SocketCAN and queueing disciplines: Final Report

M. Sojka, R. Lisový, P. Píša
Czech Technical University in Prague

July 20, 2012
Version 1.2

Abstract

This document investigates the possibilities of using Linux traffic control subsystem with CAN traffic. By using queueing disciplines provided by this subsystem, one can solve certain type of priority inversion problems that arise from simultaneous use of a CAN interface by multiple applications. The document contains a brief introduction to traffic control under Linux, evaluates existing queueing disciplines for the use with CAN and gives the instructions for setting useful queueing discipline configurations. The queueing disciplines are also benchmarked to evaluate their fitness for small (and slow) embedded systems.

Contents

Document history

Version	Date	Description
1.1	2011-11-22	First public version
1.2	2012-07-20	Added description of `canid` ematch and the corresponding benchmarks. All examples with `can` filter were changed to use `canid` ematch. The description of `can` filter was shortened and the filter was marked as obsolete.

1. Introduction

In the past, most CAN bus devices (both embedded systems and host computers) run only a single application that communicated via CAN bus. With the birth of SocketCAN – the CAN subsystem of Linux – this paradigm changed. Now it is easily possible to run as many applications as one wants in a single Linux-based device. Due to the architecture of the Linux networking subsystem, this may lead to certain unwanted effects that would not exist if the applications were run on separate nodes. In this report, we examine the possibility of using Linux traffic control subsystem and its queueing disciplines with CAN networks as a way to remove or reduce the unwanted effects.

1.1. The problem

The unwanted effects mentioned above are in general called *priority inversion*. In case of CAN, it happens during frame transmission when high priority frames have to wait for lower priority ones to be transmitted. This is caused by the fact that, by default, there is only a single queue for storing the frames waiting for the transmission to the bus. Therefore, time sensitive frames may be delayed by an unacceptable amount of time because the kernel is processing non-time-sensitive frames submitted for transmission beforehand.

The simple solution to this problem is to *schedule* the frames waiting for transmission inside the kernel so that the high priority (the most time sensitive) frames are transmitted first.

To be more precise, there are two different reasons that cause priority inversion to happen:

Single TX queue Consider two independent applications transmitting frames in a single node. One application transmits one frame per second and it is required that the frames appear on the bus precisely in one second intervals. The second application uses some transport protocol such as ISO-TP and sends many frames in bursts but only from time to time and does not impose any timing constraints on these frames.

Since all transmitted frames are enqueued into a single queue, the frames of the former application have to wait until all ISO-TP frames of the second application are transmitted, which causes their timing requirements to be violated. The desired behavior is to interleave the frames of both applications so that the timing constraints are respected.

Priority/ID mismatch The second reason is that in some applications the CAN frame IDs (frame priorities) are not (for whatever reason) assigned according to the timing requirements for those frames. If a frame with strict timing requirements is assigned a low priority ID, it might be delayed in a CAN controller transmit buffer waiting for all higher priority frames being transmitted from other nodes.

In other words it means that the priority inversion can happen not only when two applications with different timing requirements reside in a single node but also when each of them resides in a different node.

1.2. Proposed solution

The solution to the above described problem is to develop a so called queueing discipline (qdisc) that will schedule frames in a way that the priority inversion does not occur or the probability of its occurrence is lowered. Linux kernel already contains many ready-to-use qdiscs so we investigate here their suitability for use with CAN.

Figure 1.1 shows the structure of queueing discipline that, we think, is able to fulfill the most requirements of the typical CAN application. In particular:

1. Some applications (App 1 and 2 in the figure) generate urgent frames that must be transmitted as fast as possible. This is achieved by storing those frames in a queue with the highest priority. Frames from this queue are transmitted preferably to anything else.

2. For the lower priority frames, there is a fairness requirement. When multiple applications send frames with equal priority, they are served in more or less round-

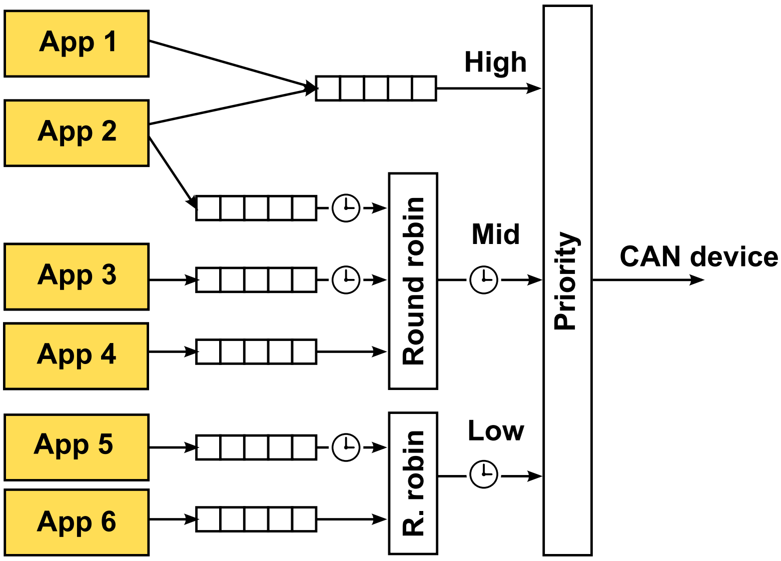

Figure 1.1.: Ideal CAN queueing discipline.

robin fashion. In the figure, this is represented by "Round-robin" boxes for middle and low priorities.

3. It should be possible to throttle the traffic generated by selected applications. This is represented by leftmost clock symbols in the figure.
4. Throttling is not only useful for individual applications but also for sets of applications. The rightmost clock symbols in the figure represent throttling of the joint traffic with certain priority. This may be used to not fully load the bus so that other nodes have a chance of transmitting their frames. This can help in the case of priority/ID mismatch mentioned above.

The rest of this document is structured as follows. Section 2 summarizes the terminology of Linux traffic control and should give the reader basic understanding of traffic control functionality and usage. Then, in Section 3, we describe which qdiscs are suitable for use with CAN and how to use them. Section 4 deals with benchmarks performed on particular qdiscs used with SocketCAN. We conclude the report in Section 5.

Appendix A contains real-world examples of qdisc configurations with short explanations. Appendix B shortly mentions qdisc available in Linux traffic control subsystem not suitable to be used with SocketCAN.

1.3. Acknowledgment

This work was financially supported by Volkswagen AG. The authors would like to thank Oliver Hartkopp for his feedback on this work.

2. Linux traffic control basics

This section tries to summarize the terminology used in Linux traffic control and the basic principles of its operation.

2.1. Terminology

From our experience, the terminology in this area is sometimes confusing. Therefore, we try to summarize the meaning of terms used in this document. Besides Linux source code, we took the information from [1, 2].

Qdisc (queueing discipline) is the basic element in Linux traffic control. Different qdiscs vary widely in what they actually are. A common denominator of all qdiscs is that they represent an *algorithm* to enqueue and dequeue packets. Besides the algorithm the qdiscs can *optionally* consist of the actual queue for queueing packets, one or more classes, a classifier and filters.

Qdiscs can be hierarchically composed. Every *classful* qdisc (i.e. qdisc with classes) may have attached internal child-qdiscs to some of the classes. Simple qdiscs that do not allow attaching of child-qdiscs are called *classless*.

Classful qdiscs typically do not enqueue packets by themselves. Instead, they use the child-qdiscs to perform the queueing.

The top-level qdisc (attached to the network device) is called *root* qdisc. The *root* qdisc is the only qdisc accessed directly by the networking subsystem when it transmits a packet.

Class is an abstract container within the qdisc that is used for internal routing of the packets going through the qdisc. Qdiscs supporting multiple classes can split the traffic to several classes and treat each class differently. For instance, different classes can have different priority.

Classes do have child-qdiscs attached to them. Child-qdiscs only deal with the traffic assigned to the class they are attached to.

Classifier determines the class the packet is classified to. It can be hardcoded in the qdisc or it can use one or more filters to perform the classification.

Filter User configurable algorithm to classify packets to classes. A filter can be attached only to a classful qdisc.

Packet/Frame A unit of network communication. Frame is used in CAN networks, packet is the term from IP networks. Qdiscs work on the level of packets/frames, which are represented by `sk_buff` structure. In this document we use these two terms interchangeably.

skb The abbreviation for `sk_buff` structure, i.e. the structure used by Linux kernel to describe received or to-be-transmitted packets.

Flow/Stream A sequence of frames/packets that are logically related to each other.

SFF/EFF Abbreviation for CAN frame formats i.e. Standard Frame Format and Extended Frame Format.

Work-Conserving Qdisc Qdisc that never delays a packet if the network adaptor is ready to send one.

2.2. Packet flow through qdiscs

The kernel only interacts with the root qdisc, i.e. it enqueues/dequeues packets to/from root qdisc and it never touches directly the internal qdiscs. These internal qdiscs are handled internally by their parent qdiscs according to their algorithm. This means that when the packet is being enqueued, it traverses the qdisc hierarchy from the root to the leaf qdisc, where it may stay queued for some time. When it is being dequeued, it goes in the opposite direction.

In case of CAN, frames are enqueued in `af_can.c` in function `can_send()` as it calls `dev_queue_xmit()`. Dequeueing happens when the kernel thinks it is appropriate, which is either directly after the frame is enqueued or sometime later in `NET_TX_SOFTIRQ`.

2.3. The `tc` tool

Traffic control is configured by the *tc* tool which is a part of *iproute2* package. It allows to configure how the qdiscs are interconnected and how packets are classified into classes.

To use the tool, it is important to understand how qdiscs and their classes are identified. Every qdisc or class is identified by a handle composed of major and minor numbers that are separated by a colon, e.g. 1:3. Qdiscs have minor numbers equal to zero and classes of the particular qdisc have the same major number as the qdisc and nonzero minor number. Refer to Figure 2.1 for an example of qdisc and class IDs.

