Linux Network Traffic Control — Implementation Overview

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

February 4, 2001

Abstract

Linux offers a rich set of traffic control functions. This document gives an overview of the design of the respective kernel code and explains its structure.

Contents

TODO: condense

1 Introduction

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 (Integrated Services [2]), and has been further extended to serve as the basis for supporting the more recent work of "Diffserv" (Differentiated Services [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 2.4.0 (final).

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: packets arrive on via an input interface, where they may be policed. Policing discards undesirable packets, e.g. if traffic is arriving too fast. After policing, packets are 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 second 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

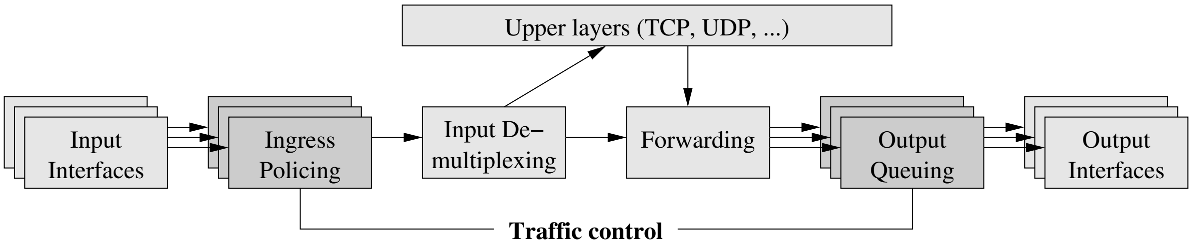

Figure 1: Processing of network data.

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.

Section 2 gives an overview and explains some terminology. Sections 3 to 6 describe the elements of traffic control in the Linux kernel in more detail. Section 7 discusses locking issues in traffic control.

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 (and related concepts)

Each network device has a *queuing discipline* associated with it, which controls how packets enqueued on that device are treated. Figure 2 shows the symbol we use for a queuing discipline without externally visible internal structure. For example, `sch_fifo` is such a simple queuing discipline, which just consists 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 shows an example of such a queuing discipline. Note that multiple filters may map to the same class.

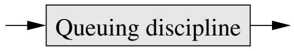

Figure 2: A simple queuing discipline without classes.

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 store 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 the *token bucket filter* (TBF) queuing discipline, 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

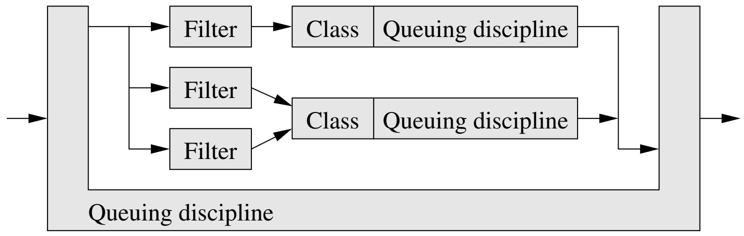

Figure 3: A simple queuing discipline with multiple classes.

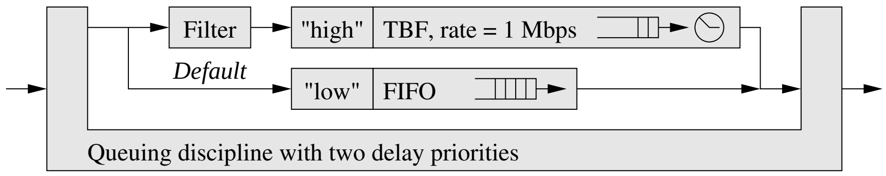

Figure 4: Combination of priority, TBF, and FIFO queuing disciplines.

`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 to which class they were attributed. 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. See section 6 for further details.

TODO: rubbish

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.

2.1 Changes from 2.2 to 2.4

The most visible conceptual change in traffic control between 2.2 and 2.4 kernels is the addition of the ingress queuing discipline (see section 6.2). Extensions for ATM and Diffserv, that existed as separate patches for 2.2, have been integrated into the mainstream kernel for 2.4.

This is the list of changes in detail:

- Addition of the ingress queuing discipline (`sch_ingress`)
- Addition of the ATM queuing discipline (`sch_atm`)
- Addition of support for Diffserv (`cls_tcindex`, `sch_dsmark`, `sch_gred`, `skb->tc_index` field)
- Fine-grained locking model and replacement of bottom-half handlers by more SMP-friendly soft-interrupts
- The return codes of `enqueue` and `requeue` functions have changed

2.2 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, e.g. for the Linux kernel this is usually `/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://ftp.inr.ac.ru/ip-routing/` or `ftp://ftp.funet.fi/pub/mirrors/ftp.inr.ac.ru/ip-routing/` (the latter is usually much faster).

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/v2.4/`.

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.

2.3 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`.

3 Queuing disciplines

Instances of queuing disciplines are identified by 32-bit numbers, which are split into a major and a minor number (currently 16 bits each). The usual notation is *major*:*minor*. For queuing disciplines, the minor number is always zero. Attempting to create a queuing discipline with a non-zero minor number yields an error.[1]

Major numbers have to be unique per interface, e.g. there can be only one queuing discipline 1:0 on `eth0`, but there may be some other queuing discipline 1:0 on, say, `eth1`.

Major numbers assigned by the user should be in the range 1...0x7fff. Major number 0 means "unspecified", and a major number in the range 0x8000...0xffff is automatically allocated by the kernel if creating a queuing discipline with an unspecified major number.

Numbers in the range ffff:fff0 to ffff:ffff are reserved or have a special meaning (e.g. ffff:ffff selects the top-level egress queuing discipline of an interface). Special values are defined in `include/linux/pkt_sched.h` and have names beginning with `TC_H_`.

Note that these major and minor numbers are not related to the major and minor numbers used for device special files.

When creating a new queuing discipline, the message sent to the kernel (`struct tcmsg` in `include/linux/rtnetlink.h`) contains the identification number in `tcm_handle` and the insertion point in `tcm_parent`. The value for the insertion point can be the number of an existing queuing discipline, or one of the special values `TC_H_ROOT` for the top-level queuing discipline on egress, or `TC_H_INGRESS` for the ingress queuing discipline.

