Linux Network Traffic Control — Implementation Overview

Werner Almesberger, EPFL ICA
`Werner.Almesberger@epfl.ch`

April 23, 1999

Abstract

Linux offers a rich set of traffic control functions. This document gives an overview of the design of the respective kernel code, describes its structure, and illustrates the addition of new elements by describing a new queuing discipline.

1 Introduction

Recent Linux kernels offer a wide variety of traffic control functions. The kernel parts for traffic control, and several user-space programs to control them have been implemented by Alexey Kuznetsov `<kuznet@ms2.inr.ac.ru>`. That work was inspired by the concepts described in [1], but it also covers the mechanisms required for supporting the architecture developed in the IETF "intserv" group [2], and will serve as the basis for supporting the more recent work of "diffserv" [3]. See also [4] for further details on how intserv and diffserv are related. This document illustrates the underlying architecture and describes how new traffic control functions can be added to the Linux kernel. The kernel version we used is 2.2.6.

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: incoming packets are examined and then either directly forwarded to the network (e.g. on a different interface, if the machine is acting as a router or a bridge), or they are passed up to higher layers in the protocol stack (e.g. to a transport protocol like UDP or TCP) for further processing. Those higher layers may also generate data on their own and hand it to the lower layers for tasks like encapsulation, routing, and eventually transmission.

"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.

Sections 2 to 4 give an overview and explain some terminology. Sections 5 to 8 describe the elements of traffic control in the Linux kernel in more detail. Section 9 describes a queuing discipline that has been implemented by the author.

2 Overview

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

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

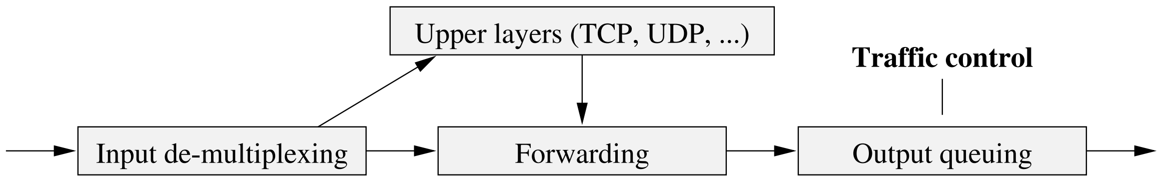


Figure 1: Processing of network data.

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.

Queuing discipline

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 at all. But flexibility doesn't end yet – classes normally don't 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.

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 better ways to accomplish what we've done here, e.g. by using *class-based queuing* (CBQ) [5].

Packets are enqueued as follows: when the `enqueue` function of a queuing discipline is called, it runs one filter after the other until one of them indicates a match. 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) may 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.

We will not try to introduce new terminology to distinguish among algorithms, their implementations, and instances of such elements, but rather use the terms queuing discipline, class, and filter throughout most of this document, to refer to all three levels of abstraction at the same time.

3 Resources

Linux traffic control is spread over a comparably large number of files. Note that all path names are relative to the base directory of the respective component,

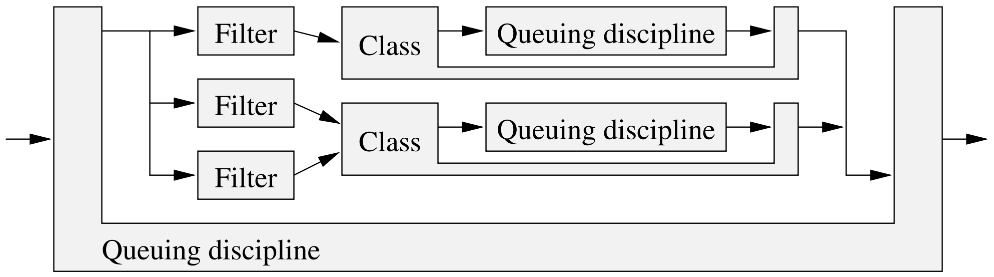


Figure 3: A simple queuing discipline with multiple classes.

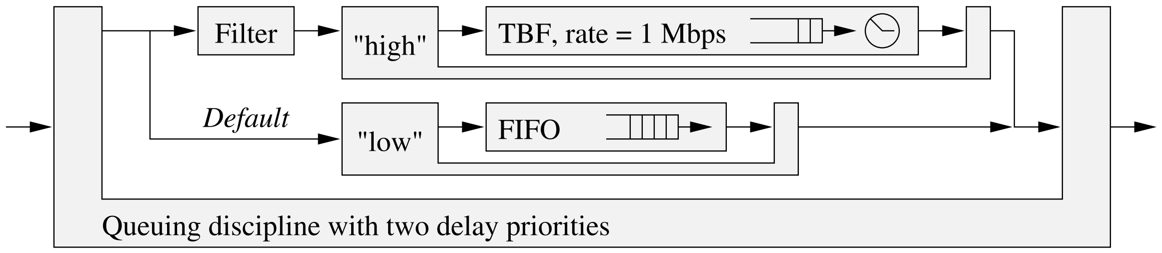


Figure 4: Combination of priority, TBF, and FIFO queuing disciplines.

e.g. for the Linux kernel this is `/usr/src/linux/`, for the `tc` program `iproute2/tc/`.

`tc` is a user-space program used to manipulate individual traffic control elements. Its source is in the file `iproute2-`*version*`.tar.gz`, which can be obtained from `ftp://linux.wauug.org/pub/net/ip-routing/`.

The kernel code resides mainly in the directory `net/sched/`. The interfaces between kernel traffic control elements and user space programs using them are declared in `include/linux/pkt_cls.h` and `include/linux/pkt_sched.h`. Declarations used only inside the kernel and the definitions of some inline functions can be found in `include/net/pkt_cls.h` and `include/net/pkt_sched.h`.

