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

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; [1]) provides an infrastructure for applications, users, or providers to select the network service that best suits their needs. Services may differ in many ways, such as delay or loss goals.

Diffserv defines local node services in terms of the forwarding behavior of individual routers (the so-called Per-Hop-Behavior; PHB). Diffserv defines only PHBs which can be used to define end-to-end services, however the actual use of these building blocks to define end-to-end services is beyond the current scope of the IETF Diffserv Working Group [2].

When forwarding a packet, a node selects the PHB to apply based on the content of the Diffserv field (short "DS field") in the IP header [3]. This value is called the Diffserv Code Point (DSCP). Note that each network may decide on its own mapping between DSCP values and PHBs. Nevertheless, each PHB definition also proposes a default DSCP value.

The Diffserv design allows PHBs to be defined, implemented, and deployed in a largely independent way. It is therefore important to preserve this flexibility in any implementation.

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 introduces the concepts of the Diffserv architecture. Section 3 gives a brief overview of traffic control functions in recent Linux kernels. Section 4 discusses where the existing model needed to be extended. Section 5 describes the new components in more detail.

2 Differentiated Services

Figure 1 shows the general structure of the forwarding path in a Diffserv node.

Figure 1: *General Diffserv forwarding path.*

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

2.1 Classification and metering

Diffserv distinguishes two types of classification: a "behavior aggregate classifier" distinguishes packets based only on their DS fields. A "micro-flow classifier" may take into account the whole packet, e.g. the source and destination IP addresses, port numbers, etc.

Classification based on packet contents may also be supplemented by metering of traffic flows, e.g. in order to accept only limited traffic for a given PHB.

2.2 Marking

The process of setting or modifying the DS field is called marking. Marking is necessary in several cases, for example:

- Whenever a packet from a non-Diffserv network reaches the edge of a Diffserv network, its DS field has to be initialized to the appropriate DSCP.
- Diffserv-capable hosts need to be able to set the DS field of packets they originate.
- Since different parts of a network may use different DSCP to PHB mappings, edge routers may have to change the DS field in packets crossing such a boundary.
- A PHB group may use multiple PHBs and hence multiple DSCPs to convey additional information (e.g. some form of congestion indication). In this case, the DS field may change at any Diffserv-capable node along the path.

2.3 PHBs

Three groups of PHBs are currently being defined in the Diffserv WG:

- PHBs for compatibility with historical use of the IPv4 TOS byte (defined in [3])
- Expedited forwarding, a simple high-priority PHB [4]
- Assured Forwarding, a group of PHBs with different delay and drop priorities [5]

3 Linux Traffic Control

Figure 2 shows roughly how the kernel processes data received from the network, and how it generates new data to be sent on the network.

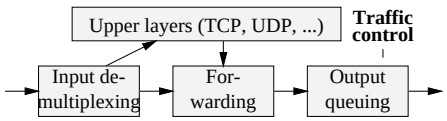

Figure 2: *Processing of network data.*

"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.

3.1 Components

The traffic control code in the Linux kernel consists of the following major conceptual components: (1) queuing disciplines; (2) classes (within a queuing discipline); (3) filters; and (4) 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.

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.

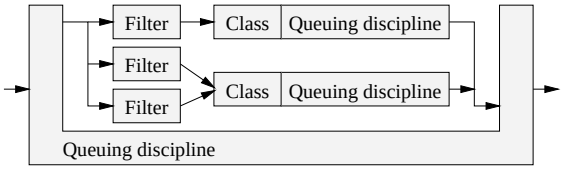

Figure 3: *A simple queuing discipline with multiple classes.*

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.

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.

4 Diffserv extensions to Linux traffic control

The traffic control framework available in recent Linux kernels [6] 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.

4.1 Overview

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`.

4.2 Classification and marking

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

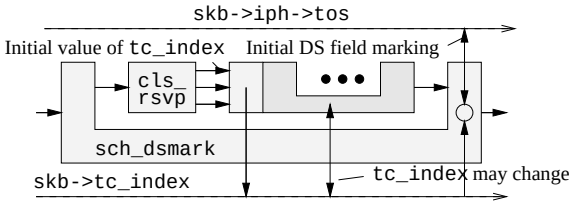

Figure 4: *Micro-flow classifier.*

Figure 4 shows the use of `sch_dsmark` and `skb->tc_index` in a micro-flow classifier based on `cls_rsvp`. Figure 5 shows a behavior aggregate classifier using `cls_tcindex`.

