that with a value of 0xb8. Note that: This is equivalent to ignoring the ECN bits, and setting the code point value to 0x2e (which happens to be the DSCP for EF). In a similar manner, the third line instructs the dsmarker to remark the CP of classid 1:2 to 0x1a (DSCP for AF31). The fourth line adds a remarking the class 1:3 DSCPs to 0x12 (DSCP for AF21). These three lines in effect are also registering the classes 1:2, 1:3 and 1:4.

Line 5 adds a u32 classifier with priority of 4. Line 6 adds another classifier of a lower priority. Line 7 maps all packets with a source IP address of 10.0.0.0/24 to class 1:1. Line 7 and 8 show how one can attach a meter to a classifier and the reaction to an exceeding of the rate. Basically, the trick is to define two filters matching the same headers with a higher priority one attached with a meter and policing action. The operator `continue` is used to allow a lookup of the next lower priority matching filter. In this case, should the metering be exceeded in class 1:1, the flow is reclassified to class 1:2. Line 9 selects all TCP packets from source subnet 10.1.1.0/16 destined towards subnet 192.1.1.0/16 and sends them to the queue for class 1:3.

The overall effect is: all packets coming in from source subnet address 10.0.0.0/24 will get their packets marked with a DSCP of 0x2e (EF class/PHB) up to a point where they start exceeding their allocated rate (of 1Mbps and burst of 2K). In this case, the packets are demoted to class 1:2 where they will be remarked to DSCP 0x18 (AF21). Any TCP packets of origin subnet 10.1.1.0/16 destination subnet 192.1.1.0/16 will be remarked to 0x1A (AF22). It is easy to see that one can build a multi-color marking scheme of large depths using using such cascading filter/metering schemes.

5.2 Core device: EF using CBQ

The script below is the output of the EF Perl script on the Linux Diffserv Web site.

```
1. tc qdisc add dev eth0 handle 1:0 root dsmark indices 64
        set_tc_index
2. tc filter add dev eth0 parent 1:0 protocol ip prio 1
        tcindex mask 0xfc shift 2
3. tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq
        bandwidth 10Mbit allot 1514 cell 8 avpkt 1000 mpu 64
4. tc class add dev eth0 parent 2:0 classid 2:1 cbq
        bandwidth 10Mbit
        rate 1500Kbit avpkt 1000 prio 1 bounded isolated
        allot 1514 weight 1 maxburst 10 defmap 1
5. tc qdisc add dev eth0 parent 2:1 pfifo limit 5
6. tc filter add dev eth0 parent 2:0 protocol ip prio 1
        handle 0x2e tcindex classid 2:1 pass_on
7. tc class add dev eth0 parent 2:0 classid 2:2 cbq
        bandwidth 10Mbit rate 5Mbit avpkt 1000 prio 7
        allot 1514 weight 1 maxburst 21 borrow
8. tc qdisc add dev eth0 parent 2:2 red limit 60KB min 15KB
        max 45KB burst 20 avpkt 1000 bandwidth 10Mbit
        probability 0.4
9. tc filter add dev eth0 parent 2:0 protocol ip prio 2
        handle 0 tcindex mask 0 classid 2:2 pass_on
```

Line 1 attaches to the root node on interface eth0 a dsmarker which copies the TOS byte into `skb->tc_index`. Line 2 adds a filter to the root node which exists merely to mask out the ECN bits and extract the DSCP field by shifting to the right by two bits. A classful qdisc using CBQ is attached to node 2:0 (2:0 is the child of the root node 1:0) – this is in line 3. Two child classes are defined out of the 2:0 node. 2:1 is of type CBQ which is bound to a rate of 1.5 Mbps (line 4). A packet counting FIFO qdisc (`pfifo`) with a maximum queue size of 5 packets is attached to the CBQ class as the buffer management scheme (line 5). Line 6 adds a `tcindex` classifier which will redirect all packets with a `skb->tc_index` 0x2e (the DSCP for EF) to classid 2:1 – non 0x2e are allowed to fall through so they can be matched by another filter. Line 7 defines another CBQ class, 2:1, emanating out of node 2:0 – this is intended to be the Best Effort class. The rate is limited to 5 Mbps; however, the class is allowed to borrow extra bandwidth if it is not being used (via the operator `borrow`). Since the EF class does not lend its bandwidth (operator `isolated` line

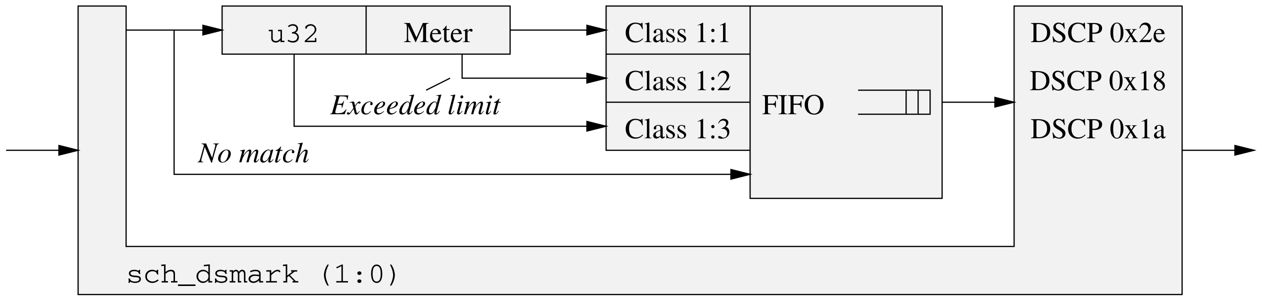

Figure 12: Packet re-marking at the edge.