The *rtnetlink* mechanism used for communication between traffic control elements in user-space and in the kernel is implemented in `net/core/rtnetlink.c` and `include/linux/rtnetlink.h`. rtnetlink is based on *netlink*, which can be found in `net/netlink/` and `include/linux/netlink.h`.

The kernel source can be obtained from the usual well-known places, e.g. from `ftp://ftp.kernel.org/pub/linux/kernel/v2.2/`.

The example in section 9 is included in the ATM on Linux distribution, which can be downloaded from `http://icawww1.epfl.ch/linux-atm/dist.html`.

The Differentiated Services on Linux project (`http://icawww1.epfl.ch/linux-diffserv/`) has produced further examples for extensions of Linux traffic control and their use.

4 Terminology

Unfortunately, the terminology used to describe traffic control elements is far from consistent in literature, and there are some variations even within Linux traffic control. The purpose of this section is to help to put things into context.

Figure 5 shows the architectural models and the terminology used in the IETF groups “intserv” [6] and “diffserv” [7, 8], and how elements of Linux traffic control are related to them. Note that classes play an ambivalent role, because they determine the final outcome of a classification and they can also be part of the mechanism that implements a certain queuing or scheduling behaviour.

Table 1 summarizes the keywords used at the `tc` command line, the file names used in the kernel (in `net/sched/`), and the file names used in the source of `tc`.

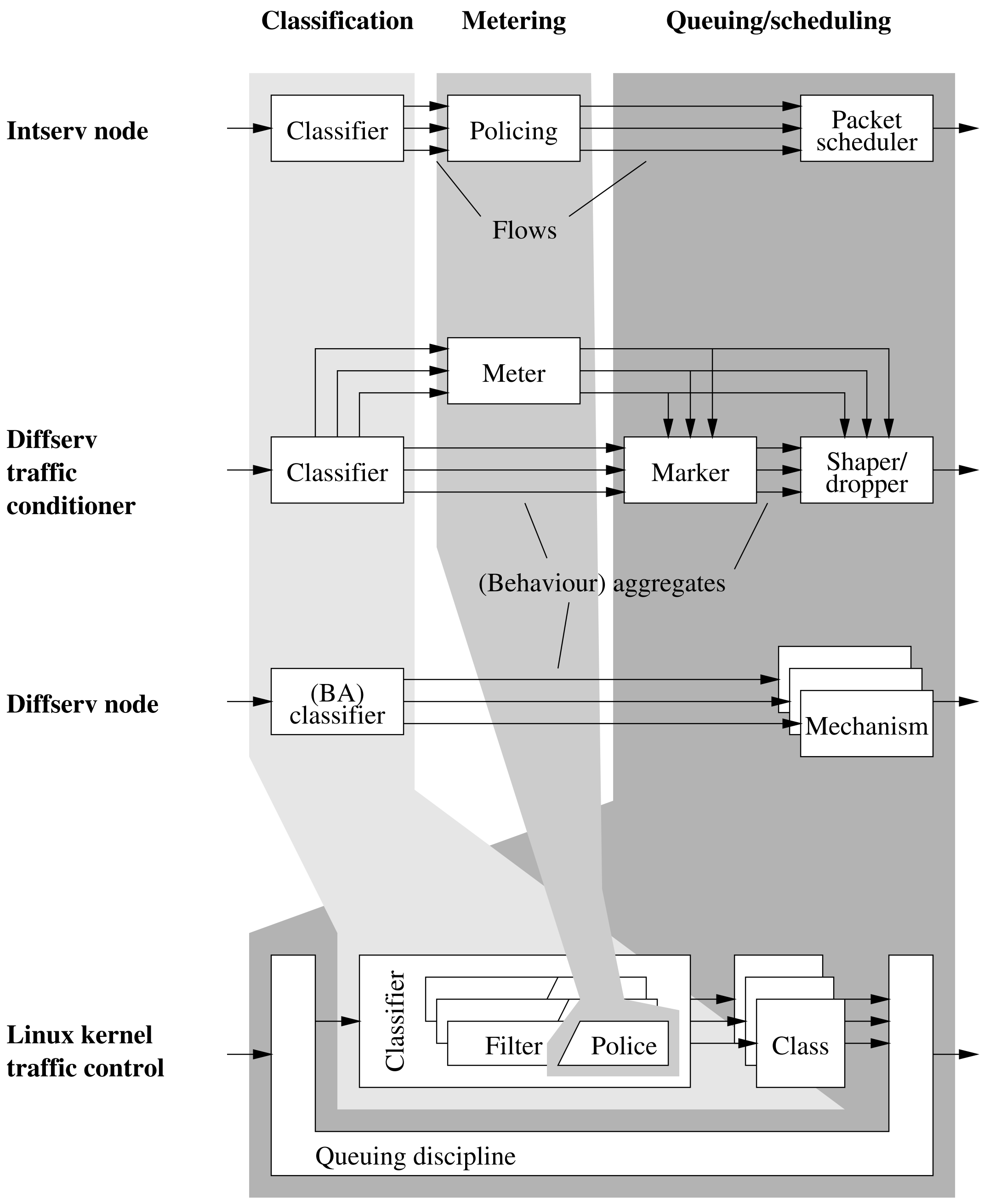


Figure 5: Relation of elements of the intserv and diffserv architecture to traffic control in the Linux kernel.

Element	tc keyword	File name prefix	
		Kernel	tc
Queuing discipline	qdisc	sch_	q_
Class	class	(sch_)	(q_)
Filter	filter	cls_	f_

Table 1: Keywords and file names used for traffic control elements.