2.3.1. Configuring qdiscs/filters

Qdiscs, filters and some types of classes are created and configured by the `tc` tool. The main command used for most of the tasks looks as follows:

```
tc ${ENTITY} ${COMMAND} dev ${DEV} parent ${PARENT} handle ${HANDLE} ...
```

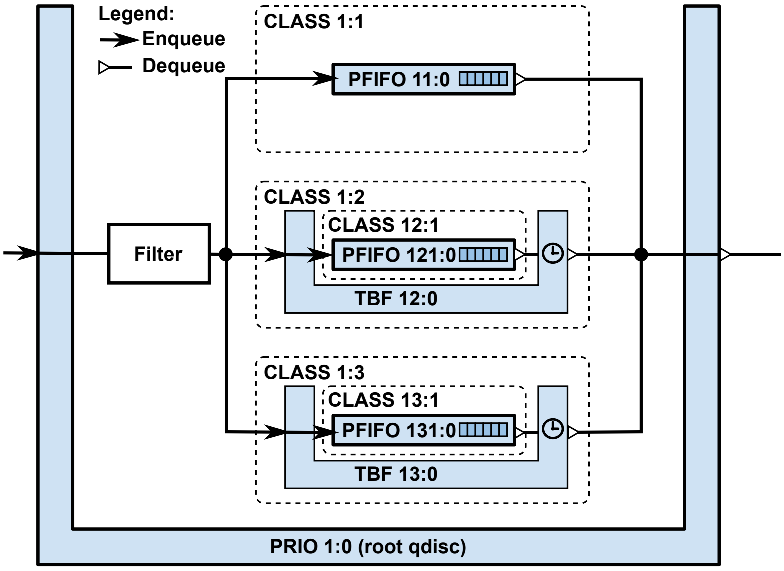

Figure 2.1.: Graphical representation of hierarchical qdisc structure. Queueing disciplines are shown as blue shapes. Only the pfifo qdiscs actually queue the packets. Token bucket filter (TBF) is one particular qdisc used to rate-limit/throttle traffic.

`ENTITY` determines the type of object, which will be configured. Possible values are: `qdisc`, `filter`, `class`.

`COMMAND` states the type of operation to be executed. It can be one of `add`, `del`, `change`, `show` (and some other, not so common, commands).

`DEV` is the name of the particular device, e.g. `can0`.

`PARENT` is the handle of a qdisc/class the configured `ENTITY` belongs to. When creating *root* qdisc, “`parent ${PARENT}`” is replaced with keyword `root`.

`HANDLE` specifies a *handle* of the particular `ENTITY`, which is used for refering to the entity later.

`...` are replaced with the command specific for the particular qdisc/filter/class.

Some basic examples of how to use the `tc` tool follows:

- Creating a qdisc:

```
tc qdisc add dev can0 root handle 1: prio
```

- Deleting a (root) qdisc:

```
  tc qdisc del dev can0 root
```

- Creating a filter (see also Section 3.2.6):

  ```
  tc filter add dev can0 parent 1:0 prio 1 handle 0xa \
      basic match canid\(sff 0x123 sff 0x500:0x700\) flowid 1:1
  ```

- Changing a filter:

  ```
  tc filter change dev can0 parent 1:0 prio 1 handle 0xa \
      basic match canid\(sff 0x111 eff 0x111\) flowid 1:1
  ```

- Showing information about a filter:

  ```
  tc filter show dev can0
  ```

- Deleting a filter:

  ```
  tc filter del dev can0 parent 1:0 prio 1 handle 0xa can
  ```

See Appendix A for more advanced real-world examples of how to use the `tc` tool. For detailed information see `tc` manual page.

2.3.2. Obtaining information about configured qdiscs

When investigating already configured qdiscs, it is possible to use diagnostic commands of the `tc` tool. The command to show static information about the configuration is:

```
tc [-d|-s] $ENTITY show dev $DEV
```

where `$ENTITY` can be one of the `qdisc`, `filter`, `class`. Optional parameter `-d` can be used to show more detailed information (only some qdiscs actually implement it).

To show much more *dynamic* information, there are statistics available for each configured qdisc and class. To show them, use the above command (with `$ENTITY` set to one of `class` or `qdisc`) with `-s` parameter.

3. Using qdiscs for controlling CAN traffic

This section gives the reader the detailed information about how to use queuing disciplines for CAN traffic. After reading this section, one should be able to configure Linux traffic control to implement the ideal CAN qdisc from Figure 1.1.

3.1. Suitable qdiscs

There are many qdiscs available in the Linux traffic control subsystem. This section describes the qdiscs suitable for using with CAN. For a brief analysis of remaining qdiscs see Appendix B.

Most of the subsections contain an example `tc` command showing how to set up the particular queueing discipline.

3.1.1. pfifo/bfifo

Qdisc called `pfifo` is one of the simplest qdiscs. It represents a simple queue of fixed length. In the case of queue overflow, the enqueued packets are dropped.

An example of how to configure `pfifo` qdisc with its size limited to 50 packets:

```
tc qdisc add dev can0 root handle 1: pfifo limit 50
```

For more information see `tc-pfifo` manual page.

The `bfifo` qdisc is almost the same as `pfifo`. The only difference is that the limit is given in bytes instead of packets. As a CAN frame has always a fixed size (typically 16 bytes), `bfifo` does not offer any functionality that is not available in `pfifo`.

3.1.2. pfifo_fast

`pfifo_fast` is the default qdisc and it is only a bit more complicated than `pfifo`. The reason why this qdisc is *fast* is that it does not maintain any statistics. It contains three internal queues (called *bands*) of different priority – the packets are enqueued into them based on `skb->priority` field (see Section 3.2.5). The priority is used as an index to *priomap*, which is a hardcoded table[1] used to map the priority to the band. Band 0 has the highest priority and dequeueing happens in the order of decreasing priority.

[1] `http://lxr.linux.no/#linux+v3.0.4/net/sched/sch_generic.c#L414`

The `pfifo_fast` qdisc has no configuration parameters. The size of the queue is determined from `txqueuelen` parameter of the particular device. This parameter is configured with `ip` tool. For example:

```
  ip link set can0 txqueuelen 100
```

For more information see `tc-pfifo_fast` manual page.

3.1.3. pfifo_head_drop

Like `pfifo` but in contrast to it, this queueing discipline drops the earliest enqueued packet in the case of queue overflow. As a result the queue contains always the freshest packets.

The command used to configure such a qdisc with queue length of one packet is:

```
tc qdisc add dev can0 root handle 1: pfifo_head_drop limit 1
```

3.1.4. prio

The `prio` qdisc is a simple classful queueing discipline that contains an arbitrary number of classes (also called bands) of different priority. When dequeueing, class 0 is tried first and only if it does not deliver a packet dequeueing continues with class 1, then with class 2 and so on. In case of three classes (i.e. the default setting), the most urgent packets should be classified into class 0, best effort packets into class 1 and background packets into class 2.

The following command can be used to create `prio` qdisc with 5 classes:

```
tc qdisc add dev can0 root handle 1: prio bands 5
```

The classifier implemented in `prio` qdisc offers three methods for how the packet can be classified.

1. If `skb->priority` is greater 0x10000 and higher 16 bits of `skb->priority` match the major number of the qdiscs handle, then the lower 16 bits are used as an one-based(!) index into priomap (see below).

2. Otherwise, user configurable filters are consulted for classification. See Section 3.2 for more details about CAN frame classification. There is also an example of `prio` qdisc with filters in Figure 4.2.

3. If no filters are attached to the `prio` qdisc or none of them matches the packet, `prio` uses `skb->priority` field, which is in this case less than 0x10000, for classification. The priority is used as an zero-based index to `priomap` the same way as for `pfifo_fast`.

Unlike for `pfifo_fast` the `prio` allows to chnage the `priomap`. The default value is (1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1) and it can be changed with the following command:

```
tc qdisc add dev can0 root handle 1: prio bands 5 priomap 0 1 2 3 4
```

This command sets `priomap` so that the priority of the skb represents directly the priority of the packet.

For more information see `tc-prio` manual page.

3.1.5. TBF

Token Bucket Filter (TBF) is a classful queueing discipline (with only one internal class). Packets are enqueued without limitations, during dequeueing TBF ensures that the outgoing data rate does not exceed some administratively set rate, but with the possibility to allow short bursts in excess of this rate. The rate of these bursts can also be limited. The default qdisc created inside of the TBF is `bfifo`.

TBF may be set with the following command (for more information about used units see `tc` manual page):

```
tc qdisc add dev can0 root handle 1: \
    tbf rate 0.1mbit burst 160b latency 70ms
```

The `rate` parameter specifies the outgoing rate. In case of CAN the rate does not precisely correspond to the actual rate on the CAN bus. TBF tries to calculates the real bus rate from `skb->len`, which is always 16 bytes in case of CAN, and from the table (TCA_TBF_RTAB) sent to the kernel by the `tc` tool. The content of the table can be influenced by the following parameters passed on `tc` command line: `mpu`, `linklayer`, `overhead`, `mtu`, but setting these parameters has not much sense for CAN. With default settings, TBF assumes that every transmitted CAN frame is 128 bits long. This roughly corresponds to 8 byte CAN frames, which are, depending on the number stuffed bits, about 120 bits long.

The above command therefore limits the traffic to $0.1 \times 10^6/128 = 781.25$ CAN frames per second. The second parameter (`burst`) specifies the size of burst. Since every CAN frame counts for 16 bytes, the configuration in the example allows for sending maximum 10 frames without traffic limiting. The smallest working size of burst parameter seems to be 24 bytes. This is probably due to some rounding errors inside TBF. With smaller value, TBF always returns `ENOBUFS` error to the application.