Figure 6 illustrates the use of the parent in an example with a classful queuing discipline with hierarchical classes (e.g. CBQ). For simplicity, filters are not shown.

3.1 Functions

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 in-

[1] With the exception of ingress queuing disciplines, which use the special number ffff:fff1.

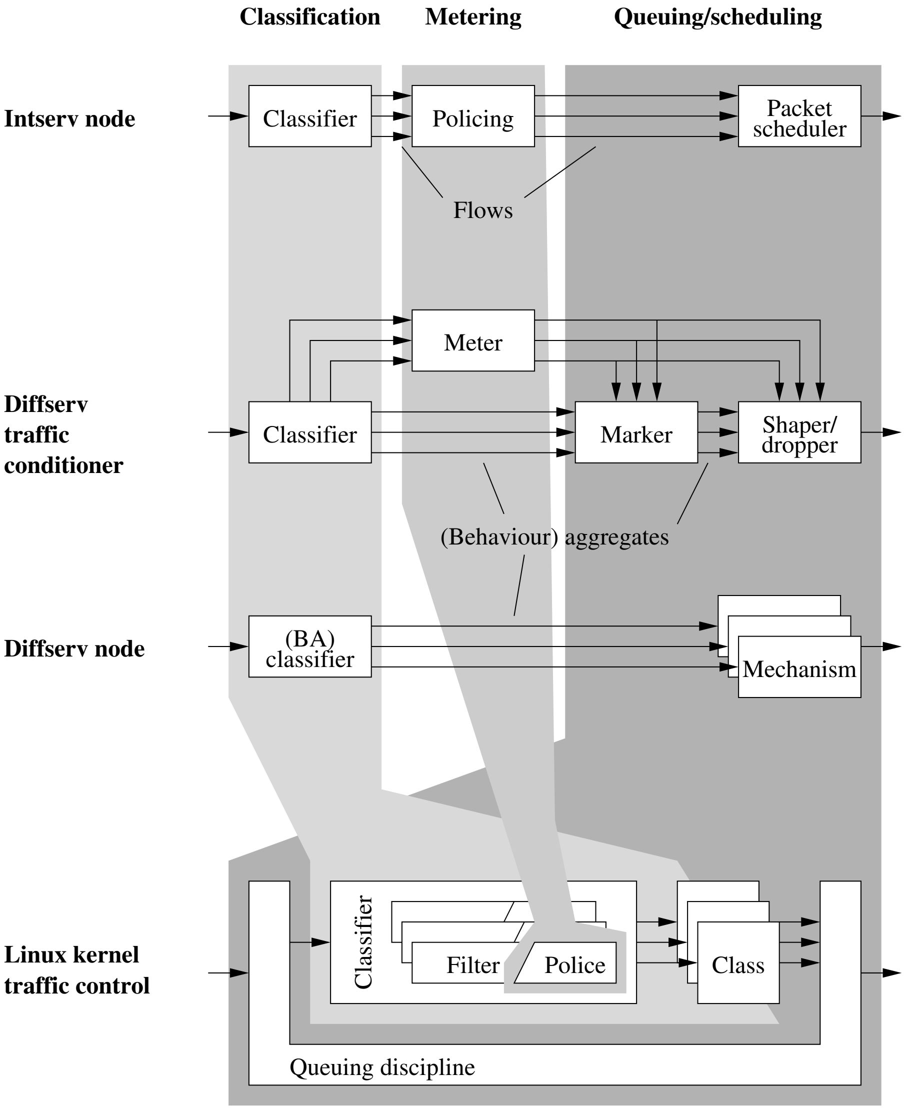

Figure 5: Relation of elements of the Intserv and Diffserv architecture to traffic control in the Linux kernel.

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.

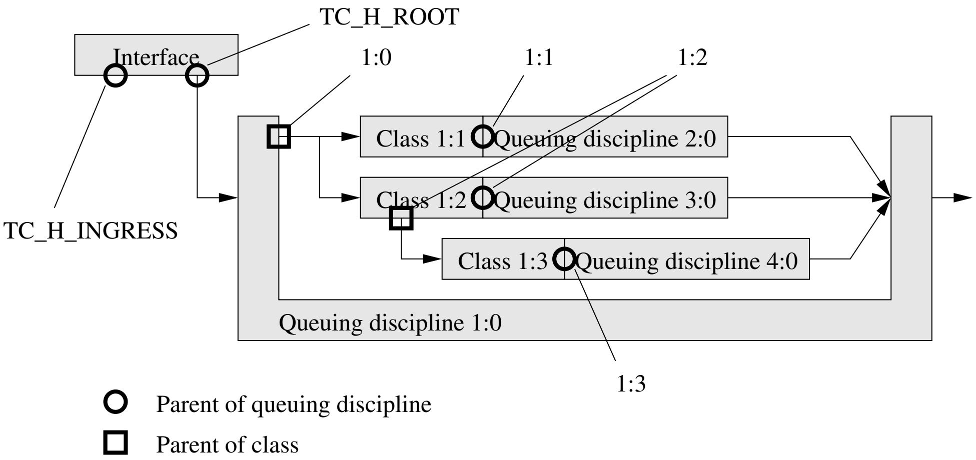

Figure 6: Addressing for queuing disciplines and classes.

vokes the `enqueue` function of the corresponding "inner" 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.

`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).

`change` changes the configuration of a queuing discipline.

`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 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 2.2.

3.2 Return values

The `enqueue` and `requeue` functions return 0 (`NET_XMIT_SUCCESS`) if a packet was successfully enqueued. Otherwise, they return one of the following values, defined in `include/linux/netdevice.h`:

`NET_XMIT_DROP` *a* packet was dropped

`NET_XMIT_CN` a packet was dropped due to congestion, and the sender should try to slow down

`NET_XMIT_POLICED` the packet was dropped by policing

`NET_XMIT_BYPASS` the packet was accepted, but will not leave the queuing discipline via the `dequeue` function

`NET_XMIT_DROP` and `NET_XMIT_CN` are somewhat vague, because they can also indicate that a packet other than the one passed to the enqueuing function was dropped. Note that it is not possible to drop more than one packet per invocation of an enqueuing function without confusing the caller's notion of the queue length (`sch->q.qlen`).