5 Queuing disciplines

Each queuing discipline provides the following set of functions to control its operation (see `struct Qdisc_ops` in `include/net/pkt_sched.h`):

`enqueue` enqueues a packet with the queuing discipline. If the queuing discipline has classes, the `enqueue` function first selects a class and then invokes the `enqueue` function of the corresponding queuing discipline for further enqueuing.

`dequeue` returns the next packet eligible for sending. If the queuing discipline has no packets to send (e.g. because the queue is empty or because they're not scheduled to be sent yet), `dequeue` returns NULL.

`requeue` puts a packet back into the queue after dequeuing it with `dequeue`. This differs from `enqueue` in that the packet should be queued at exactly the place from which it was removed by `dequeue`, and that it should not be included in the statistics of cumulative traffic that has passed the queue, because that was already done in the `enqueue` function.

`drop` drops one packet from the queue.

`init` initializes and configures the queuing discipline.

`change` changes the configuration of a queuing discipline.

`reset` returns the queuing discipline to its initial state. All queues are cleared, timers are stopped, etc. Also, the `reset` functions of all queuing disciplines associated with classes of this queuing discipline are invoked.

`destroy` removes a queuing discipline. It removes all classes and possibly also all filters, cancels all pending events and returns all resources held by the queuing discipline (except for the data structure describing the queuing discipline itself).

`dump` returns diagnostic data used for maintenance. Typically, the `dump` functions returns all sufficiently important configuration and state variables.

For all these functions, queuing disciplines are usually referenced by a pointer to the corresponding `struct Qdisc`.

When a packet is enqueued on an interface (`dev_queue_xmit` in `net/core/dev.c`), the `enqueue` function of the device's queuing discipline (field `qdisc` of `struct device` in `include/linux/netdevice.c`) is invoked. Afterwards, `dev_queue_xmit` calls `qdisc_wakeup` in `include/net/pkt_sched.h` on that device to try sending the packet that was just enqueued.

`qdisc_wakeup` immediately calls `qdisc_restart` in `net/sched/sch_generic.c`, which is the main function to poll queuing disciplines and to send packets. `qdisc_restart` first tries to obtain a packet from the queuing discipline of the device, and if it succeeds, it invokes the device's `hard_start_xmit` function to actually send the packet. If sending fails for some reason, the packet is returned to the queuing discipline via its `requeue` function.

`qdisc_wakeup` can also be invoked by a queuing discipline when that queuing discipline notices that a packet may be due for sending, e.g. on expiration of a timer. TBF is an example of such a queuing discipline. `qdisc_restart` is also called via `qdisc_run_queues` from `net_bh` in `net/core/dev.c`. `net_bh` is the "bottom-half" handler of the networking stack and is executed whenever packets have been queued up for further processing.

Figure 6 illustrates the procedure. For simplicity, calls made by the queuing discipline (e.g. for classification) are not shown.

Note that queuing disciplines never make direct calls to delivery functions. Instead, they have to wait until they are polled.

If a queuing discipline is compiled into the the kernel, it should be registered by `pktsched_init` in `net/sched/sch_api.c`. Alternatively, is can also be registered from some other place using `register_qdisc`, e.g. from the `init_module` function if the queuing

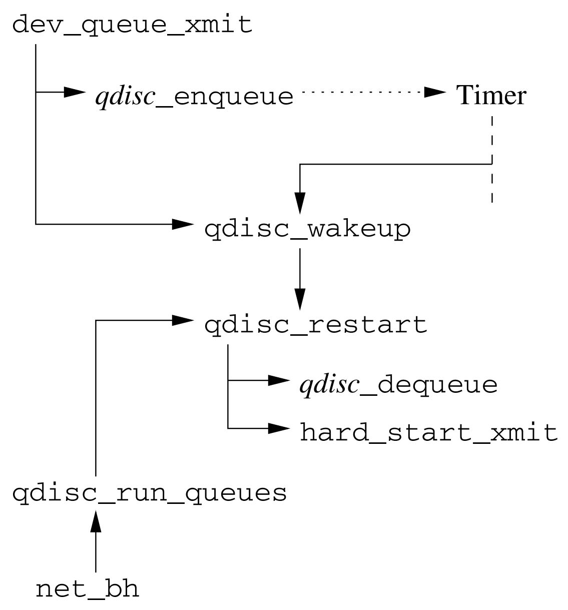


Figure 6: Functions called when enqueuing and sending packets.

discipline is compiled as a module.

When creating or changing an instance of a queuing discipline, a vector of options (type `struct rtattr *`, declared in `include/linux/rtnetlink.h`) is passed to the `init` function. Each option is encoded with its type, the length of the value, and the value (i.e. zero or more data bytes). Option types and the data structures used for values are declared in `include/linux/pkt_sched.h`. The option vector is parsed by calling `rtattr_parse`, which returns an array of pointers to the individual elements, indexed by the option type. The length and content of an option can be accessed via the macros `RTA_PAYLOAD` and `RTA_DATA`, respectively.

Option vectors are passed between user-space programs and the kernel using the rtnetlink mechanism. Explaining rtnetlink and the underlying netlink is beyond the scope of this paper. The location of the respective source files is described in section 3.

Instances of queuing disciplines are identified by 32-bit numbers, which are split into a major and a minor number. The usual notation is *major*:*minor*. For queuing disciplines, the minor number is always zero. Note that these major and minor numbers are not related to the numbers used for device special files.

6 Classes

Classes can be identified in two ways: (1) by the *class ID*, which is assigned by the user, and (2) by the *internal ID*, which is assigned by the queuing discipline. The latter has to be unique within a given queuing discipline and may be an index, a pointer, etc. Note that the value 0 is special and means "not found" when returned by `get`. The class ID is of type `u32`, while the internal ID is of type `unsigned long`. Inside the kernel, the usual way to refer to a class is by its internal ID. Only `get` and `change` use the class ID instead.