The last parameter in the example, `latency` (or `limit`) specifies how many frames can be stored in the TBF's queue. The latency is converted to the number of bytes, which are used as the limit for the internal `bfifo` qdisc.

For more information see `tc-tbf` manual page.

3.1.6. SFQ

Stochastic Fairness Queueing (SFQ) is a classless queueing discipline that attempts to fairly distribute opportunity to transmit data to the network among an arbitrary number of flows. It accomplishes this by using a hash function to separate the traffic into one of the 128 internally maintained FIFOs which are dequeued in round-robin fashion. In the case of CAN protocol, the input of hashing function is the address of the originating socket (`skb->sk`). This means that SFQ distinguishes at least between different applications.

The command used to setup SFQ qdisc is as follows:

```
tc qdisc add dev can0 root handle 1: sfq perturb 10
```

Parameter `perturb` sets the interval in seconds after which the hashing algorithm is changed. If two sockets hashes to the same FIFO, it is likely that this will be no longer true after 10 seconds. For more information see `tc-sfq` manual page.

3.1.7. HTB

Hierarchical Token Bucket (HTB) is an advanced qdisc which allows for complex setting of how is the available bandwidth distributed between different flows. HTB instances contain classes, which are organized hierarchically. The leaf classes have attached another qdisc, by default `pfifo_fast`.

HTB ensures that the amount of bandwidth provided to each class is at least the minimum of *the amount it requests* and *the amount assigned to it*. When a class requests less than the assigned bandwidth, the remaining (excess) bandwidth is distributed to other classes with the same parent and which request the service – the default scheme is to distribute the excess bandwidth to the other classes in proportion to their allocations. It is also possible to prioritize some classes so that the excess bandwidth is offered to them in the first place. It is also possible to limit maximum excess bandwidth that a class can use (i.e. how much bandwidth can the class “borrow”). Child classes can never use more bandwidth than it is allocated to their parent.

An example configuration of HTB qdisc and its classes with bandwidth limits is as follows:

```
tc qdisc add dev can0 root handle 1: htb

tc class add dev can0 parent 1:  classid 1:1  htb rate 100kbps ceil 100kbps
tc class add dev can0 parent 1:1 classid 1:10 htb rate  30kbps ceil 100kbps
tc class add dev can0 parent 1:1 classid 1:11 htb rate  10kbps ceil 100kbps
tc class add dev can0 parent 1:1 classid 1:12 htb rate  60kbps ceil 100kbps
```

It must be noted that the rate specification is probably a subject to the same limitations as in case of TBF (see Section 3.1.5). For more information see `tc-htb` manual page.

3.2. Classifying CAN frames

Network packets can be classified either by hardcoded qdisc classifiers, which typically classify the packets based on the various fields of `sk_buffer` (e.g. by `sk_buffer->priority`), or/and by user configurable filters. This section describes two types of user configurable filters that can be used for CAN as well as the method of setting `sk_buffer->priority` which is used by hardcoded classifiers.

3.2.1. u32 filter

The filter called `u32` can be used to classify packets based on any values in the packet. For CAN traffic this means that `u32` can classify packets based on CAN IDs, packet length (`can_dlc`) and/or the data payload.

`u32` filter is configured using *selectors*. A selector specifies the size of the field to be matched (one byte (`u8`), two bytes (`u16`) or four bytes (`u32`)), the offset of this field in the packet (as it is stored in an `skb`) and a mask. Internally, `u32` uses several hash tables to locate the selectors and quickly find whether the packet matches or not.

There is one issue when using `u32` filter for matching CAN frames – CAN ID is always stored in the `skb` in the *native* endianness of the particular computer architecture, whereas `u32` filter expects the values to be stored in network byte order, i.e. big-endian.

CAN ID is stored in the first 32 bits of the `skb` (see Figure 3.1). To set a rule for matching CAN ID 0x1 on x86 (little-endian) architecture, it is necessary to specify the ID converted to the network-order (big-endian), which equals to 0x01000000.

The following command sets up the `u32` filter that matches CAN frames with ID 0x1 on x86 architecture and CAN frames with ID 0x01000000 on big-endian architectures such as PowerPC:

```
tc filter add dev can0 parent 1:0 prio 1 \
    u32 match u32 0x01000000 0xffffffff at 0 flowid 1:1
```

When matching single byte values, endianness is no more an obstacle. An example of how to match frames of different sizes is:

```
tc filter add dev can0 parent 1:0 prio 1 \
    u32 match u8 0x01 0xff at 4 flowid 1:1
tc filter add dev can0 parent 1:0 prio 2 \
    u32 match u8 0x02 0xff at 4 flowid 1:2
```

Here, frames with 1 byte of data payload are classified into class 1:1 and frames with 2 bytes of data payload into class 1:2.

3.2.2. Basic filter and extended matches

Extended match (ematch) is a simple classifier to classify packets based on various criteria. There are many different ematches suitable for different needs (e.g. *meta*,

```
typedef __u32 canid_t;

struct can_frame {
    canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8    can_dlc; /* data length code: 0 .. 8 */
    __u8    data[8] __attribute__((aligned(8)));
};
```

Figure 3.1.: The exact position of particular fields in CAN frame (defined in `include/linux/can.h`).

u32, *text*, etc.). Ematches can be connected with logical conjunctions to form *ematch expression*. Basic filter (`cls_basic.c`) is a filter that evaluates ematch expressions to classify packets.

Ematch support in Linux kernel has to be enabled with CONFIG_NET_EMATCH option in kernel configuration file. An example of using basic filter with ematch expression is as follows:

```
tc filter add dev can0 parent 1:0 \
    basic match \
    u32\(u8 0x8 0xFF at 4\) AND \
    \( u32\(u32 0x01020304 0xFFFFFFFF at 8\) OR \
       u32\(u32 0x01020304 0xFFFFFFFF at 12\) \) \
    flowid 1:11
```

Here, `u32` ematch, whose functionality is similar to `u32` filter described above in Section 3.2.1, is used to match CAN frames carrying 8 bytes of data with 0x01020304 value in either first or second half of data payload.

The syntax of an ematch expression is as follows:

```
Usage: EXPR
where: EXPR  := TERM [ { and | or } EXPR ]
       TERM  := [ not ] { MATCH | '(' EXPR ')' }
       MATCH := module '(' ARGS ')'
       ARGS := ARG1 ARG2 ...

Example: a(x y) and not (b(x) or c(x y z))
```

3.2.3. The canid ematch

Using `u32` ematch for CAN identifier matching would lead to the same endianness problems as `u32` filter. For that reason `canid` ematch was developed. This ematch can be used like any another ematch in a *basic* filter and it can be combined with arbitrary queueing disciplines.

The sources of the `canid` ematch (`em_canid.c`) are available in our repository[2] and the particular patch is already on its way to be merged in Linux 3.6.

To use this ematch, it is necessary to modify the `tc` tool. The needed modifications are available in another repository[3]. This will be submitted upstream as soon as the kernel patches are merged.

The syntax of `canid` ematch is as follows:

```
Usage: canid(IDLIST)
where: IDLIST := IDSPEC [ IDLIST ]
       IDSPEC := { 'sff' CANID | 'eff' CANID }
       CANID := ID[:MASK]
       ID, MASK := hexadecimal number (i.e. 0x123)
```

IDLIST represents one or more (up to 500) match rules. Keywords `sff` and `eff` determine the type of the CAN frame to match. When matching an exact CAN ID, mask (e.g. 0x7ff for SFF frame) is optional.

An example of the command configuring `canid` ematch is:

```
tc filter add dev can0 parent 1:0 basic \
    match canid\(sff 0x123 sff 0x500:0x700 eff 0x00:0xff\) \
    flowid 1:1
```

This filter matches SFF frames with CAN IDs 0x123 and 0x500–0x5ff and EFF frames whose IDs end with 0x00. All those frames will be classified into class 1:1.

Internals of canid ematch

The match rules for EFF frames are stored in an array, which is traversed during classification. This means that the worst-case time needed for classification increases with the number of configured rules.

The ematch implements an optimization for matching SFF frames using a bitmap with one bit used for each ID. With this optimization, the classification time for SFF frames is nearly constant and independent of the number of configured rules.

3.2.4. Evaluation order of filters

Neither documentation nor other resources define in which order are individual filters without given priority traversed. This is not important in most cases unless a match-everything filter is used to match packets unmatched by the other filters. Such filter must be evaluated only as the last one. The observed behavior on Linux 3.4 (with `iproute2`

[2] `https://rtime.felk.cvut.cz/gitweb/lisovros/linux_canprio.git/shortlog/refs/heads/em_can`

[3] `https://rtime.felk.cvut.cz/gitweb/lisovros/iproute2_canprio.git/shortlog/refs/heads/em_can`

package in version corresponding to the kernel) seems to be that the filters are evaluated in the oposite order than they were added – thus the match-everything filter should be added as the first one.

3.2.5. Socket priority

Some qdiscs, such as `prio` or `pfifo_fast` can classify packets according to `skb->priority` field. For IP networks, this field is set automatically by the IP stack based on Type Of Service (TOS) field of the IP header. Alternatively, it can be set by the originating socket by setting `SO_PRIORITY` socket option in user-space. To use this type of classification for CAN traffic, a simple patch[4] for the kernel is needed.

3.2.6. The can filter (obsolete)

Prior developing `canid` ematch, we developed a standalone filter with the same functionality as the ematch. Since the ematch is more flexible and its code is about 50 % shorter than the code of the filter, we consider the filter as obsolete. However, some benchmarks in Section 4 were performed only with the standalone filter and not with the ematch. For interested readers, the sources of the filter are available in our repository[5].