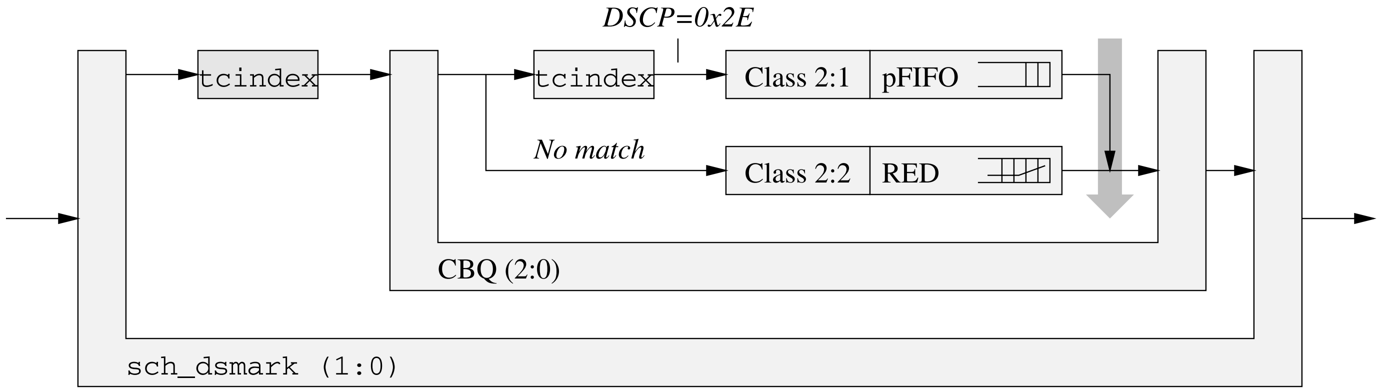

Figure 13: Configuring EF using CBQ.

4), the BE can only borrow up to a maximum of an extra 3.5Mbps. Note that in scenarios where there is no congestion on the wire, this might not be a very smart provisioning scheme since the BE traffic will probably get equivalent traffic performance as EF. The major differentiator in that case will be the priorities. The EF class' traffic will always be served first as long as there is something on the queue (prio 1 is higher than prio 8 in comparing line 4 and 7). Line 8 attaches RED as the buffer management scheme to be used by the BE class. Line 9 then maps the rest of the packets (without DSCP of 0x2e) to the classid 2:2. The description of the RED and CBQ parameters are beyond the scope of this document.

6 Conclusion

We have given a brief introduction to the elements of Linux traffic control in general, and we have explained how the existing infrastructure can be extended in order to support Diffserv. We have then shown how we implemented support for the Diffserv architecture in Linux, using the traffic control framework of recent kernels. We have also described how nodes can be configured using our work.

Our implementation provides a very flexible platform for experiments with PHBs already under standardization as well as experiments with new PHBs. It can also serve as a platform for work in other areas of Diffserv, such as edge configuration management and policy management.

Future work will focus on the elimination of a few restrictions that still exist in our architecture, and also in the simplification of the configuration procedures.

References

[1] RFC2475; Blake, Steven; Black, David; Carlson, Mark; Davies, Elwyn; Wang, Zheng; Weiss, Walter. *An Architecture for Differentiated Services*, IETF, December 1998.

[2] RFC2474; Nichols, Kathleen; Blake, Steven; Baker, Fred; Black, David. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, IETF, December 1998.

[3] Almesberger, Werner. *Linux Traffic Control — Implementation Overview*, `ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz`, Technical Report SSC/1998/037, EPFL, November 1998.

[4] RFC2598; Jacobson, Van; Nichols, Kathleen; Poduri, Kedarnath. *An Expedited Forwarding PHB*, IETF, June 1999.

[5] Floyd, Sally; Jacobson, Van. *Link-sharing and Resource Management Models for Packet Networks*, IEEE/ACM

Transactions on Networking, Vol. 3 No. 4, pp. 365-386, August 1995.

[6] RFC2597; Heinanen, Juha; Baker, Fred; Weiss, Walter; Wroclawski, John. *Assured Forwarding PHB Group*, IETF, June 1999.

[7] RFC1349; Almquist, Philip. *Type of Service in the Internet Protocol Suite*, IETF, July 1992.

[8] CISCO DWRED. *Distributed Weighted Random Early Detection*, `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.htm`

[9] Clark, David; Wroclawski, John. *An Approach to Service Allocation in the Internet*, Internet Draft `draft-clark-diff-svc-alloc-00.txt`, July 1997.

[10] Floyd, Sally; Jacobson, Van. *Random Early Detection Gateways for Congestion Avoidance*, IEEE/ACM Transactions on Networking, August 1993.

7 Author's address

Werner Almesberger
Institute for computer Communications and Applications
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne
Switzerland
email: Werner.Almesberger@epfl.ch

Jamal Hadi Salim
Computing Technology Labs
Nortel Networks
P.O.BOX 3511, Station C
Ottawa, Ontario
Canada K1Y 4H7
email: hadi@nortelnetworks.com

Alexey Kuznetsov
Institute for Nuclear Research (INR)
60th October Anniversary pr. 7a
Moscow 117312
Russia
email: kuznet@ms2.inr.ac.ru