Note that multiple class IDs may map to the same internal class ID. In this case, the class ID conveys additional information from the classifier to the queuing discipline or class.

Class IDs are structured like queuing discipline IDs, with the major number corresponding to their instance of the queuing discipline, and the minor number identifying the class within that instance.

Queuing disciplines with classes provide the following set of functions to manipulate classes (see `struct Qdisc_class_ops` in `include/net/pkt_sched.h`):

`graft` attaches a new queuing discipline to a class and returns the previously used queuing discipline.

`leaf` returns the queuing discipline of a class.

`get` looks up a class by its class ID and returns the internal ID. If the class maintains a usage count, `get` should increment it.

`put` is invoked whenever a class that was previously referenced with `get` is dereferenced. If the class maintains a usage count, `put` should decrement it. If the usage count reaches zero, `put` may remove the class.

`change` changes the properties of a class. `change` is also used to create new classes, where applicable – some queuing disciplines have a constant number of classes which are created when the queuing discipline is initialized.

`delete` handles requests to delete a class. It checks if the class is not in use, and de-activates and removes it in this case.

`walk` iterates over all classes of a queuing discipline and invokes a callback function for each of them. This is used to obtain diagnostic data for all classes of a queuing discipline.

`tcf_chain` returns a pointer to the anchor of the list of filters associated with a class. This is used to manipulate the filter list.

`bind_tcf` binds an instance of a filter to the class. `bind_tcf` is usually identical to `get`, except when the queuing discipline needs to be able to explicitly refuse class deletion. (E.g. `sch_cbq` refuses to delete classes while they are referenced by filters.)

`unbind_tcf` removes an instance of a filter from the class. `unbind_tcf` is usually identical to `put`.

`dump_class` returns diagnostic data, like `dump` does for queuing disciplines.

Classes are selected in the `enqueue` function of the queuing discipline usually by invoking `tc_classify` in `include/net/pkt_cls.h`, which returns a `struct tcf_result` (in `include/net/pkt_cls.h`) containing the class ID (`classid`) and possibly also the internal ID (`class`), see section 7. The return value of `tc_classify` is either -1 (`TC_POLICE_UNSPEC`) or the policing decision returned by the filter (see section 8). The return values of `tc_classify` are declared in `include/linux/pkt_cls.h`.

There is also a shortcut for classification of locally generated traffic: if `skb->priority` contains the ID of a class of the current queuing discipline, that class is used and no further classification is attempted. `skb->priority` (`struct sk_buff` in `include/linux/skbuff.h`) is set to `sk->priority` (`struct sock` in `include/net/sock.h`) when locally generating a packet. `sk->priority` can be set with the `SO_PRIORITY` socket option (`sock_setsockopt` in `net/core/sock.c`). This type of classification can be useful for implementing functionality like the one provided by Arequipa [9].

Note that kernels up to at least 2.2.3 limit the value that can be set with `SO_PRIORITY` to the range $0\ldots7$, so that this shortcut classification does not work. However, all queuing disciplines support it. Also note that `skb->priority` can contain other priority values, e.g. the priority obtained from the TOS byte of the IPv4 header. All such values are below the smallest valid class number, 65536.

After selecting the class, the `enqueue` function of the respective inner queuing discipline is invoked. The way how this queuing discipline is stored in the data structure(s) associated with the class can vary among queuing discipline implementations.

The option vector passed to the `change` function is of the same structure as the vectors passed to the `init` functions of queuing disciplines. The corresponding declarations are also in `include/linux/pkt_sched.h`.

7 Filters

Filters are used by a queuing discipline to assign incoming packets to one of its classes. This happens during the enqueue operation of the queuing discipline.

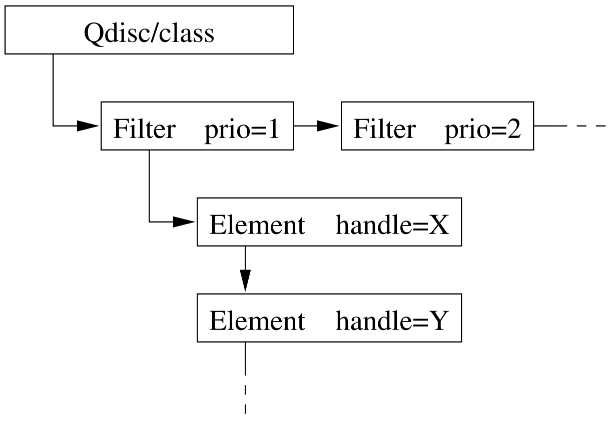


Figure 7: Structure of filters, with a list of elements belonging to the first filter, and no internal structure for the second filter.

Filters are kept in filter lists which can be maintained per queuing discipline or per class, depending on the design of the queuing discipline. Filter lists are ordered by priority, in ascending order. Furthermore, the entries are keyed by the protocol for which they apply. Those protocol numbers are also used in `skb->protocol` and they are defined in `include/linux/if_ether.h`. Filters for the same protocol on the same filter list must have different priorities.

A filter may also have an internal structure: it may control internal elements, which are then referenced by 32-bit handles. These handles are similar to class IDs, but they are not split into major and minor numbers. Handle 0 always refers to the filter itself. Like classes, also filters have internal IDs, which are obtained with the `get` function. The internal organization of a filter can be arbitrary. Figure 7 shows a filter with a list of internal elements.

Figure 8 shows the order in which filters and their elements can be examined. A linked list that is processed sequentially is of course only one of many possible internal structures of a filter.