`NET_XMIT_BYPASS` is used by `sch_atm` to indicate that the packet must not be counted for queue length computation, but that no error has occurred. (Just returning `NET_XMIT_DROP` in such cases might cause the sender to slow down, hence the special return value.)

`drop` returns 1 if a packet was successfully dropped, 0 otherwise.

3.3 Statistics

Each queuing discipline must maintain at least the following statistics:

`sch->q.qlen` is the current queue length. This is the number of packets the queuing discipline will eventually deliver via its `dequeue` function. In particular, the dequeuing framework may stop polling a queuing discipline if `sch->q.qlen` is zero.

`sch->stats.bytes` is the cumulative number of bytes enqueued.

`sch->stats.packets` is the cumulative number of packets enqueued.

`sch->stats.drops` is the cumulative number of packets dropped.

The statistics are stored in `struct tc_stats`, declared in `include/linux/pkt_sched.h`. Classful queuing disciplines may also maintain similar statistics for each class.

Note that it is not always possible to provide exact counts in `sch->stats`, e.g. if an inner queuing discipline's `enqueue` function drops a packet different from the one being enqueued, it returns `NET_XMIT_DROP`, which is sufficient for correctly counting the number of packets in the queue, but not for also counting the number of bytes enqueued.

3.4 Data flow

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

`qdisc_run` immediately calls `net/sched/sch_generic.c:qdisc_restart`, 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_run` can also be indirectly 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. It does this by requesting a soft-interrupt (via `include/linux/netdevice.h:netif_schedule` and `__netif_schedule`), which then, via `net/core/dev.c:net_tx_action`, calls `qdisc_run`.

The transmit soft-interrupt (`include/linux/interrupt.h:NET_TX_SOFTIRQ`) is also scheduled whenever packets have been queued up for further processing.

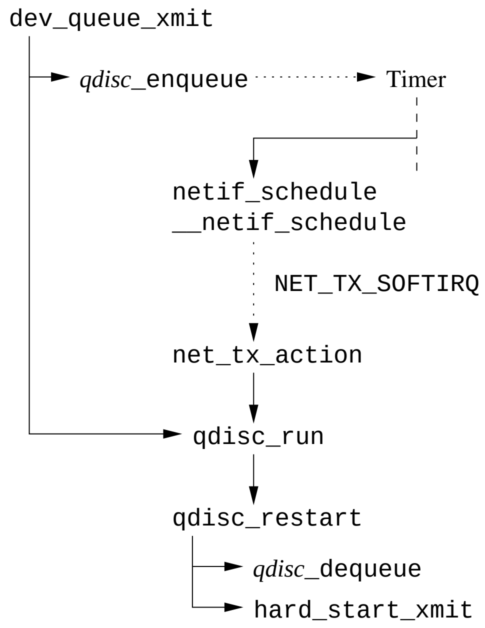

Figure 7: Functions called when enqueuing and sending packets.

Figure 7 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.

3.5 Adding a new queuing discipline

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

Configuration and status parameters specific to the new queuing discipline and its classes are listed in an enumeration type in `include/linux/pkt_sched.h`. Likewise, data structures use for configuration or status reporting are declared in this file.

4 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. The minor number can be in the range 0. . .0xffff.

Class IDs and internal class IDs are unique per queuing discipline. Since class IDs also contain the queuing discipline number, they are also unique per interface.

Like queuing disciplines, classes have parents. This is shown in the example in figure 6. Note that the same parent value (e.g. 1:2) can identify different locations when used for queuing disciplines or classes.

4.1 Functions

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 by any filters, and deactivates 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 informs the queuing discipline that an instance of a filter element is about to be bound to the class. bind_tcf is usually identical to get, except when the queuing discipline needs to be able to explicitly refuse class deletion if a class has filters attached to it, or if other sanity checks need to be performed. (E.g. sch_cbq refuses to delete classes while they are referenced by filters, and it also refuses the addition of filters elements selecting classes closer to the root of the class tree than the parent of the filter.)

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.

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.

4.2 Class selection

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 5. The return value of tc_classify is either -1 (TC_POLICE_UNSPEC) or the policing decision returned by the filter (see section 6). 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].

Also, the default queuing discipline (a three-band priority scheduler, confusingly called pfifo_fast, and defined in net/sched/sch_generic.c), selects the priority band exclusively based on skb->priority.

Note that kernels up to 2.2.3 limit the value that can be set with SO_PRIORITY to the range $0 \ldots 7$, so that this shortcut classification does not work. However, all classful queuing disciplines except sch_ingress 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 (1:0).

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.

5 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.

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.

5.1 Structure

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

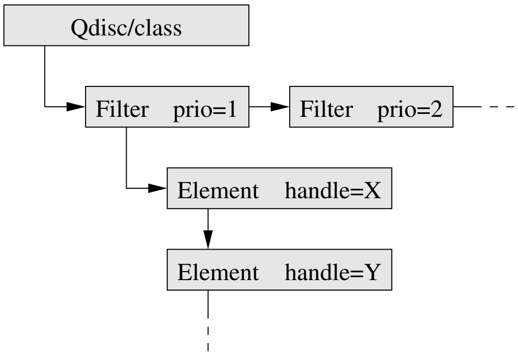

Figure 8: Structure of filters, with a list of elements belonging to the first filter, and no internal structure for the second filter.

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. Internal filter IDs are of type `unsigned long`, and the value 0 is reserved to indicate "not found". Handles and internal filter IDs they are unique per filter instance. The internal organization of a filter can be arbitrary. Figure 8 shows a filter with a list of internal elements.

Note that, unlike classes, filters or their elements have no "filter ID" that points directly to a specific filter or filter element on an interface. Instead, they are identified by the queuing discipline or class for which they are registered, the protocol, their priority among the filters there, and, in the case of filter elements, by their handle.

Figure 9 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. For example, one could also use a hash table, a tree, or other structures which make sense for the specific filter.