3.3. Qdiscs and virtual CAN interface

It is possible to use qdiscs with virtual CAN interface (`vcan`). This can be used, for example, to roughly simulate the transmission delay of the real CAN interfaces. One only needs to insert `vcan` module with `echo` parameter set to 1, e.g.:

```
# modprobe vcan echo=1
# ip link add vcan0 type vcan
# ip link set vcan0 up txqueuelen 100
# ip link show dev vcan0
12: vcan0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc noqueue state UNKNOWN qlen 100
    link/can
```

By default, there is no qdisc attached (which is different from real CAN interfaces). The qdisc can be attached the same way as shown above, e.g.:

```
tc qdisc add dev vcan0 root tbf rate 1mbit burst 24b limit 10000
```

This command makes `vcan0` to behave almost like real 1 Mbit CAN interface. See Section 3.1.5 for description of TBF parameters.

[4] http://rtime.felk.cvut.cz/gitweb/lisovros/linux_canprio.git/commitdiff/da8161bbd1a4b1b13ddba598e8c3f24657ea9878

[5] https://rtime.felk.cvut.cz/gitweb/lisovros/linux_canprio.git/blob/canprio:/net/sched/cls_can.c

3.4. Blocking the application when the queue is full

Many SocketCAN users experience a problem with `write()/send()` failing with `ENOBUFS` error. Since this is related to the use of queueing disciplines, this section describes why it happens and what can be done against it.

In the default configuration, CAN interfaces have attached `pfifo_fast` queuing discipline which, when enqueueing the packet, checks whether the number of queued packets is greater then `dev->tx_queue_len` (which is 10 for CAN devices by default). If it is the case, it returns `NET_XMIT_DROP` which is translated to `-ENOBUFS` in `net_xmit_errno()` called from `can_send()`. The problem is, that there is no way for the application to be blocked until the queue becomes empty again.

How can be the application made to block when the queue is full instead of getting `ENOBUFS` error? In general there are two mechanisms that limit the number of queued packets. First, there is the already mentioned per-device `tx_queue_len` limit and second, the per-socket `SO_SNDBUF` limit. The application only blocks when the latter limit is reached. Therefore, the solution is to set `SO_SNDBUF` low enough that this limit is reached before `tx_queue_len` limit.

Now, the question is to which value set the `SO_SNDBUF` limit. First, the minimum value is `SOCK_MIN_SNDBUF/2`, i.e. 1024. When the user supplies a smaller value the minimum is used instead. The more tricky thing is how is the value interpreted. The value represents the maximum socket send buffer in bytes. The kernel always doubles the value supplied by user (i.e. for the kernel the minimum is 2048) and stores it in `sk->sk_sndbuf`. When a packet is sent, a per-socket counter is increased by `sizeof(can_frame) + sizeof(skb)` (which is a value around 200, depending on kernel configuration and architecture). When the counter is greater or equal to `sk->sk_sndbuf`, the application blocks.

The following piece of code sets the `SO_SNDBUF` value to its minimum:

```
int sndbuf = 0;
if (setsockopt(s, SOL_SOCKET, SO_SNDBUF, &sndbuf, sizeof(sndbuf)) < 0)
        perror("setsockopt");
```

Typically, the minimum value causes the application to block when there are about 15 frames queued. If we want all CAN applications in the system to block instead of receiving `ENOBUFS`, it is necessary to set the `txqueuelen` (see Section 3.1.2) to the number of simultaneously used CAN sockets in the system multiplied by 15.

If the application does not wish to block, it sets `O_NONBLOCK` flag on the socket by using `fcntl()` call. After that, when the `SO_SNDBUF` is reached, the application receives `EAGAIN` error instead of `ENOBUFS`.

3.5. Summary

To summarize this section, we return to Figure 1.1 and match the blocks from that figure to the particular qdiscs described above.

The rectangle labeled “Priority” can be implemented by `prio` qdisc configured as a root qdisc. The clock symbols can be implemented by `tbf` and round-robin scheduling (even with throttling) by `htb`. If it is not required to throttle individual applications (clock symbols left to round-robin boxes), the round-robin can be also implemented by `sfq`. The queues in the figure (five rectangles) are created as default child qdiscs of `prio` or `tbf`. If it is desired to limit their length or change the behavior, they can be created explicitly, perhaps as `pfifo` or `pfifo_head_drop` qdiscs.

The selection of which qdisc should handle which frames can be configured by attaching `basic` filters with `canid` ematches to the root `prio` qdisc and/or to `htb` qdisc(s).

4. Benchmarks

In order to evaluate the cost of individual queueing disciplines and of our filter implementation, we conducted several experiments to measure the time spent in `can_send()` function (in the kernel).

Measurements were performed by using `function_graph` tracer in *ftrace* – the Linux kernel function tracer. This particular tracer timestamps entry and exit points of the traced functions and stores the duration of the function execution. Ftrace was configured in *dynamic* mode as shown in Figure 4.1. This allows us to trace only `can_send()` function, not affecting the performance of any other (not traced) function.

```
FTRDIR=/sys/kernel/debug/tracing
sysctl kernel.ftrace_enabled=1
echo "function_graph" > ${FTRDIR}/current_tracer
sleep 1
echo "can_send" > ${FTRDIR}/set_ftrace_filter
echo 1 > ${FTRDIR}/tracing_on
```

Figure 4.1.: Exact configuration of ftrace function tracer.

The first set of experiments was performed with the original `can` filter implementation (Section 3.2.6). After the `canid` ematch was developed, we compared the performance of the original filter with the ematch. Since there are no significant differences between the two, the results of the original experiments are still relevant.

4.1. Qdisc comparison

In this set of experiments we compared the performance of different qdiscs. As was described above, the time spent in `can_send()` was measured with *ftrace*. Measurements were performed for 5000 CAN frames generated by `cangen` utility.

4.1.1. Test environment

The tests were conducted on two different computers:

PC-P4 is a PC with Intel(R) Pentium(R) 4 CPU 2.40 GHz with 1 GB of RAM running 2.6.36.2 Linux kernel with custom `.config`. The used CAN interface card was *PCIcan-Q* card from Kvaser AB (PCI ID 10e8:8406).

MPC5200 is an embbeded PowerPC CPU (e300 core, G2_LE), 396 MHz, with 128 MiB of RAM running 2.6.36.2 Linux kernel with custom `.config`.
Root filesystem of this embedded computer was stored on a remote computer, accessed through NFS protocol. To not influence the networking subsystem during benchmarks, all the measured data were stored into directory mounted as `tmpfs` (located only in RAM) and copied to the NFS filesystem afterwards.

In both computers CAN devices were configured with the following commands:

```
ip link set can0 type can bitrate 1000000
ip link set can0 txqueuelen 1000
```

CAN traffic was generated with the command:

```
cangen can0 -I $ID -L 8 -D i -g $GAP -n 5000
```

where

- `$ID` denotes CAN frame ID. It was set either to some fixed value or as "`i`" which causes the ID to be incremented for every frame.
- `$GAP` denotes the delay in milliseconds between each sent frame. Different tests used different values (e.g. 0, 1, 2).

4.1.2. Tested configurations

The configurations of different tests are summarized in Table 4.1. The tests differ in qdisc and filter configuration and `cangen` parameter `-I` (CAN frame ID). All tests except `prio128_array` were performed with CAN filter compiled with SFF frame rules stored as a bitmap. For `prio128_array`, the SFF frame rules were stored in the array (see Section 3.2.6 for details).

Test	**Qdisc configuration**	**cangen params.**
pfifo_fast	The default qdisc (pfifo_fast) automatically set when running `tc qdisc del dev $DEV root`.	`-I 123`
pfifo_fast_inc	Same as *pfifo_fast*.	`-I i`
prio1	Simple `prio` qdisc with four priorities. Configured with the commands from Figure 4.2.	`-I 123`
prio1_inc	Same as *prio1*.	`-I i`
prio128	Prio qdisc with similar configuration as the one above, using 128 rules for each filter.	`-I i`
prio128_array	Same as *prio128* except that the can filter was recompiled to store SFF rules in an array instead of in a bitmap.	`-I i`
htb	HTB qdisc configured as shown in Figure 4.3.	`-I 123`

htb_inc	Same as for *htb* experiment.	`-I i`
prio_sfq	Prio and SFQ qdiscs configured as shown in Figure 4.4. Because of fixed size of SFQ qdisc, modified cangen (able of setting SO_SNDBUF) was used.	`-I 123`
prio_sfq_inc	Same as for *prio_sfq* experiment.	`-I i`

Table 4.1.: Configurations of the qdisc experiments.

4.1.3. Results

The graphs in Figures 4.5 and 4.6 show the execution times of `can_send()` function for different queueing disciplines and platforms. They show the minimum measured time and its 50th (median) and 90th percentile. Maximums are not shown as the measured time includes not only the execution of `can_send()` but also all hard- and soft-irqs that interrupted the execution of `can_send()`. Therefore, the real maximum was an order of magnitude higher than 90th percentile and as such it is not important for interpretation of the results.

Figure 4.5 shows the results for virtual CAN interface (`vcan`). Here, the measured times include the processing time of both the qdisc (enqueued and dequeue operations[1]) and `vcan` driver.