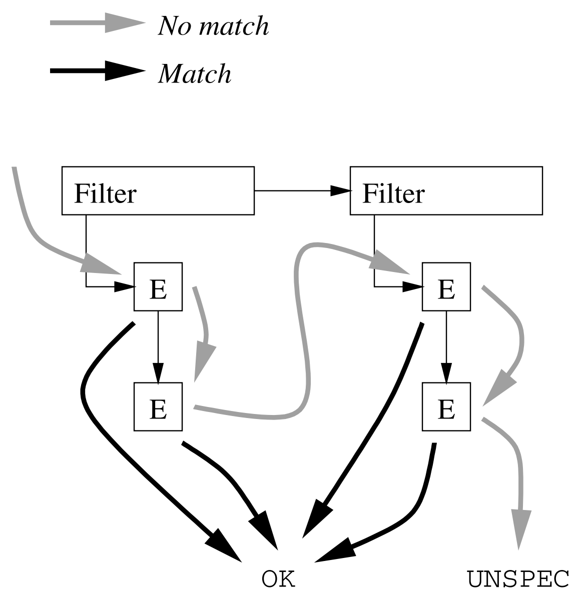


Figure 8: Looking for a match.

Filters are controlled via the following functions (see `struct tcf_proto_ops` in `include/net/pkt_cls.h`):

`classify` performs the classification and returns one of the `TC_POLICE_`... values described in section 8. If the result is not `TC_POLICE_UNSPEC`, it also returns the selected class ID and optionally also the internal class ID in the `struct tcf_result` pointed to by `res`. If the internal class ID is omitted, the value zero must be stored in `res->class`.

`init` initializes the filter.

`destroy` is invoked to remove a filter. Also the queuing disciplines `sch_cbq` and `sch_atm` use `destroy` to remove stale filters when deleting classes. If the filter or any of its elements were registered with classes, these registrations are canceled by calling `unbind_tcf`.

`get` looks up a filter element by its handle and returns the internal filter ID.

`put` is invoked when a filter element previously referenced with `get` is no longer used.

`change` configures a new filter or changes the properties of an existing filter. Configuration parameters are passed with the same mechanism as used for queuing disciplines and classes. `change` registers the addition of a new filter or filter element to a class by calling `bind_tcf`.

`delete` deletes an element of a filter. To delete the entire filter, `destroy` has to be used. This distinction is transparent to the user and is made in `net/sched/cls_api:tc_ctl_tfilter`. If the filter element was registered with a class, that registration is canceled by calling `unbind_tcf`.

`walk` iterates over all elements of a filter and invokes a callback function for each of them. This is used to obtain diagnostic data.

`dump` returns diagnostic data for a filter or one of its elements.

Note that the code for the RSVP filters is in `cls_rsvp.h`. `cls_rsvp.c` and `cls_rsvp6.c` only contain the right set of includes and set some parameters (mainly `RSVP_DST_LEN`), which control the type of filter generated from `cls_rsvp.h`.

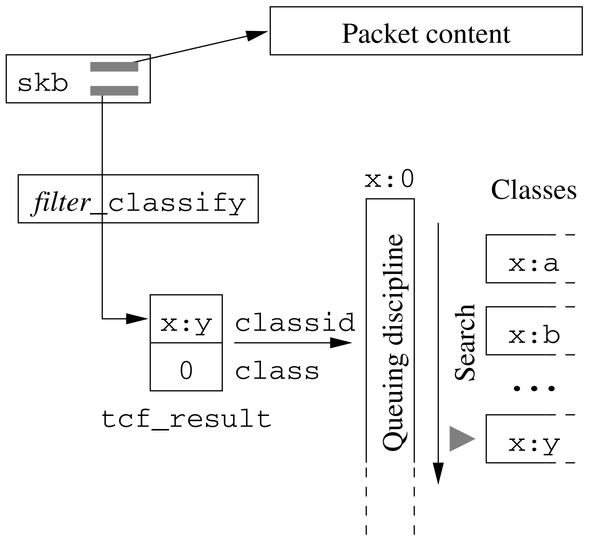


Figure 9: Generic filter.

Filters vary in the scope of packets their instances can classify: When using the `cls_fw` and `cls_route` filters, one instance per queuing discipline can classify packets for all classes. Those filters take the class ID from the packet descriptor, where it was stored before by some other entity in the protocol stack, e.g. `cls_fw` uses the marking functionality of the firewall code. We call such filters *generic*. They are illustrated in figure 9.

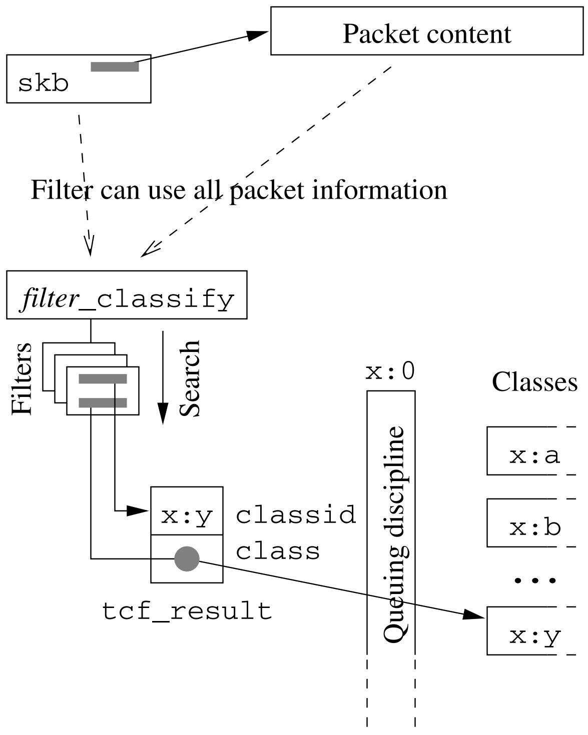


Figure 10: Specific filter, with a pointer to the class used as the internal class ID.