5.2 Functions

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

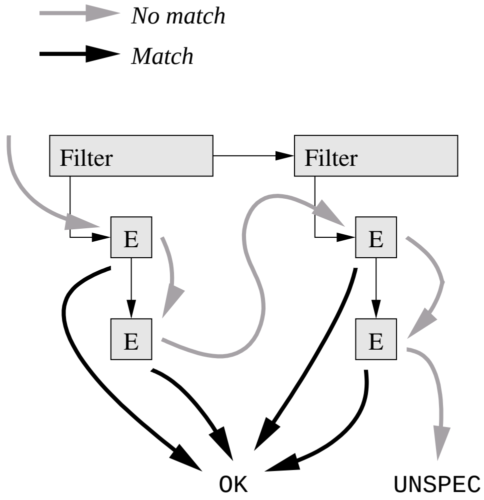

Figure 9: Looking for a match.

`classify` performs the classification and returns one of the `TC_POLICE_`... values described in section 6. 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_atm`, `sch_cbq`, `sch_dsmark`, and `sch_ingress` 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.

5.3 Filter types

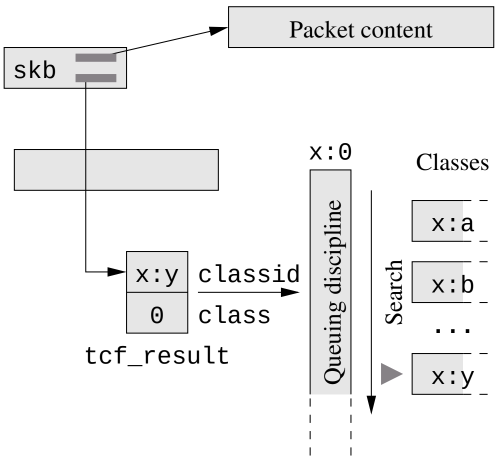

Figure 10: Generic filter.

Filters vary in the scope of packets their instances can classify: When using the `cls_fw`, `cls_route`, and `cls_tcindex` 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 10.

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*.

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

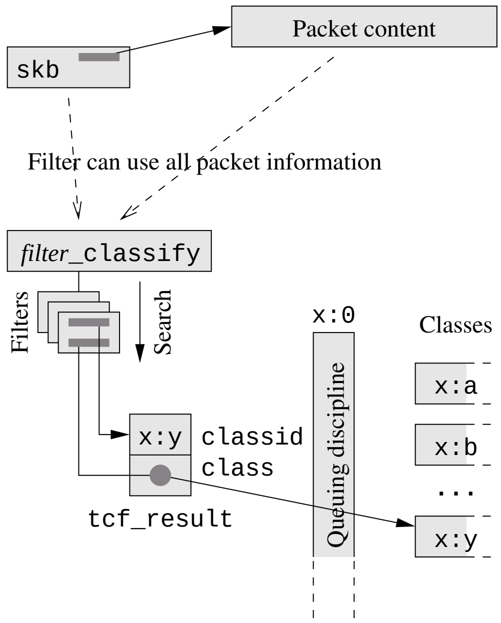

Figure 11: Specific filter, with a pointer to the class used as the internal class ID.

11 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. Similarly, `cls_tcindex` first tries to find a specific match, and calculates the result from `skb->tc_index` if there is none.

5.4 Adding a new filter

If a filter is compiled into the the kernel, it should be registered by `net/sched/cls_api.c:tc_filter_init`. Alternatively, is can also be registered from some other place using `net/sched/cls_api.c:register_tcf_proto_ops`, e.g. from the `init_module` function if the filter is compiled as a module.

Configuration and status parameters specific to the new filter are listed in an enumeration type in `include/linux/pkt_cls.h`. Likewise, data structures use for configuration or status reporting are declared in this file.

6 Policing and related concepts

The purpose of policing is to ensure that traffic does not violate certain constraints. 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 five types of policing mechanisms: (1) policing decisions by filters, (2) policing at ingress, (3) refusal to enqueue a packet, (4) dropping a packet from an "inner" queuing discipline, and (5) dropping a packet when enqueuing a new one. Figures 12 to 16 illustrate the four mechanisms.

6.1 Policing by filter

The first type of actions are decisions taken by filters (figure 12). The `classify` function of a filter can return four 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. (The value of `TC_POLICE_OK` is 0.)

`TC_POLICE_RECLASSIFY` Packet was selected by filter but it violates certain contraints and should be re-classified (see below).

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

`TC_POLICE_UNSPEC` The match by the filter is ignored, and matching should continue with the next filter element, or, if no further elements are eligible, the next filter. (The value of `TC_POLICE_UNSPEC` is -1.)

All filters 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_atm` and `sch_cbq` handle `TC_POLICE_RECLASSIFY` and `TC_POLICE_SHOT`. `sch_dsmark` and `sch_ingress` only handle `TC_POLICE_SHOT`. `sch_prio` 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 the packet can be accepted, `tcf_police` updates the meter and returns the decision code stored in `result`.

A filter must return valid class information (i.e. `class` and `classid`) if returning `TC_POLICE_OK` or `TC_POLICE_RECLASSIFY`.

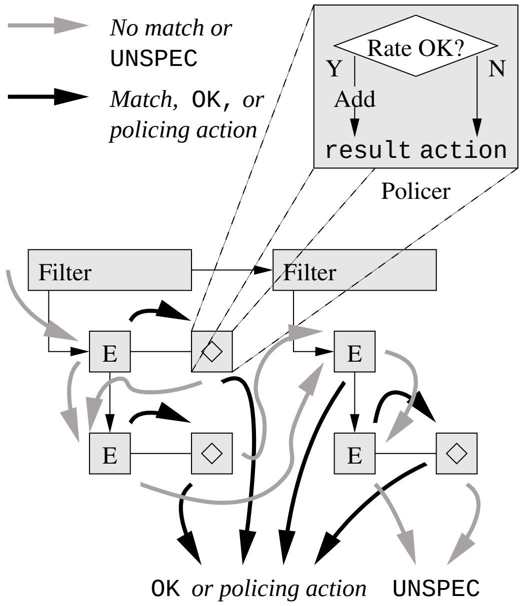

Figure 13: Looking for a match, with policing.