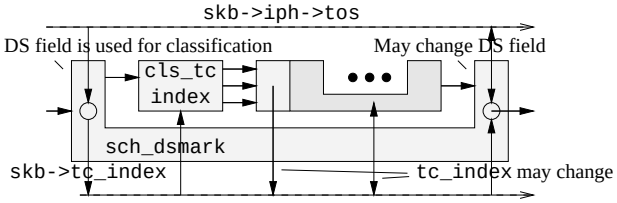

Figure 5: *Behaviour aggregate classifier.*

4.3 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.

4.4 Implementing PHBs

PHBs based only on delay priorities, e.g. Expedited Forwarding [4], can be built using CBQ [7] or the more simple `sch_prio`.

Besides four delay priorities, which can again be implemented with already existing components, Assured Forwarding [5] 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 "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.

4.5 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.

4.6 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 [8] 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.

5 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 4 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 micro-flow 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 5 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.

5.1 `sch_dsmark`

As illustrated in figure 6, 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`

5.2 cls_tcindex

As shown in figure 7, 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.

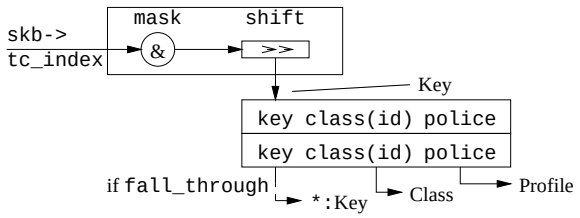

Figure 7: *The* `tcindex` *classifier.*

5.3 sch_gred

Figure 8 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 header classifier or meter. For example, CISCO's DWRED [9] is tied to mapping VQ selection based on the precedence bits classification.

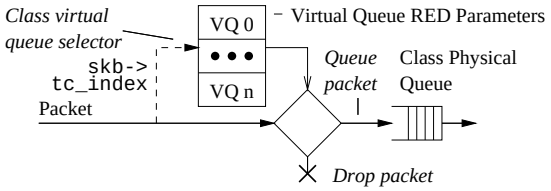

Figure 8: *Generic RED and the use of* `skb->tc_index`

On the other hand, RIO [10] 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 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 [10]. 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).

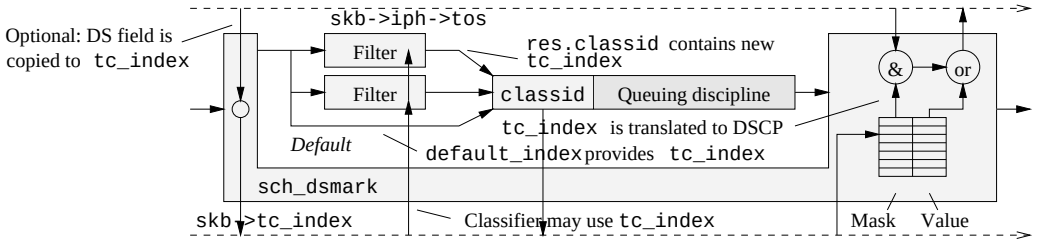

Figure 6: *The* `dsmark` *queuing discipline.*

GRED is configured in two steps. First the generic parameters are configured to select the number of virtual queues `DPs` and whether to turn on the RIO-like buffer sharing scheme (`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. 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 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: $\text{burst} + 1 - \frac{\text{qmin}}{\text{avpkt}} < (1-(1-\text{W})^{\text{burst}})/\text{W}$ as described in [11].
- `probability` defines the drop probability in the range $[0 \ldots)$.
- `DP` identifies the virtual queue assigned to these parameters.
- `prio` identifies the virtual queue priority if `grio` was set in the general parameters.

6 Conclusion

We have given a brief introduction to the Diffserv architecture and 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.

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] IETF, Differentiated Services (diffserv) working group. `http://www.ietf.org/html.charters/diffserv-charter.html`

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

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

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