Performance evaluation of Linux CAN-related system calls

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

October 29, 2014
Version 1.0

Contents

Document history

Version	Date	Description
0	2014-01-14	First draft
1	2014-10-29	Final version

Abstract

Linux kernel contains a full featured CAN bus networking subsystem. It can be accessed from applications via several different interfaces. This paper compares the performance of those interfaces and tries to answer the question, which interface is most suitable for capturing traffic from a big number of CAN buses. Motivation for this work is the development of various CAN traffic analyzers and intrusion detection systems. Besides traditional UNIX interfaces we also investigate the applicability of recently introduced "low-latency sockets" to the Linux CAN subsystem. Although the overhead of Linux in general is quite large, some interfaces offer significantly better performance than others.

1 Introduction

Controller Area Network (CAN) is still the most widespread automotive networking standard today, even in the most recent vehicle designs. Although there are more modern solutions available on the market [1, 2], CAN represents reliable, cheap and proven solution, making it the preferred choice in the industry. Although the newer technologies such as FlexRay or Ethernet are used more and more, it seems unlikely that CAN is going to be phased out in foreseeable future.

Linux kernel contains a full featured CAN bus networking subsystem. Together with `can-utils` package, it represents versatile and user friendly way for working with CAN networks. Linux CAN subsystem is based on Linux networking subsystem and shares a lot of code with it. This is a source of significant overhead [3], because the networking subsystem is tuned for high throughput Ethernet networks with big packets and not for up to eight byte long CAN frames. For example, one `sk_buff` structure that represents one CAN frame in the Linux kernel has size 448 bytes on our system. Nevertheless, due to user friendliness and widespread availability of Linux, its CAN subsystem is used a lot.

Recently, after publishing information about CAN bus-related attacks on automotive electronic control units [4, 5], automobile manufacturers started taking cybernetic security more seriously. Not only that people started proposing message authentication techniques for CAN bus [6, 7, 8], but various other techniques to increase automobile security are discussed. One possibility is to use intrusion detection systems (IDS) that analyze CAN bus traffic and try to detect anomalies. Before developing an IDS one needs a way to efficiently receive CAN frames from the whole car so that IDS can process them. It is natural to use Linux for prototyping the IDS and the goal of this paper is to evaluate which Linux system calls have the lowest overhead and are thus suitable for logging of big number of CAN interfaces.

The contribution of this paper is manifold. First, we present comprehensive performance evaluation of multiple Linux system calls that allow an application to interact with CAN networks. Second, we investigate the applicability of recently introduced "low-latency sockets" to the Linux CAN subsystem. Third, we compare the overhead of the whole Linux CAN subsystem with the raw hardware performance represented by lightweight RTEMS executive.

This paper is structured as follows. Section 2 provides a brief introduction to the internals of the Linux networking stack. We describe our methodology for system call evaluation in Section 3, followed by results and discussion in Section 4. The paper is concluded in Section 5.

2 Linux networking stack

The architecture of the Linux networking stack is depicted in Figure 2.1. We describe the main components of the stack, which are necessary for understanding the remainder of this paper.

First, we describe the transmission path (green arrows on the left). Application initiates transmission process by invoking `send()` or `write()` system call on a socket. Both calls end up in the socket layer. In case of CAN raw sockets this is `raw_sendmsg()` function in can/raw.c. This function allocates an `sk_buff` structure, copies the CAN frame to it and passes the result to the lower layer. Protocol layer (af_can.c) immediately forwards the received `sk_buff` to the queuing discipline. If the device driver can accept a CAN frame for transmission, queueing discipline just passes the frame to the driver, which instructs the CAN controller (hardware) to transmit the frame. Then, `send()` returns. If, on the other hand, the device is busy, queuing discipline stores the frame and `send()` returns at that time. Later, when the device is ready, the devices driver asks the queueing discipline to supply a frame for transmission.