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].

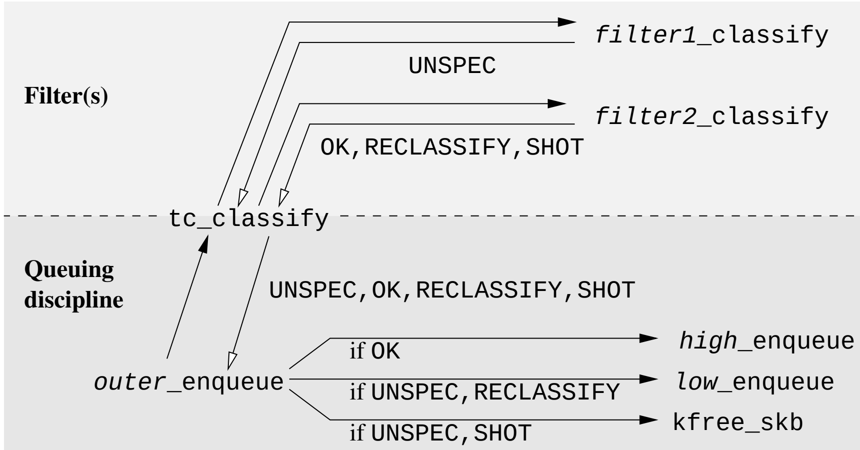

Figure 12: Policing when enqueuing; decision taken by filter.

Figure 13 illustrates how the matching process changes when policing is involved.

6.2 Policing at ingress

This is the most traditional approach of policing: undesired packets are discarded immediately after reception, without further wasting system resources on them.

The selection is accomplished by filters, like in the first case. Instead of one of the regular queuing disciplines, `sch_ingress` must be used. `sch_ingress` essentially provides a convenient framework for placing filters. It always returns immediately after the classification, without storing the packet in any queue. Also, its `enqueue` function returns netfilter (firewall) response codes (`NF_ACCEPT` or `NF_DROP`, defined in `include/linux/netfilter.h`), instead of the usual return values.

Note that trying to use any queuing discipline but `sch_ingress` at ingress is unlikely to yield a desirable effect.

6.3 Dropping by inner qdisc

The third type of policing occurs when a queuing discipline fails to enqueue a packet (figure 14). 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

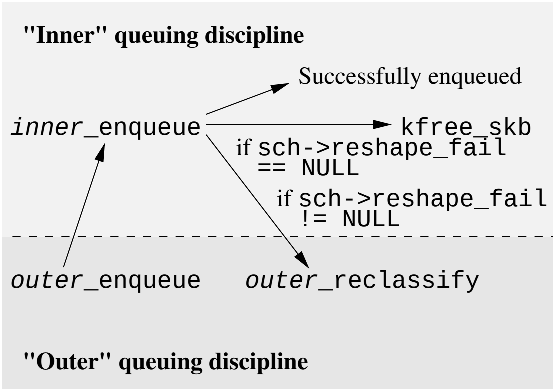

Figure 14: Policing when enqueuing; decision taken by "inner" queuing discipline.

"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.

6.4 Dropping from inner qdisc

The fourth 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 15). This is done using the `drop` function. The `cbq_dequeue_prio` function of `sch_cbq` uses this via `cbq_under_limit` to remove

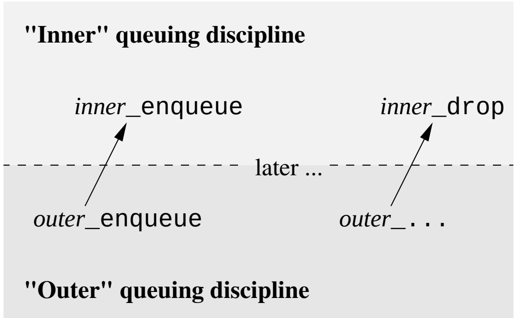

Figure 15: Policing after enqueuing; decision taken by "outer" queuing discipline.

packets from classes which are over limit.

6.5 Evicting a queued packet

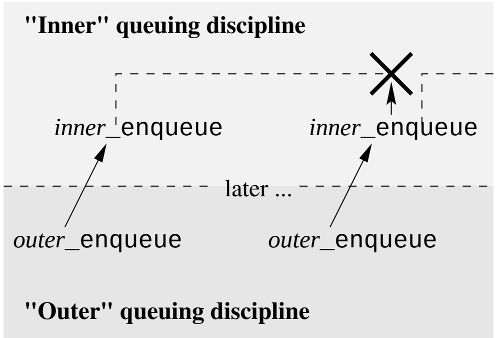

Figure 16: Older packet is discarded to make room for new packet.

Also the fifth mechanism (figure 16) 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.

7 Serialization

Data structures associated with traffic control elements are accessed in the following cases:

- when receiving a packet (at ingress)
- when enqueuing or dropping a packet
- when dequeuing or requeuing a packet (packet transmission)
- during configuration

In order to avoid race conditions, particularly on multi-processor systems, accesses must be serialized. Traffic control mainly relies on explicit locking for this purpose.

In this section, we will first give a brief overview of concurrency concepts, and synchronization mechanisms provided by the Linux kernel. Then we describe how the various activities affecting traffic control are initiated, and how they can lead to concurrency. Finally, we explain how accesses are serialized. We conclude this section with a call graph of the most important of traffic control and related components.

Further reading: [11] is a more comprehensive guide on locking mechanisms in the Linux kernel. A more detailed discussion of locking issues can be found in [12].

7.1 Concurrency issues

For traffic control, we consider user processes, hardware interrupts, and soft-interrupts. The latter are similar to hardware interrupts, but they are, as their name suggests, raised by software.

In the Linux kernel, there are three types of concurrent execution of code:

interruption occurs when an event of high priority (e.g. a hardware interrupt) needs to be handled while processing code of lower priority (e.g. a user process)

preemption occurs when a user process is temporarily suspended to run other processes, e.g. because the user process has used up its time slice, or because it is waiting for some event (e.g. user input)

parallel execution occurs in multi-processor systems

Hardware interrupts can interrupt soft-interrupts and user processes. Soft-interrupts can interrupt user processes. User processes can preempt other user processes. Parallel execution is possible among all three modes, e.g. in a multi-processor system, one CPU can run a user process, while another CPU is processing an interrupt handler.