The other type of filters (`cls_rsvp` and `cls_u32`) needs one or more instances of the filter or its internal elements per class. We call such filters *specific*. Multiple instances of such a filter (or its elements) on the same filter list (e.g. for the same class) are distinguished by an *internal filter ID*, which is similar to the internal ID used for classes. However, unlike classes, filters have no "filter ID". Instead, they are identified by the queuing discipline or class for which they are registered, and their priority among the filters there.

Because specific filters have at least one instance or element per class, they can of course store the internal ID of that class and provide it as a result of classification. This then allows quick retrieval of class information by the queuing discipline. Figure 10 illustrates this scenario, where a pointer to the class structure is used as the internal ID. Unfortunately, generic filters have no means to provide this information. Therefore, they set the `class` field in `struct tcf_result` to zero and leave the lookup operation to the queuing discipline.

Starting with kernel version 2.2.5, also the generic filters `cls_fw` `cls_route` can become specific filters. This configuration change happens automatically when explicitly binding classes to them.

8 Policing

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

We consider four types of policing mechanisms: (1) policing decisions by filters, (2) refusal to enqueue a packet, (3) dropping of a packet from an "inner" queuing discipline, and (4) dropping of a packet when enqueuing a new one. Figures 11 to 15 illustrate the four mechanisms.

The first type of actions are decisions taken by filters (figure 11). The `classify` function of a filter can return three types of values to indicate a policy decision (the values are declared in `include/linux/pkt_cls.h`:

`TC_POLICE_OK` No special treatment requested.

`TC_POLICE_RECLASSIFY` Packet was selected by filter but it exceeds certain bounds and should be reclassified (see below).

`TC_POLICE_SHOT` Packet was selected by filter and found to violate the bounds such that it should be discarded.

Currently, the filters `cls_rsvp`, `cls_rsvp6`, and `cls_u32` support policing. The policing information is returned via `tc_classify` (in `include/net/pkt_cls.h`) to the `enqueue` function of the queuing discipline. It is then up to the queuing discipline to take an appropriate action. The queuing disciplines `sch_cbq` and `sch_atm` handle `TC_POLICE_RECLASSIFY` and `TC_POLICE_SHOT`. The `sch_prio` queuing discipline ignores any policing information returned by `tc_classify`.

Filters can use the function `tcf_police` (in `net/sched/police.c`) to determine if the flow they select conforms to a token bucket. The bucket parameters (declared in `struct tc_police` in `include/linux/pkt_cls.h` and later on stored in `struct tcf_police` in `include/net/pkt_sched.h`) are roughly the same as for TBF: maximum packet size (`mtu`), average rate (`rate`), peak rate (`peakrate`), and bucket size (`burst`). The field `action` contains the policy decision code returned when accepting the packet would exceed the limits. If

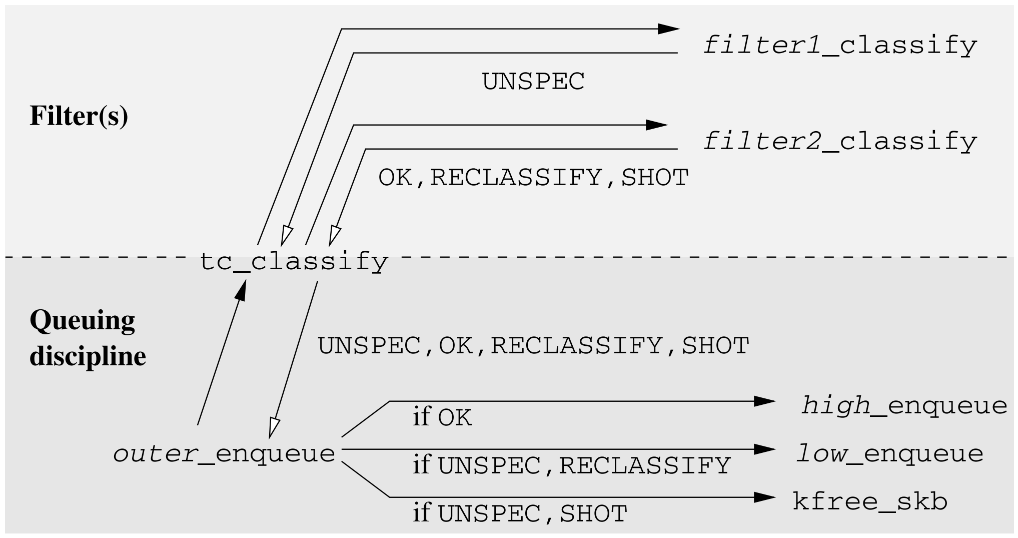


Figure 11: Policing when enqueuing; decision taken by filter.

the packet can be accepted, `tcf_police` updates the meter and returns the decision code stored in `result`.

If no matching filter was found, `tc_classify` returns `TC_POLICE_UNSPEC`. In this case, a queuing discipline will typically either discard the packet or treat it with low priority.

Sometimes, it is desirable to police traffic with respect to more than a single token bucket, e.g. to partition traffic into "low", "high", and "excess" packets. In order to build such configurations, multiple policing functions need to be consulted. To accomplish this, `tcf_police` returns `TC_POLICE_UNSPEC`, upon which the filter proceeds with the next element, or, if the current filter has no more eligible elements, the next filter is invoked. An example of such a configuration is given in [10].

Figure 12 illustrates how the matching process changes when policing is involved.