The top graph depicts the times measured on PC. The gray (left) bars show the results for flood traffic (i.e. frames were sent as fast as possible), yellow (right) bars show the results when frames were sent with 1 ms delay. In case of `vcan` interface, there is no fundamental difference between these two cases. It can be seen that `pfifo_fast` is really fast and the other qdiscs are slower. In the case of traffic with delays, the measured times

[1] The exception was `pfifo_fast`, because is has `TCQ_F_CAN_BYPASS` flag set, which means that the qdisc was completely bypassed in this case.

```
tc qdisc add dev ${DEV} root handle 1: prio

tc filter add dev ${DEV} parent 1:0 prio 1 handle 0xa \
    can sffid 0x123:0xffff sffid 0x124:0xffff flowid 1:1
tc filter add dev ${DEV} parent 1:0 prio 2 handle 0xb \
    can sffid 0x125:0x7ff effid 0x125:0x7ff flowid 1:2
tc filter add dev ${DEV} parent 1:0 prio 3 \
    can sffid 0x223:0xffff flowid 1:2
tc filter add dev ${DEV} parent 1:0 prio 4 \
    can sffid 0x0:0x0 effid 0x0:0x0 flowid 1:3
```

Figure 4.2.: Qdisc configuration of `prio1` experiment.

```
#Create root qdisc
    tc qdisc add dev ${DEV} root handle 1: htb
#Create individual HTB classes
    tc class add dev ${DEV} parent 1: classid 1:1 \
       htb rate 100kbps ceil 100kbps
    tc class add dev ${DEV} parent 1:1 classid 1:10 \
       htb rate 30kbps ceil 100kbps
    tc class add dev ${DEV} parent 1:1 classid 1:11 \
       htb rate 10kbps ceil 100kbps
    tc class add dev ${DEV} parent 1:1 classid 1:12 \
       htb rate 60kbps ceil 100kbps
#Add filters
    tc filter add dev ${DEV} parent 1:0 prio 1 \
       can sffid 0x123:0x7ff effid 0x123:0x7ff flowid 1:10
    tc filter add dev ${DEV} parent 1:0 prio 2 \
       can sffid 0x124:0x7ff effid 0x124:0x7ff flowid 1:11
    tc filter add dev ${DEV} parent 1:0 prio 3 \
       can sffid 0x125:0x7ff effid 0x125:0x7ff flowid 1:12

    tc qdisc add dev ${DEV} parent 1:12 handle 40: sfq perturb 10
```

Figure 4.3.: Qdisc configuration of `htb` experiment.

```
qdisc add dev ${DEV} root handle 1: prio

tc filter add dev ${DEV} parent 1:0 prio 1 \
       can sffid 0x111 effid 0x111 flowid 1:1
tc filter add dev ${DEV} parent 1:0 prio 2 \
       can \
       sffid 0x123 effid 0x123 \
       sffid 0x124 effid 0x124 \
       sffid 0x125 effid 0x125 \
       flowid 1:2
tc filter add dev ${DEV} parent 1:0 prio 3 \
       can sffid 0x0:0x0 effid 0x0:0x0 flowid 1:3 # default class

# SFQ
tc qdisc add dev ${DEV} parent 1:2 handle 10: sfq perturb 10
tc qdisc add dev ${DEV} parent 1:3 handle 11: sfq perturb 10
```

Figure 4.4.: Qdisc configuration of `prio_sfq` experiment.

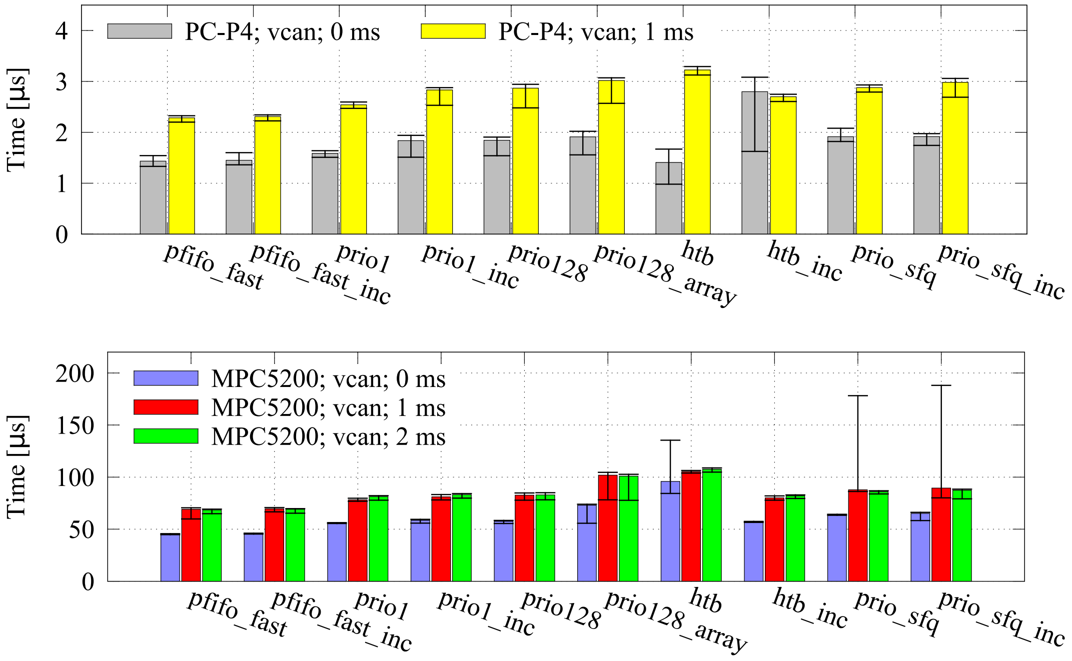

Figure 4.5.: Time spent in `can_send()` (minimum, 50th and 90th percentiles) with **virtual CAN interface**, depending on qdisc configuration, platform and traffic. CAN filter was compiled with SFF rules stored in a bitmap except from prio128_array experiment where the filtering rules were stored in an array.

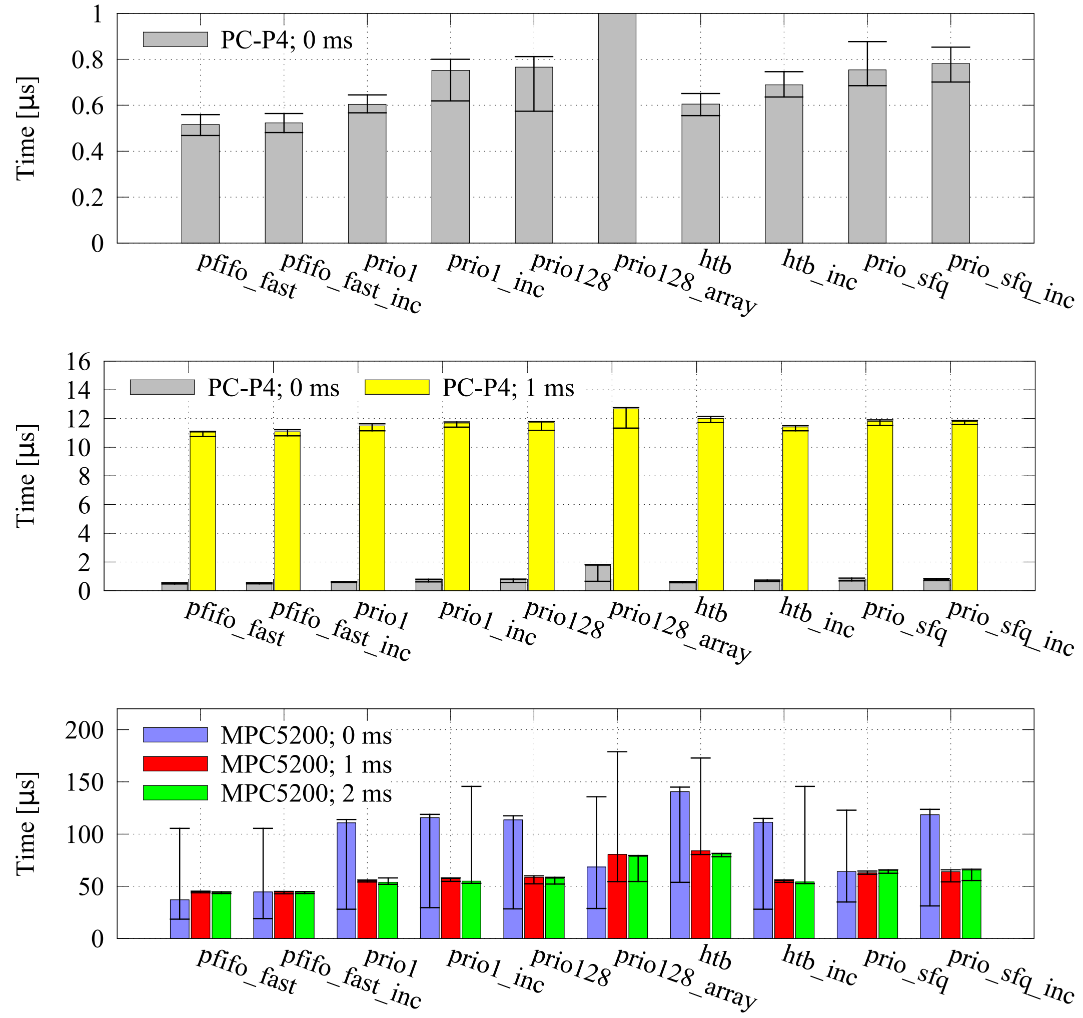

Figure 4.6.: Time spent in `can_send()` (minimum, 50th and 90th percentiles) with **real CAN interface** depending on qdisc configuration, platform and traffic. CAN filter was compiled with SFF rules stored in a bitmap except from prio128_array experiment where the filtering rules were stored in an array.

are higher, perhaps because the CPU caches were trashed by some other activities that run during the delay.

The bottom graph in Figure 4.5 shows the same experiments but this time run on MPC5200. The results are very similar to those measured on PC (except for that MPC5200 is much slower).

Figure 4.6 shows the performance on real CAN hardware. The top graph shows the PC with flood traffic and the middle one compares this to the case with 1 ms delay between frames. Now, the differences between those two cases is that in the first case (flood traffic) the queue is almost always full and the frames are only enqueued for processing later. In the case with the delay between frames, the queue is always empty and frames are enqueued, dequeued and sent to the driver. It can be seen that the driver part is dominant in this case and the difference between the qdiscs is negligible.