Figure 17 shows how two CPUs in a multi-processor system execute various user processes, soft-interrupts, and a hardware interrupt.

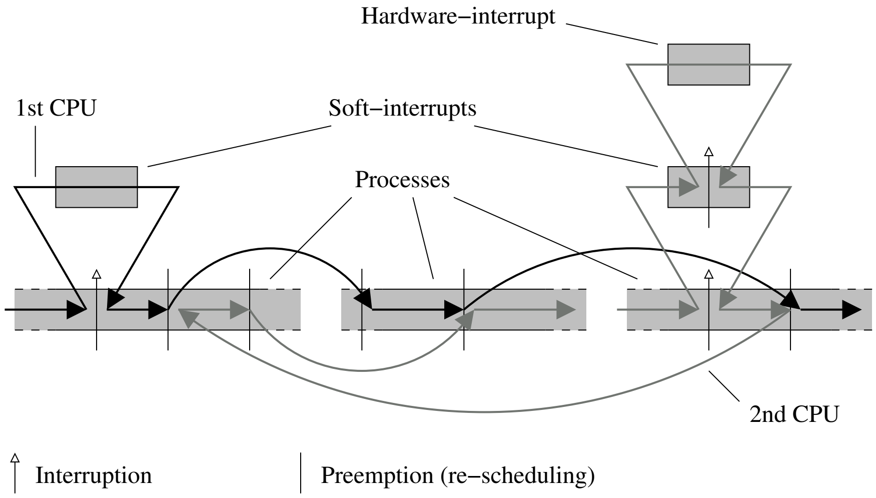

Figure 17: Concurrency examples for a dual-CPU system.

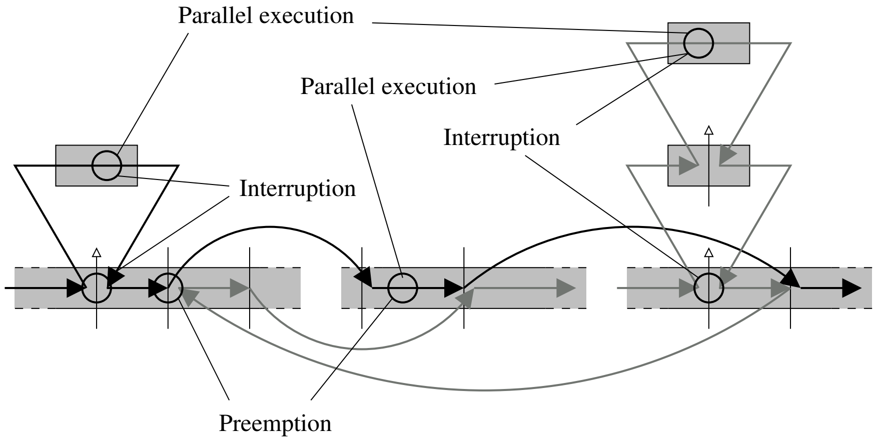

Figure 18: Conflicts caused by concurrent execution of a critical region.

Figure 18 illustrates how concurrency can cause conflicts by allowing execution of a critical region while the same critical region is already being executed somewhere else.

Note that a "critical region" is defined by accesses to shared data, and not necessarily by shared code.

7.2 Serialization mechanisms

The three basic mechanisms for serializing execution of concurrent regions are blocking of interrupts, looping until the critical region can be entered, and scheduling of other activity while the critical region is occupied.

Hardware interrupts can be blocked only on the same CPU. In order to prevent a hardware interrupt on another CPU from entering a critical region, a spinlock has to be used. (Spinlocks are described in [11].)

Soft-interrupts can be blocked on all CPUs. Furthermore, they have the property that each interrupt handler may run at most once per CPU, e.g. each CPU in a multi-processor system may run `net_rx_action`, but while `net_rx_action` or any function invoked by it is running, no further instance of `net_rx_action` will be started on the same CPU. Also, each CPU may only process one soft-interrupt at a time, i.e. soft-interrupts cannot interrupt each other.

Soft-interrupts are implemented mainly in kernel/softirq.c:do_softirq. Some related functions can also be found in include/linux/interrupt.h and include/asm/softirq.h.

Scheduling of other activities can be found in two quite different flavours: (1) user processes can suspend execution ("sleep") until the critical region is free; and (2) a soft-interrupt that needs to enter an already busy critical region can — instead of waiting, or if the critical region is kept busy by code that was interrupted by the soft-interrupt — simply raise itself again, and leave.

7.3 How activities are initiated

Packet reception can be triggered from various sources, e.g. from a hardware interrupt generated by a network device on packet reception, but in some cases also from a user process.[2] In all cases, the packet is passed to the stack via net/core/dev.c:netif_rx, which dispatches actual processing via the NET_RX_SOFTIRQ soft-interrupt, which in turn calls net/core/dev.c:net_rx_action.

Similarly, packet transmission is performed via the NET_TX_SOFTIRQ and then net/core/dev.c:net_tx_action, as described in section 3.4.

All configuration changes (i.e. creation, reconfiguration, and deletion of traffic control elements), and configuration dumps are initiated via the netlink interface. Those functions are always invoked from process context.

We group traffic control functions as follows:

Queuing invoked from a soft-interrupt, for arriving (ingress), and for departing packets

Configuration change invoked for creation, reconfiguration, and deletion of traffic control elements

Configuration dump invoked for dumping configuration, status, and statistics of traffic control elements

A function may be in more than one group.