The reception path is shown with red arrows on the right. When a CAN frame is received by the CAN controller, it signals to the CPU an interrupt request (IRQ), which is handled by the device driver in the interrupt service routine (ISR). The ISR runs in the so called hard-IRQ context of the Linux kernel. It checks the reason of the interrupt and in case of receiving a new frame it asks Linux to schedule a so called NETRX soft-IRQ at some time later. In normal situation, soft-IRQ is executed just after exiting the ISR. It calls device driver's `poll()` function, which reads the CAN frames out of the hardware, stores them into `sk_buff`s and passes them to the protocol layer. Protocol layer clones the received `sk_buf` in order to deliver it to every socket interested in receiving this

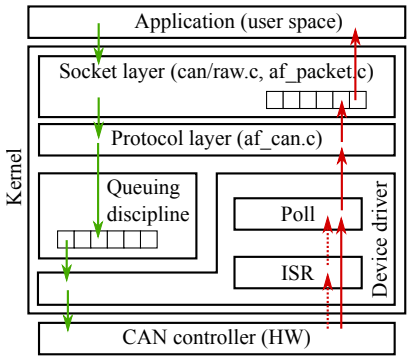

Figure 2.1: Linux networking stack architecture. Solid green arrows represent transmit data flow, red ones receive data flow. Dotted arrows represent notifications.

frame. Then, the cloned `sk_buff` is passed to the socket layer for storing the frame in the socket queue. When an application invokes `read()` or `recv()` system calls, it receives the frame from the socket queue.

3 Methodology

We developed two methods for system call performance evaluation. One serves mainly for comparison of system call overhead and does not require any special hardware setup. The second method takes into account all components involved in communication such as device drivers, protocol layer and system calls, but requires complex hardware setup.

3.1 Virtual CAN interface method

This method can be used for performance comparison of `read()` system call with `recvmmsg()` and of `write()` with `sendmmsg()`. The difference between those system calls is that `read()` and `write()` operate on a single message (CAN frame) whereas their `mmsg` counterparts can operate on multiple messages (hence the name). This method requires the use of virtual CAN interface (`vcan`). Therefore, it evaluates only the overhead of the system calls and the socket and protocol layers. It does not take into account the overhead of device drivers and queuing disciplines.

The method works as depicted in Figure 3.1. Two raw CAN sockets are created, one for transmission and one for reception. Both are bound to the same virtual CAN interface. For the reception socket we set the maximum size of the reception socket queue so that it can hold all frames that we plan to send. This is accomplished by calling `setsockopt()` with `SO_RCVBUFFORCE` parameter. Then we send the planned number of CAN frames to the transmission socket and measure the time of this operation. Every send operation involves copying the CAN frame from the application to the kernel, creating `sk_buff` structure representing the sent frame and storing it in the reception queue of the receiving socket (green arrow in the figure). The latter step is performed by the `af_can.c` when it detects that the socket is bound the virtual CAN interface. Once all frames are sent, we start reading them out of the reception socket. We measure how long does the reading take (red arrow).

The source code of the application implementing this method is available in our repos-

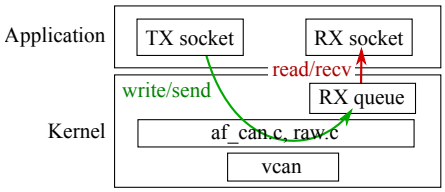

Figure 3.1: Virtual CAN method.

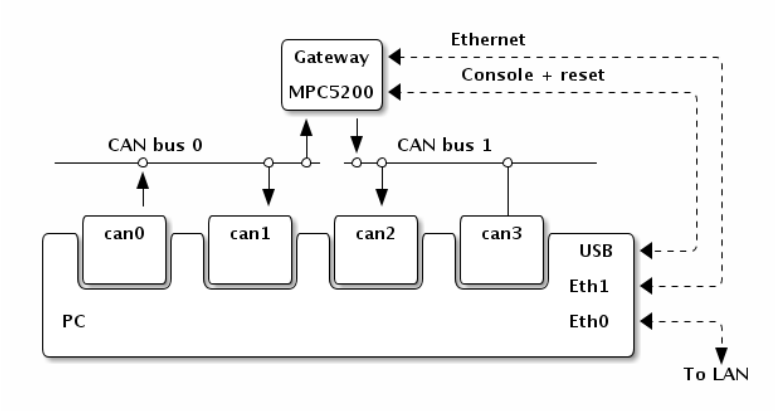

Figure 3.2: Testbed configuration.

itory[1].

3.2 Gateway-based method

This method evaluates the CAN system call performance by creating a Linux-based CAN gateway and measuring how long does it take for the gateway to route a frame from one CAN bus to another. We use the same hardware for the gateway but change the software that implements it. This method evaluates the performance of all network stack layers i.e. device driver, queuing discipline, protocol layer and sockets.

3.2.1 Test bed

The test bed used for this method is depicted in Figure 3.2. There is a PC with Kvaser PCI quad-CAN SJA1000-based card and the CAN gateway running on a MPC5200 (PowerPC) system. The PC and the gateway (GW) are connected via a dedicated Ethernet network ("crossed" cable) which is used only for booting the GW over network. This is not used when running the experiments. The gateway is implemented either in Linux or in RTEMS. In case of Linux, only the gateway application and the Linux kernel are running in the system. No other user processes are run.