The second type of policing occurs when a queuing discipline fails to enqueue a packet (figure 13). In this case, it normally simply discards the packet (i.e. by calling `kfree_skb`). Some queuing disciplines also provide more sophisticated feedback to the calling queuing discipline and give it a second chance for enqueuing the packet: if the `reshape_fail` callback function has been set (in `struct Qdisc`), the "inner" queuing discipline may invoke it instead to allow the "outer" queuing discipline to select a different class. If `reshape_fail` is not set or if it returns a non-zero value, the packet must be discarded. Currently, only `sch_cbq` provides a `reshape_fail` function. `sch_fifo` and `sch_tbf` make calls to `reshape_fail`, if available.

The third policing mechanism is applied if a queuing discipline decides to drop a packet from an "inner" queuing discipline after that packet was enqueued, e.g. in order to create space for packets of a more important class (figure 14). This is done using the `drop` function. The `cbq_dequeue_prio` function of `sch_cbq` uses this via `cbq_under_limit` to remove packets from classes which are over limit.

Also the fourth mechanism (figure 15) discards packets that have already been successfully enqueued: if the `enqueue` function of a queuing discipline considers a new packet to be more important than some older one, it can discard the old packet and enqueue the new one instead. It indicates this to the caller by returning zero.

9 The sch_atm queuing discipline

As an example of how new traffic control elements can be added, we examine the ATM queuing discipline in more detail. It is used to re-direct flows from the default path (e.g. through a given interface) to ATM VCs. Each flow can have its own ATM VC, but multiple flows can also share the same VC. Figure 16 illustrates the structure of this queuing discipline.

While its classification and queuing part is fairly generic, the ATM queuing discipline differs from

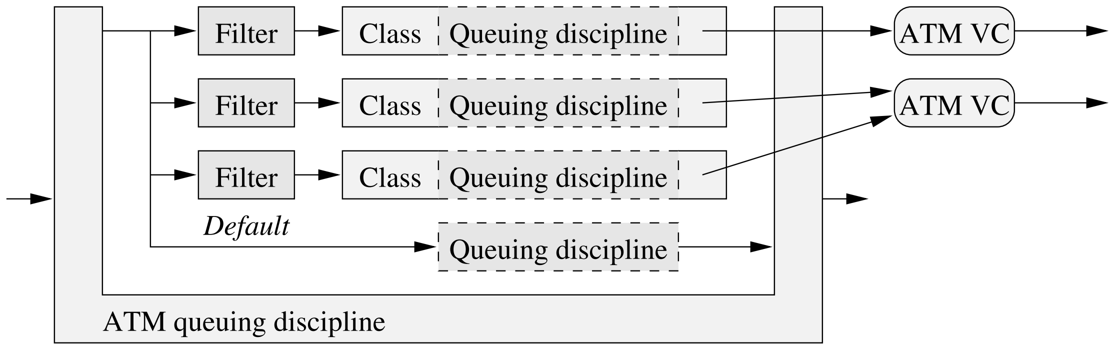


Figure 16: The ATM queuing discipline.

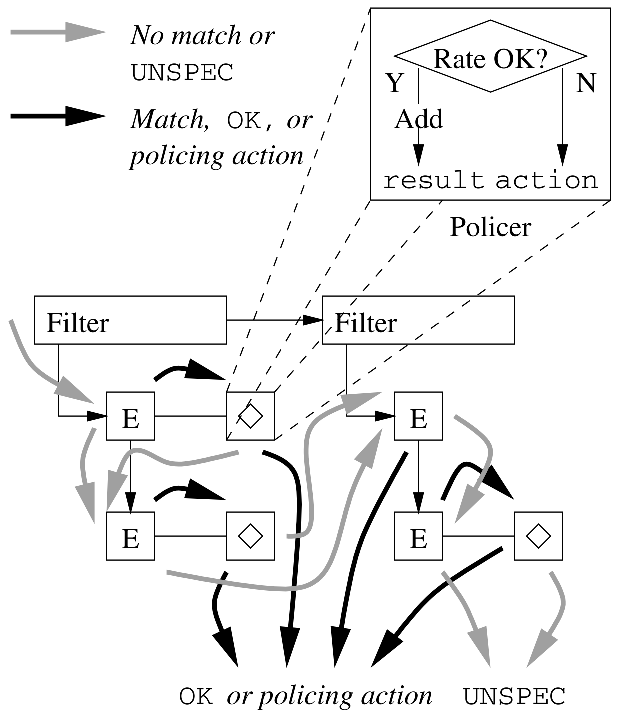
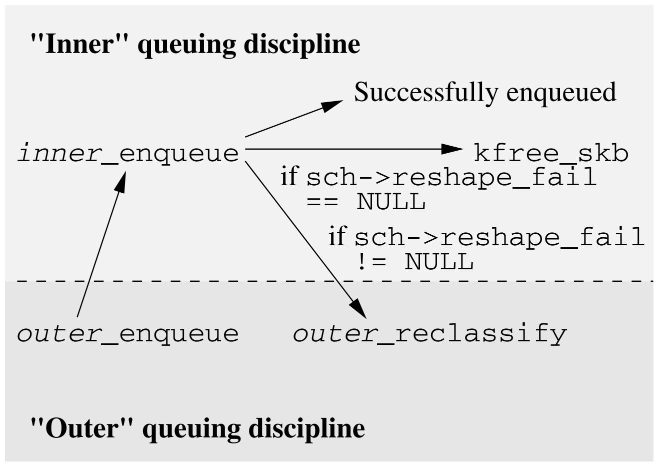


Figure 12: Looking for a match, with policing.