Queuing discipline functions fall into those groups as follows:

Function	Queuing	Cfg. change	Cfg. dump
enqueue	•	·	·
dequeue	•	·	·
requeue	•	·	·
drop	•	·	·
init	·	•	·
reset	·	•	·
destroy	·	•	·
change	·	•	·
dump	·	•	•

For classes, the association is as follows:

Function	Queuing	Cfg. change	Cfg. dump
graft	·	•	·
leaf	·	•	·
get	·	•	•
put	·	•	•
change	·	•	·
delete	·	•	·
walk	·	•	•
tcf_chain	·	•	•
bind_tcf	·	•	·
unbind_tcf	·	•	·
dump	·	•	•

And finally, filters:

Function	Queuing	Cfg. change	Cfg. dump
classify	•	·	·
init	·	•	·
destroy	·	•	·
get	·	•	·
put	·	•	·
change	·	•	·
delete	·	•	·
walk	·	·	•
dump	·	•	•

Since configuration dumps only read data, they do not need to be serialized with respect to each other. However, all other groups may conflict. In the following three sections, we describe how the conflicts are eliminated.

7.4 Locking around traffic control

Probably the most important lock for traffic control is the per-device queue lock (a spinlock, field queue_lock of struct net_device in include/linux/netdevice.h). It serializes accesses to the queue(s)

[2] E.g. via the ATM over TCP device (drivers/atm/atmtcp.c) when using Classical IP over ATM (net/atm/clip.c).

of a device. dev_queue_xmit and net_tx_action obtain the queue lock before calling traffic control functions. Therefore, queuing functions are serialized when transmitting packets.

In order to synchronize with the queue lock from other places, also soft-interrupts have to be disabled, e.g. by using the functions include/linux/spinlock.h:spin_lock_bh and spin_unlock_bh.

All configuration changes are globally serialized by net/core/rtnetlink.c:rtnetlink_rcv via net/core/rtnetlink.c:rtnl_sem. Note that the call to include/linux/rtnetlink.h:rtnl_exlock_nowait in net/core/rtnetlink.c:rtnetlink_rcv_msg has no effect, because already the "shared" rtnl lock is implemented with exclusive locks.

Unlike configuration *changes*, configuration *dumps* are not entirely serialized with respect to configuration changes: while dumps are initiated with rtnl_sem held (by calling net/netlink/af_netlink.c:netlink_dump_start in net/core/rtnetlink.c:rtnetlink_rcv_msg), the dumping functions are invoked afterwards for providing more data when net/netlink/af_netlink.c:netlink_recvmsg is called. netlink_recvmsg calls net/netlink/af_netlink.c:netlink_dump, which then, via dumpit of struct rtnetlink_link, calls one of net/sched/sch_api.c:tc_dump_qdisc, net/sched/sch_api.c:tc_dump_tclass, or net/sched/cls_api.c:tc_dump_tfilter.

Note that the netlink/rtnetlink functions provide no synchronization with any soft-interrupts.

The mechanisms described in this section already avoid a large number of conflicts, but there are still some left:

- enqueue/dequeue versus configuration changes and dumps
- configuration changes versus configuration dumps

Note that it may seem that also transmit and receive operations need to be serialized with respect to each other, because their soft-interrupts may execute in parallel. This is not the case, because, at least as far as traffic control elements are concerned, no data is shared between ingress and egress.

7.5 Locking in traffic control framework

The traffic control framework handles most of the locking for configuration changes and configuration dumps. It serializes with enqueuing and dequeuing functions via the queue lock, and it synchronizes among configuration changes and dumps via the read/write lock net/sched/sch_generic.c:qdisc_tree_lock.

Since we usually have to grab both locks, include/net/pkt_sched.h provides the convenience functions sch_tree_lock, sch_tree_unlock, tcf_tree_lock, and tcf_tree_unlock.

Note that the serialization provided by rtnetlink and qdisc_tree_lock partially overlap. However, it is not possible to eliminate one of the two locks entirely: eliminating qdisc_tree_lock would cause races between configuration changes and dumps, because the latter are not entirely protected by rtnl_sem. Likewise, eliminating rtnl_sem would cause many races among functions changing configuration data.

While many functions are protected by qdisc_tree_lock and the queue lock, there are still several exceptions:

General

- change functions are only protected by rtnl_sem
- the dump functions are protected by rtnl_sem or qdisc_tree_lock, but never by the queue lock

Queuing disciplines

- init is only protected by rtnl_sem
- reset is usually invoked with rtnl_sem, qdisc_tree_lock, and the queue lock held, but it can also be called from a graft function

Classes

- leaf, delete, and graft are only protected by rtnl_sem
- bind_tcf is invoked from a filter's change function, and unbind_tcf is invoked from filter's change, delete, and destroy functions. They are therefore only only protected by rtnl_sem.
- get, put, tcf_chain and walk are protected by rtnl_sem or qdisc_tree_lock, but never by the queue lock

Filters

- init, get, put, and delete are only protected by rtnl_sem

- walk is protected by rtnl_sem or qdisc_tree_lock, but never by the queue lock

Also, the enqueue function of the ingress queuing discipline is not yet protected at all.

7.6 Locking in traffic control elements

One of the conflicts left from sections 7.4 and 7.5 is the serialization of the ingress queuing discipline's enqueue function with everything else. This is handled by getting the queue lock in net/sched/sch_ingress.c:ing_hook. This way, ingress behaves exactly like the egress queuing disciplines, as far as locking is concerned.

At the end of section 7.5, there is a large number of functions apparently only weakly protected. Fortunately, for most of them, only serialization with configuration changes is necessary (i.e. what is already provided by netlink_rcv). Since they never touch per-packet data, they do not need to hold the queue lock. Also, since they do not change the structure of traffic control elements, they can run in parallel to dumps. This includes all leaf, get, put, bind_tcf, unbind_tcf, tcf_chain, and walk functions. No rule without exception: net/sched/sch_cbq.c:cbq_put has to hold the queue lock while removing the short-cut q->rx_class. init is similar in that it is called only on a private copy of the queuing discipline or filter, so concurrent access could not happen yet.

dump can usually avoid explicit locking, because it only reads data. So it only needs to get the queue lock if it cannot obtain consistent information with an atomic read operation. Copying of extended statistics (xstats) in sch_cbq is an example for this.

graft has to hold the queue lock while exchanging the queuing disciplines and calling qdisc_reset. delete needs to hold the queue lock if making changes to the structure of elements (e.g. when deleting a class from the CBQ tree, or removing a filter element from a list). If the inner structure is simple enough (e.g. sch_prio), no locking is required.

change is a bit more complicated. First, if a change function makes non-atomic changes to the state of an active element, and this state is shared with the enqueue/dequeue functions, it needs to hold the queue lock.