The PC is an old Pentium 4 box running at 2.4 GHz with hyper-threading, 2 GB RAM. The second Ethernet interface is connected to the local LAN. This interface is also disabled while running the experiments. The application that runs on the PC generates the test traffic and measure the gateway latency (as detailed below). It has assigned real-time priority (SCHED_FIFO) and all its memory is locked i.e. it cannot be swapped out to the disk.

[1] `https://rtime.felk.cvut.cz/gitweb/can-benchmark.git/blob/HEAD:/recvmmsg/can_recvmmsg.c`

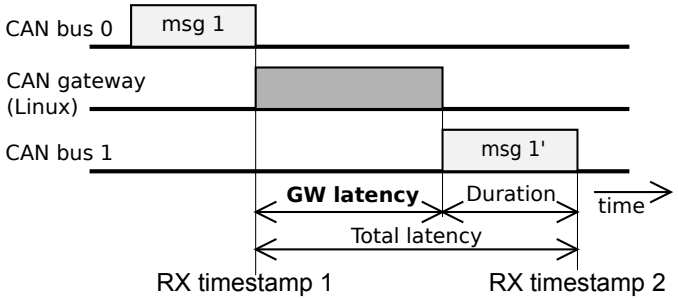

Figure 3.3: Definition of latency.

3.2.2 Gateway latency measurements

The PC sends frames from the `can0` interface and receives them on the `can1` interface as well as on `can2`, after passing through the gateway. We measure the latency of the frame, which is the time interval between receptions of the frame on `can1` and `can2`. The time of frame reception is determined from the timestamps taken by the kernel at interrupt time and is accessed from the user-space via `SO_TIMESTAMPNS` socket option. Figure 3.3 shows the measured latency in the time chart. The final latency of the gateway is calculated as the difference of the two timestamps decreased by the duration (including bit stuffing) of the frame on the bus.

Since the frames are generated and received in the same computer, the TX and RX timestamps are measured with the same clock (TSC/HPET) and thus the measured latencies are very accurate. We evaluated the accuracy of this method in [9] by comparing our results with an independent measurement with a CAN analyzer. The worst case error of our method was 10 μs and this error occurred in less than 0.1% of measurements. Since then, we improved the accuracy even more by disabling all unrelated Ethernet traffic during experiments.

Whenever a frame is received on interface `can2`, it is necessary to find the other timestamp associated with that frame in order to calculate the frame latency. This procedure must be immune to frame losses and changes in the reception order. Therefore we use the first two bytes of every frame to store a unique number which allows the frame to be tracked.

3.2.3 Traffic generator

In our experiments, we generated the traffic by periodically sending CAN frames. The period differed depending on the experiment, but in general it was between 120 and 500 μs. We used the frames in standard (SFF) format with two bytes of data and the CAN bus bit rate was 1 Mbps. The average length of such frames is 65 bits, which corresponds to 65 μs of transmission time (i.e. "duration" in Fig. 3.3).

3.2.4 Gateway implementations

We used several implementations of the gateway in order to compare various Linux system calls. In addition to the user space gateway, we compare our results with the gateway implemented in the kernel space and with a non-Linux gateway implemented in RTEMS executive. All implementations are described bellow and summarized in Table 3.1.

rtems This gateway is not based on Linux. Instead, it uses RTEMS [10] executive, which is more lightweight than Linux. We believe that the performance of this gateway is very close to raw hardware performance. We include it in our experiments to see the overhead of Linux in general. The sources of the gateway are available in our repository[2].

kernel The gateway that is a part of the Linux kernel[3] configured with the following command:

```
cangw -A -s can0 -d can1
```

read-write The simplest possible user space gateway. It uses `read()` system call to receive CAN frames and `write()` system call to forward them to another CAN interface.

readnb-write The same as *read-write*, but with the `O_NONBLOCK` flag set for the receiving socket. This flag causes the socket to be *non-blocking*, which means that whenever there is no frame in the socket queue, the call returns immediately with an error instead of blocking the caller until a frame is received. This means that the application busy-waits in a loop around the `read()` in the user space.

readbusy-write The same as *read-write*, but with `SO_BUSY_POLL` (a.k.a. low-latency sockets) option set for the receiving socket. This option was added to the Linux kernel recently (in version 3.11) and its effect is similar to the *readnb-write*, except that the busy-waiting happens transparently in the kernel. In order for this to work, it was necessary to patch the Linux kernel. For more details see the discussion in Section 4.3. The timeout for polling was set to $300\,\mu s$.

mmap-mmap This gateway uses `PF_PACKET` sockets which store or read packets (CAN frames) to/from a ring buffer memory that is shared between the kernel and the user space (via `mmap()` system call). With these sockets, extra copying of the frames between the kernel and the application is avoided and the application can receive or send multiple packets in one system call or, in case of reception, without calling the system at all.