"Inner" queuing discipline
Successfully enqueued
inner_enqueue
kfree_skb
if sch->reshape_fail == NULL
if sch->reshape_fail != NULL
outer_enqueue
outer_reclassify
"Outer" queuing discipline

Figure 13: Policing when enqueuing; decision taken by "inner" queuing discipline.

other queuing disciplines in that packets enqueued on it may leave via other paths than through the `dequeue` function or being dropped: whenever `dequeue` is called, it first checks all inner queuing disciplines for packets to send, and sends them over the respective ATM VCs. After that, it returns whatever it gets from the default queue, which receives the packets that don't get attributed to any of the classes.

In order to prevent VCs from being removed while the queuing discipline is still using them, the reference count of the corresponding socket is increased when attaching a VC to a class of the ATM queuing discipline. This happens in the function `sockfd_lookup` in `net/socket.c` which `atm_tc_change` calls to translate the socket descriptor number to a pointer to the socket structure. When the class is removed, it returns the socket using `sockfd_put`, which then decrements the reference count. This pair of functions performs roughly the equivalent of `fdopen` and `close`.

The ATM queuing discipline supports the policing responses `TC_POLICE_SHOT` and `TC_POLICE_RECLASSIFY`. The latter can be handled in two different ways: (1) by assigning the packet to a new class (as configured by the user), or (2) by setting the cell loss priority bit in outgoing ATM cells.

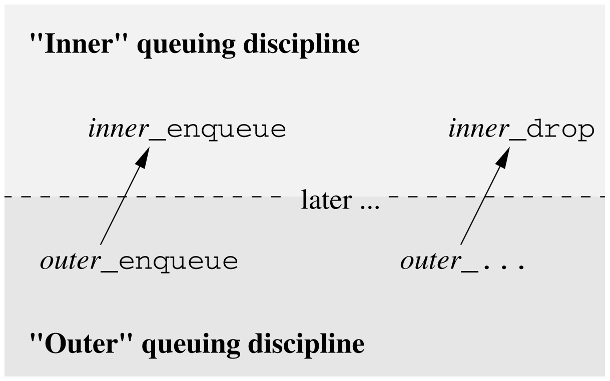


Figure 14: Policing after enqueuing; decision taken by "outer" queuing discipline.

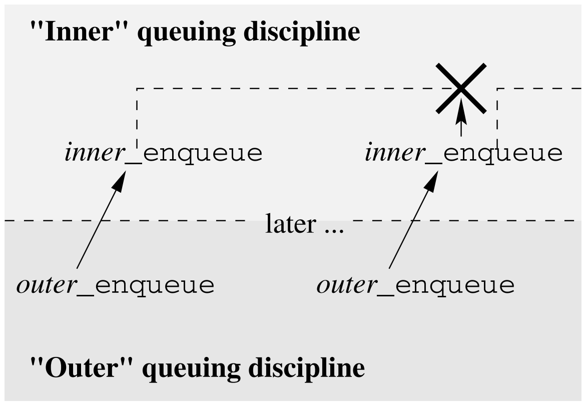


Figure 15: Older packet is discarded to make room for new packet.

The code of the ATM queuing discipline is in `net/sched/sch_atm.c`. In addition to that file, `include/linux/pkt_sched.h` contains the option types (prefix `TCA_ATM_`), and `net/sched/sch_api.c` contains the initialization. Furthermore, the usual changes had to be made to `net/sched/Config.in` and `net/sched/Makefile` to include the new queuing discipline in the configuration and build process.

The use of the ATM queuing discipline is described in the file `atm/extra/tc/README` in the ATM on Linux distribution.

10 Conclusion

Linux traffic control consists of a large variety of elements, which interact with each other in many ways. The modular approach chosen results in a very versatile design that can be readily applied to most current traffic control tasks, and which can be easily extended to accommodate less typical applications, such as the link-layer selection implemented in the ATM queuing discipline. It also forms the basis for the Linux implementation of Differentiated Services, which unify and advance many of the existing traffic control concepts.

We have described queuing disciplines, classes, filters, and elements within filters, we have illustrated the most important interactions between these components, and we have briefly introduced the design of a new queuing discipline. We hope this information to be useful for people aiming to understand the inner workings of Linux traffic control, and in particular also to implementors of new traffic control functions.

11 Acknowledgements

The author would like to thank Jamal Hadi Salim for very helpful discussions and suggestions on this paper, and Alexey Kuznetsov for a critical review and for explaining many of his design decisions and the deeper details of traffic control.

References

[1] Clark, David D.; Shenker, Scott; Zhang, Lixia. *Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism*, Proceedings of SigComm'92, Baltimore, MD, August 1992. `http://ana-www.lcs.mit.edu/anaweb/ps-papers/csz.ps`

[2] IETF, Integrated Services (intserv) working group. `http://www.ietf.org/html.charters/intserv-charter.html`

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

[4] Bernet, Yoram; Yavatkar, Raj; Ford, Peter; Baker, Fred; Zhang, Lixia; Nichols, Kathleen; Speer, Michael; Braden, Bob. *Interoperation of RSVP/Int-Serv and Diff-Serv Networks* (work in progress), Internet Draft `draft-ietf-diffserv-rsvp-02.txt`, February 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] RFC2205; Braden, Bob (Ed.); Zhang, Lixia; Berson, Steve; Herzog, Shai; Jamin, Sugih. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*, IETF, September 1997.

[7] 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.

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

[9] RFC2170; Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)*, IETF, July 1997.

[10] Almesberger, Werner; Hadi Salim, Jamal; Kuznetsov, Alexey. *Differentiated Services on Linux* (work in progress), Internet Draft `draft-almesberger-wajhak-diffserv-linux-01.txt`, May 1999.