The fastest qdisc is again `pfifo_fast`, the slowest is `prio_sfq` and (surprisingly) `prio` alone. It can be also seen that the overhead of 128 filters stored in an array (in case of `prio128_array`) is significant, when compared to bitmap (see also Figure 4.7).

The situation on slower MPC5200 platform (bottom graph) is different. `pfifo_fast` is again the fastest, the slowest is `htb` when all frames are classified into a single class. Even `prio` with 128 filters stored in the array is faster than `htb`. On the other hand, when frames fall into more classes `htb` performs faster than `prio`. However, these differences are quite small and are probably caused by some cache effects rather than by the qdiscs algorithms.

What is interesting on MPC5200 platform, and we are not able to explain it, is that with flood traffic (left bars, 0 ms) the time spent in `can_send()` is most of the time greater than in case of throttled traffic. This is the opposite of what happens on PC. The minimum is smaller, but for the median of the measured times is greater.

A careful reader may also notice the big difference between 90$^{\text{th}}$ percentiles in 1 ms and 2 ms cases (e.g. for htb experiment). We suppose that this is due to some interference between hard- or soft-irqs and `can_send()`. For example, a TX IRQ from CAN controller could interrupt the `can_send()` of the next frame with high probability. It can be seen that by changing the period of sending frames (either from 1 to 2 ms or vice versa), this “anomaly” disappears.

4.2. Performance of can filter

In this experiment we compared the performance of two different implementations of `can` filter (see Section 3.2.6). Figure 4.7 shows how the time spent in `can_send()` depends on the number of filters configured in `can` filter. It was measured on MPC5200 platform with `prio` root qdisc. The kernel was the same as described in Section 4.1.1. As the array implementation of the filter can match up to 128 rules, in the experiment, we attached four filters to the qdisc. The rules were arranged in such a way, that only the last rule matched.

It can be seen, that the bitmap implementation has constant overhead irrespective of the number of filters. For a small number of rules (20 or so) the difference between both

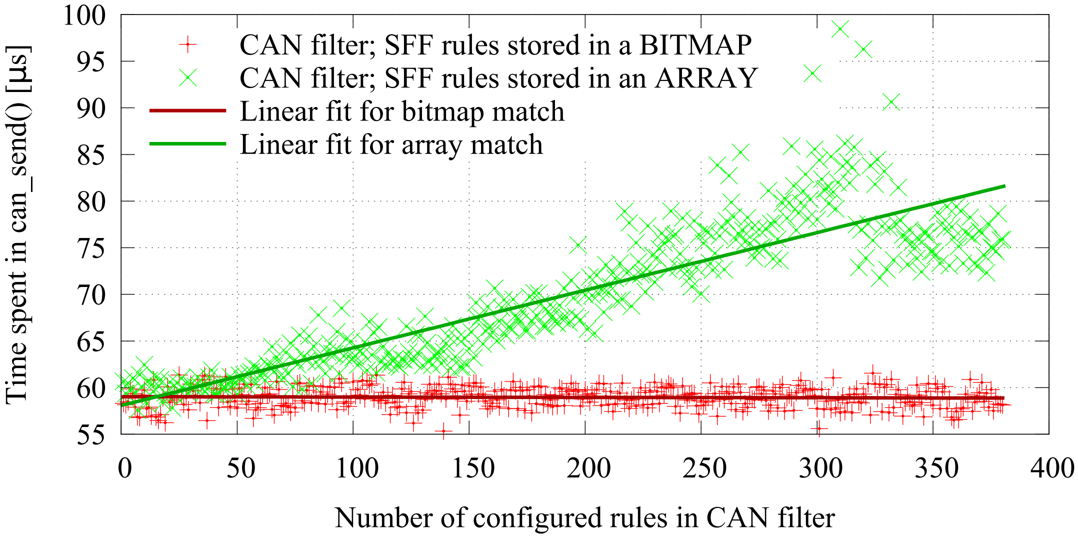

Figure 4.7.: Time spent in `can_send()` function (on MPC5200) depending on number of SFF rules in a filter (each filter has maximum count of 128 rules).

implementations in negligible.

4.3. Comparison of canid ematch and can filter

In this experiment we compared the performance of our former `can` filter (Section 3.2.6) and newer `canid` ematch (Section 3.2.3). The functionality of both is the same.

4.3.1. Test environment

These tests were conducted on the same MPC5200-based computer described in Section 4.1.1, but with Linux kernel 3.4.2.

CAN interface was configured with the following commands:

```
ip link set can0 type can bitrate 1000000
ip link set can0 txqueuelen 1000
```

CAN traffic was generated with the command:

```
cangen can0 -I $ID -L 8 -D i -g 1 -n 10000
```

where `$ID` sets CAN frame identifier. It was set to a specific fixed value so that the frame was classified either into the first or the last class.

4.3.2. Tested configurations

The configurations of different tests are summarized in Table 4.2. The tests differ mostly in filter configuration and `cangen` parameter `-I` (CAN frame ID).

Test	**Qdisc configuration**
default_qdisc	The default qdisc (pfifo_fast) automatically set when running `tc qdisc del dev $DEV root`.
prio_0	Prio qdisc with minimal configuration (2 bands – one for classification, one set as default).
prio_1first	Prio qdisc with 10 bands – for each class/band there is one filter (either `can` or `basic`) configured to match only one ID. All traffic is classified into the first class.
prio_1last	Same as above, except that all traffic is classified into the last class.
For em_canid only:	
prio_2first	Prio qdisc with 10 bands – each class/band has one `basic` classifier with 10 ematch rules (joined with OR) attached to it. All traffic is classified into the first class.
prio_2last	Same as above, except that all traffic is classified into the last class.

Table 4.2.: Configurations of the filter vs. ematch experiments.

The exact configuration of each conducted experiment would be to long to be published in this document. In Figure 4.8 is a fragment of configuration for *prio_1* experiments and in Figure 4.9 is a fragment of configuration of *prio_2* experiments.

```
    ${TC} filter add dev ${DEV} parent 1:0 basic match canid\( \
        $(printf "${PKT} 0x%x " $(seq -s " " 1 30)) \) \
        flowid 1:1
```

Figure 4.8.: Fragment of filter configuration of `prio_1*` experiment.

4.3.3. Results

The chart in Figure 4.10 shows the average execution times of `can_send()` function for the configurations from Table 4.2.

```
${TC} filter add dev ${DEV} parent 1:0 basic match \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 1  10)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 11 20)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 21 30)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 31 40)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 41 50)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 51 60)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 61 70)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 71 80)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 81 90)) \) OR \
    canid\( $(printf "${PKT} 0x%x " $(seq -s " " 91 100)) \) \
    flowid 1:1
```

Figure 4.9.: Fragment of filter configuration of prio_2* experiment.

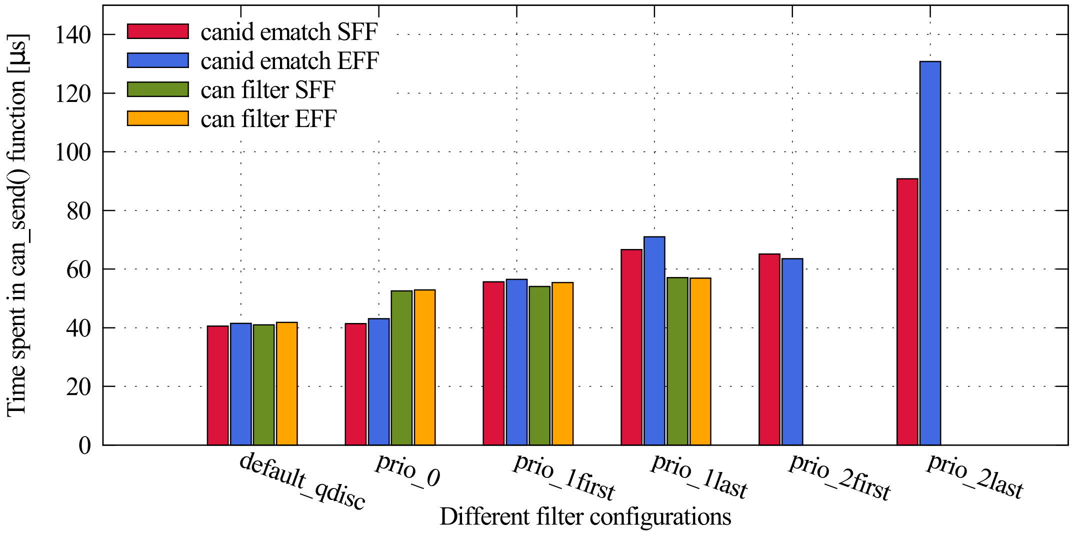

Figure 4.10.: Time spent in can_send() function (on MPC5200) depending on the number of configured filters.

5. Conclusion

In this document, we described how the Linux traffic control subsystem can be used for managing of CAN traffic. The goal was to solve priority inversion problems that happen with default SocketCAN configuration. This goal was successfully fulfilled. Almost all necessary building blocks of the solution are already included in the mainline Linux kernel. The only missing piece was a filter for easy classification of CAN frames. This filter was developed within this project and is in process of being merged into mainline.

We have also performed benchmarks of different qdiscs and the developed filters. All qdiscs are implemented quite efficiently and their overhead it typically no more than two times the overhead of the default qdisc `pfifo_fast`. With some hardware (PC, Kvaser PCI card), the overhead of the networking stack with all the qdiscs is negligible when compared to the overhead of the driver.

A. Example configurations of queueing disciplines

This section contains a set of working examples. Each of the examples consists of the exact command code used for creating the particular qdisc, an image showing the qdisc configuration and a short description.

A.1. Prio qdisc with multiple classes

Description

This example shows how to set `prio` qdiscs with more than 3 (the default) bands/classes. The number of classes of `prio` qdisc may be set only on creation – later modification is not possible. The qdisc has filters which are responsible for classifying CAN frames into different classes according their CAN IDs.