For queuing disciplines, this is the only case where locking is needed in change. Classes and filters also need locking when making structural changes. For classes, this is the insertion of a new class or any other change in the class structure. For filters, it is the addition, removal, or replacement of the class associated with the filter element. This is done in the filter's change function, when processing the TCA_*filter*_CLASSID parameter.

Last but not least, change is also used for creating classes. In this case, no special locking is necessary when populating the class data structure, just like in the init functions of queuing disciplines and filters. However, addition of the class frequently requires locking.

7.7 Call graph with locks

Figure 19 shows a call graph containing all functions provided by traffic control elements, and their callers. For simplicity, some unusual calls and intermediate functions have been omitted in some cases.

a csz_dequeue makes this call via csz_enough_tokens and csz_update

b csz_enqueue makes this call via csz_update

c prio_init makes this call via prio_tune

d prio_tune only

e this call is usually made via class destruction, which may involve further function inside the respective traffic control element

f red_init only

TODO: include common helper functions with locks, e.g. policy

For further studying the locking architecture, sch_cbq and cls_u32 can serve as comprehensive examples that exercise almost all possible cases.

8 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.

We have described queuing disciplines, classes, filters, and elements within filters, and we have illustrated the most important interactions between these components. 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.

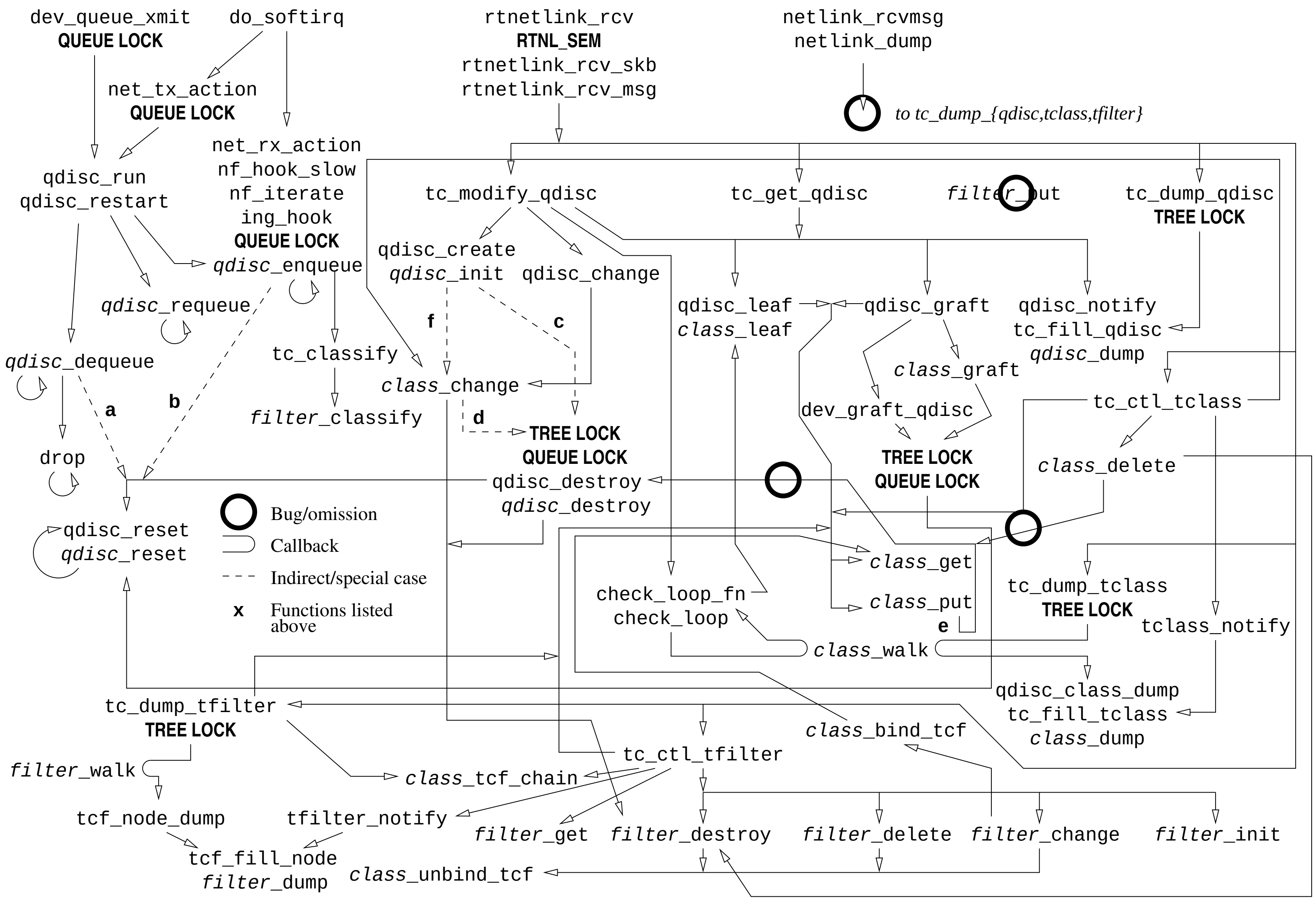

Figure 19: Call graph with locking (simplified).

9 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] RFC2998; Bernet, Yoram; Ford, Peter; Yavatkar, Raj; Baker, Fred; Zhang, Lixia; Speer, Michael; Braden, Bob; Davie, Bruce; Felstaine, Eyal; Wroclawski, John. *A Framework for Integrated Services Operation over Diffserv Networks*, IETF, November 2000.

[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.

[11] Russell, Paul. *Unreliable Guide To Locking*, `linux/Documentation/DocBook/kernel-locking.tmpl` or `http://netfilter.kernelnotes.org/unreliable-guides/kernel-locking/lklockingguide.html`

[12] Schimmel, Curtis. *Unix Systems for Modern Architectures – Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison-Wesley, 1994.