When CAN frame is enqueued, each filter (according to its priority) is tried until a match is found. The matching rule causes the frame to be enqueued into the specified in the rule.

When CAN frame is being dequeued out of the `prio` qdisc, first class is asked to perform dequeue. Only when there are no frames in the first class, second one is asked.

Achieved result

High priority traffic (classified into the first class) is always served in the first place.

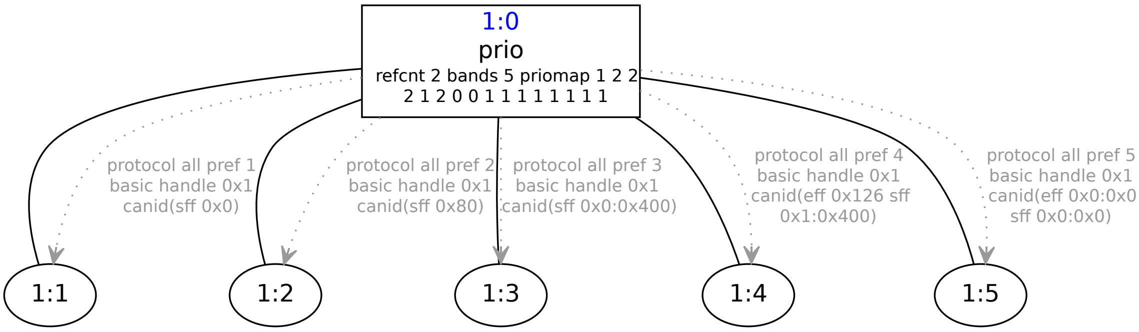

Figure A.1.: Prio qdisc with 5 classes.

```
# Create root qdisc
tc qdisc add dev ${DEV} root handle 1: prio bands 5

# Add filter to each class
tc filter add dev ${DEV} parent 1:0 prio 1 \
       basic match canid\(sff 0x0\) flowid 1:1 # CANopen NMT object
tc filter add dev ${DEV} parent 1:0 prio 2 \
       basic match canid\(sff 0x80\) flowid 1:2 # CANopen SYNC object
tc filter add dev ${DEV} parent 1:0 prio 3 \
       basic match canid\(sff 0x0:0x400\) flowid 1:3 #CANopen EMERG. -- PDO3(tx)
tc filter add dev ${DEV} parent 1:0 prio 4 \
       basic match canid\(sff 0x1:0x400 eff 0x126\) flowid 1:4
#CANopen PDO3(rx) -- NMT Error Control
tc filter add dev ${DEV} parent 1:0 prio 5 \
       basic match canid\(sff 0x0:0x0 eff 0x0:0x0\) flowid 1:5 # Default class
```

A.2. Prio qdisc with TBF child qdiscs

Description

This configuration of qdiscs (Figure A.2) uses prio qdisc with the default value of 3 classes as its main component. The behavior of the prio qdisc (described in A.1) may not be desirable because the lower priority classes would starve when there is traffic with higher priority. To avoid starvation the lower priority traffic/classes, it is possible to throttle the bandwidth of higher priority traffic. For this purpose TBF qdisc is used. This configuration uses TBF qdiscs for 2nd and 3rd class.

Achieved result

The 3rd class should not starve unless there is high priority (1st class) traffic.

```
tc qdisc add dev ${DEV} root handle 1: prio

tc filter add dev ${DEV} parent 1:0 prio 1 \
    basic match canid\(sff 0x80:0x780\) flowid 1:1 # CANopen EMERGENCY objects
tc filter add dev ${DEV} parent 1:0 prio 2 \
    basic match canid\(sff 0x0:0x400\) flowid 1:2 # CANopen PDO1(tx) -- PDO3(tx)
tc filter add dev ${DEV} parent 1:0 prio 3 \
    basic match canid\(sff 0x0:0x0 eff 0x0:0x0\) flowid 1:3 # default class

# TBF
tc qdisc add dev ${DEV} parent 1:2 handle 10: \
    tbf rate 0.1mbit burst 5kb latency 70ms
tc qdisc add dev ${DEV} parent 1:3 handle 11: \
    tbf rate 0.05mbit burst 5kb latency 70ms
```

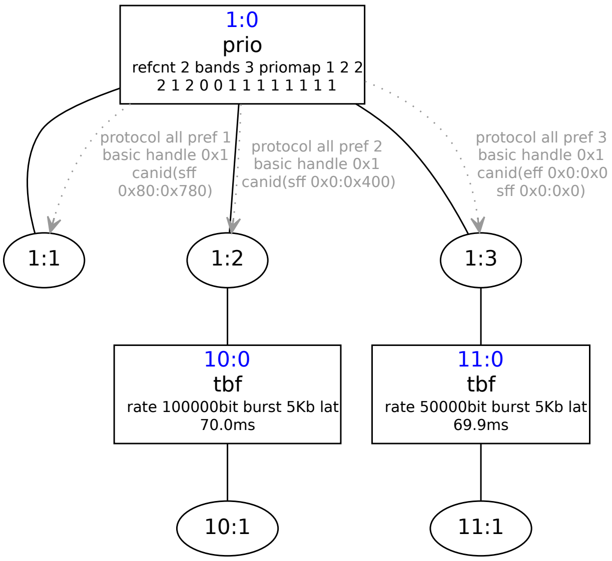

Figure A.2.: Prio qdisc with TBF child qdiscs.

A.3. Prio qdisc with SFQ child qdiscs

Description

When using `prio` qdisc, CAN frames are classified into different classes to ensure that higher priority frames reach the bus before lower priority ones. It is also useful to provide some kind of *fairness* for traffic within the same priority class – e.g. CAN frames from different application of middle priority will be dequeued in fair manner, so that each application will have the same chance to be dequeued first. This behavior can be achieved by using SFQ qdisc. The configuration is shown in Figures A.3 and **??**.

Achieved result

Frames of low priority class Its purpose is to *mix* frames belonging to one class so they will be dequeued in quasi-round-robin manner. SFQ is used to ensure dequeue out of one class in quasi-round-robin manner.

```
tc qdisc add dev ${DEV} root handle 1: prio
tc filter add dev ${DEV} parent 1:0 prio 1 \
   basic match canid\(sff 0x80:0x780\) flowid 1:1 # CANopen EMERGENCY objects
tc filter add dev ${DEV} parent 1:0 prio 2 \
   basic match canid\(sff 0x0:0x400 \
      sff 0x400:0x780\) flowid 1:2 # CANopen PDO1(tx) -- PDO3(rx)
tc filter add dev ${DEV} parent 1:0 prio 3 \
   basic match canid\(sff 0x0:0x0 eff 0x0:0x0\) flowid 1:3 # default class

# SFQ
```

```
tc qdisc add dev ${DEV} parent 1:2 handle 10: sfq perturb 10
tc qdisc add dev ${DEV} parent 1:3 handle 11: sfq perturb 10
```

A.4. Prio qdisc with TBF and SFQ child qdiscs

Description

This configuration shows the possibility of chaining classful qdiscs. To ensure fairness among frames which are in class with limited (throttled) bandwidth, one class consists of TBF qdiscs including SFQ qdisc.

Achieved result

Maximum throughput of the class is limited, moreover basic fairness is provided to frames being dequeued.

```
tc qdisc add dev ${DEV} root handle 1: prio

tc filter add dev ${DEV} parent 1:0 prio 1 \
   basic match canid\(sff 0x80:0x780\) flowid 1:1 # CANopen EMERGENCY objects
tc filter add dev ${DEV} parent 1:0 prio 2 \
   basic match canid\(sff 0x0:0x400\) flowid 1:2 # CANopen PDO1(tx) -- PDO3(tx)
tc filter add dev ${DEV} parent 1:0 prio 3 \
   basic match canid\(sff 0x0:0x0 eff 0x0:0x0\) flowid 1:3 # default class

# TBF
tc qdisc add dev ${DEV} parent 1:2 handle 10: \
   tbf rate 0.1mbit burst 5kb latency 70ms
tc qdisc add dev ${DEV} parent 1:3 handle 11: \
   tbf rate 0.1mbit burst 5kb latency 70ms

# SFQ
tc qdisc add dev ${DEV} parent 10:1 handle 101: sfq perturb 10
tc qdisc add dev ${DEV} parent 11:1 handle 102: sfq perturb 10
```

A.5. HTB qdisc

Description

HTB qdisc may represent a complex hierarchical structure of internal classes.

The main reason why to use the HTB qdisc is that each class has defined its guaranteed and maximal bandwidth. The class 1:1 in this example is used for sharing excess bandwidth among its children. It is also possible to attach another qdisc to HTB classes, e.g. SFQ qdisc as in this example.

It is possible to use the HTB qdisc hierarchy, e.g. the one shown in this example, as a qdisc attached to one class of a `prio` qdisc.

Achieved result

The overall CAN traffic from this interface is limited to ca. 100 kbit/s. This bandwidth is fairly distributed between three traffic classes with the ratio of 3/1/6. If some class does not fully use its share, the unused bandwidth from this class is distributed to the other two classes. Additionally, the bandwidth in not only fairly distributed between classes but the actual bandwidth of class 1:12 is also fairly distributed between all sockets that happen to send frames into this class. This is ensured by attaching SFQ to this class.

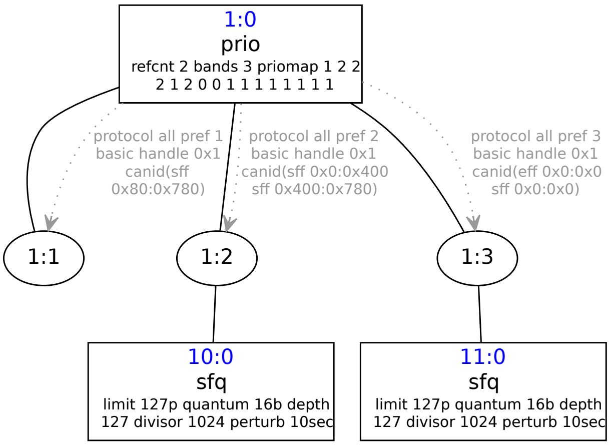
1:0
prio
refcnt 2 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
protocol all pref 1 basic handle 0x1 canid(sff 0x80:0x780)
protocol all pref 2 basic handle 0x1 canid(sff 0x0:0x400 sff 0x400:0x780)
protocol all pref 3 basic handle 0x1 canid(eff 0x0:0x0 sff 0x0:0x0)
1:1
1:2
1:3
10:0
sfq
limit 127p quantum 16b depth 127 divisor 1024 perturb 10sec
11:0
sfq
limit 127p quantum 16b depth 127 divisor 1024 perturb 10sec

Figure A.3.: Prio qdisc with SFQ child-qdiscs.

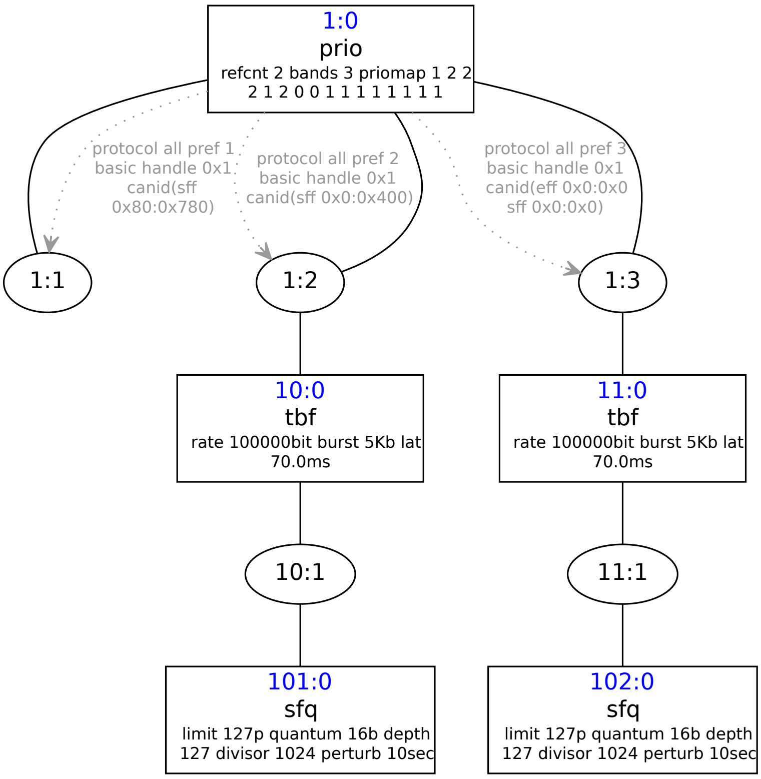
1:0
prio
refcnt 2 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
protocol all pref 1 basic handle 0x1 canid(sff 0x80:0x780)
protocol all pref 2 basic handle 0x1 canid(sff 0x0:0x400)
protocol all pref 3 basic handle 0x1 canid(eff 0x0:0x0 sff 0x0:0x0)
1:1
1:2
1:3
10:0
tbf
rate 100000bit burst 5Kb lat 70.0ms
11:0
tbf
rate 100000bit burst 5Kb lat 70.0ms
10:1
11:1
101:0
sfq
limit 127p quantum 16b depth 127 divisor 1024 perturb 10sec
102:0
sfq
limit 127p quantum 16b depth 127 divisor 1024 perturb 10sec

Figure A.4.: Prio qdisc with HTB child-qdiscs which have SFQ child-qdiscs.

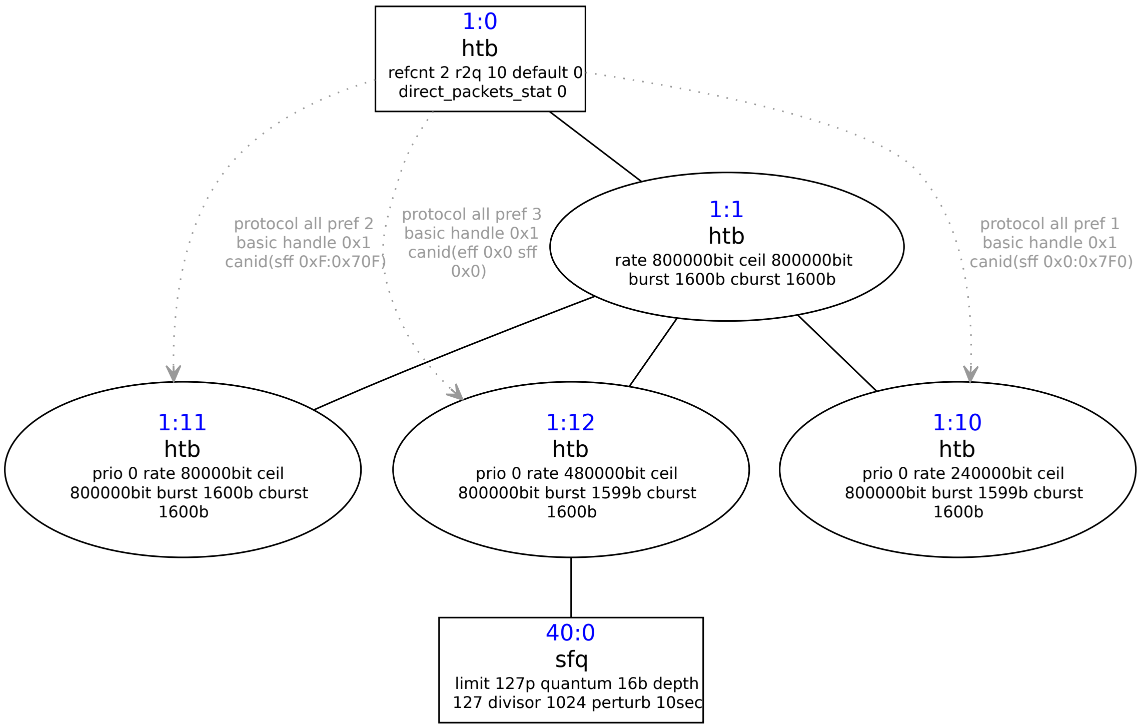
1:0
htb
refcnt 2 r2q 10 default 0
direct_packets_stat 0
protocol all pref 2
basic handle 0x1
canid(sff 0xF:0x70F)
protocol all pref 3
basic handle 0x1
canid(eff 0x0 sff
0x0)
1:1
htb
rate 800000bit ceil 800000bit
burst 1600b cburst 1600b
protocol all pref 1
basic handle 0x1
canid(sff 0x0:0x7F0)
1:11
htb
prio 0 rate 80000bit ceil
800000bit burst 1600b cburst
1600b
1:12
htb
prio 0 rate 480000bit ceil
800000bit burst 1599b cburst
1600b
1:10
htb
prio 0 rate 240000bit ceil
800000bit burst 1599b cburst
1600b
40:0
sfq
limit 127p quantum 16b depth
127 divisor 1024 perturb 10sec

Figure A.5.: HTB qdisc with its HTB classes.

```
tc qdisc add dev ${DEV} root handle 1: htb

tc class add dev ${DEV} parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev ${DEV} parent 1:1 classid 1:10 htb rate 30kbps ceil 100kbps
tc class add dev ${DEV} parent 1:1 classid 1:11 htb rate 10kbps ceil 100kbps
tc class add dev ${DEV} parent 1:1 classid 1:12 htb rate 60kbps ceil 100kbps

tc filter add dev ${DEV} parent 1:0 prio 1 \
    basic match canid\(sff 0x0:0x7f0\) flowid 1:10
tc filter add dev ${DEV} parent 1:0 prio 2 \
    basic match canid\(sff 0x0f:0x70f\) flowid 1:11
tc filter add dev ${DEV} parent 1:0 prio 3 \
    basic match canid\(sff 0x0 eff 0x0\) flowid 1:12

tc qdisc add dev ${DEV} parent 1:12 handle 40: sfq perturb 10
```

Figure A.6.: Commands used to configure hierarchical structure of HTB qdisc.

B. Queueing disciplines available in Linux TC not suitable for SocketCAN

This section shortly describes queueing disciplines available in Linux traffic control subsystem which were examined as a part of this work and they were found as not useful for purposes of SocketCAN.

SFB

Stochastic Fair Blue qdisc seems to behave similar to SFQ (i.e. provides *fairness* in dequeueing). The truth is it does not treat every flow the same way – it *penalizes inelastic* (i.e. large traffic) *flows*.

Multiq

This queueing discipline dequeues packets in round-robin fashion. It apparently works only for Intel NICs with hardware support of multiple queues.

mq

This queueing discipline dequeues packets in round-robin fashion. The main disadvantage is that it is available only for NICs with more than 1 TX queue.

drr

Deficit Round-Robin Scheduler may provide *fairness* when enqueueing packets from multiple flows. It is classful although all classes have to be of type “drr”.

CBQ

Class Based Queueing has similar usage as HTB qdisc. Its main disadvantage is that it is too complex (everybody discourages from using it). It seems there is no reason for using CBQ instead of HTB.

Bibliography

[1] W. Almesberger, “Linux network traffic control – implementation overview,” in *Proceedings of 5th Annual Linux Expo, Raleigh, NC*, May 1999, pp. 153–164. [Online]. Available: http://www.almesberger.net/cv/papers/tcio8.pdf

[2] B. Hubert *et al.*, “Linux advanced routing & traffic control howto.” [Online]. Available: http://lartc.org/howto/