Whenever the kernel receives a frame, it stores its content together with some metadata in the ring buffer and the user space application can immediately access it without any system call. If there is no frame available in the ring buffer,

[2] `https://rtime.felk.cvut.cz/gitweb/can-benchmark.git/tree/HEAD:/rtems/gw/cangw`
[3] `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/net/can/gw.c`

Implementation	Environment	Receive side				Send side	
		Socket type	**System call**	**Busy wait-ing**	**Notes**	**Socket type**	**System call**
rtems	RTEMS	—	—	No		—	—
kernel	Kernel space	—	—	No		—	—
read-write	User space	AF_CAN	read()	No		AF_CAN	write()
readnb-write	User space	AF_CAN	read()	Yes	nb	AF_CAN	write()
readbusy-write	User space	AF_CAN	read()	Yes	bp	AF_CAN	write()
mmap-mmap	User space	AF_PACKET	poll()	No		AF_PACKET	send()
mmap-write	User space	AF_PACKET	poll()	No		AF_CAN	write()
mmapbusy-mmap	User space	AF_PACKET	—	Yes		AF_PACKET	send()
mmapbusy-write	User space	AF_PACKET	—	Yes		AF_CAN	write()
readnb-mmap	User space	AF_CAN	read()	Yes	nb	AF_PACKET	send()
readbusynoirq-write	User space	AF_CAN	read()	Yes	bp, i	AF_CAN	write()
mmsg-mmsg	User space	AF_CAN	recvmmsg()	No		AF_CAN	sendmmsg()

Table 3.1: Summary of gateway implementations. Notes: *nb* – socket with `O_NONBLOCK` option enabled, *bp* – socket with `SO_BUSY_POLL` option set, *i* – intrusive changes to devices drivers and network stack needed.

the application can call the `poll()` system call to wait for some frame to arrive. Transmission happens similarly. The application writes one or more frames into the ring buffer and calls `send()` on the socket. The kernel then sends all the frames at once. Details of this type of socket are provided in [11].

Our *mmap-mmap* gateway works as follows. It reads frames from the reception ring buffer and copies them to the transmission ring buffer. Once the reception ring buffer is empty or when there is no space in the transmission ring buffer, the `send()` system call is called and the process starts anew. Then, if the reception ring buffer is still empty, we call `poll()` to wait for a new frame.

mmap-write The same as *mmap-mmap*, but frame transmission is carried out with the `write()` system call on a raw CAN socket instead of the `PF_PACKET` socket.

mmapbusy-mmap The same as *mmap-mmap*, but when the reception ring-buffer is empty, it does not call `poll()` but busy waits in a loop until a frame appears in the ring buffer.

mmapbusy-write Combination of *mmapbusy-mmap* and *mmap-write*.

readnb-mmap Non-blocking read for reception together with the `mmap()` `PF_PACKET` socket for transmission. `send()` is called only when `read()` returns that there are no more frames.

readbusynoirq-write The same as *readbusy-write* but the kernel was patched to disable receive interrupts during busy waiting. See Section 4.3 for details.

mmsg-mmsg This gateway use a combination of `recvmmsg()` and `sendmmsg()` to perform its job. Similarly to *mmap-mmap*, it can handle multiple CAN frames with one system call.

All Linux-based gateways used kernel 3.12.3 patched with board support patches for our MPC5200-based board[4]. The kernel configuration is available in our repository[5]. The source code for all above mentioned user space gateways is also in our repository[6]. It is a single program and different variants of the gateway are selected with command line switches.

[4] `https://rtime.felk.cvut.cz/gitweb/shark/linux.git/shortlog/refs/heads/shark/3.12.x`

[5] `https://rtime.felk.cvut.cz/gitweb/can-benchmark.git/blob/HEAD:/kernel/build/shark/3.12/.config`

[6] `https://rtime.felk.cvut.cz/gitweb/can-benchmark.git/blob/HEAD:/ugw/ugw.c`

4 Results

In this section we present and discuss the results obtained from the experiments described above.

4.1 Advantages of *mmsg() system calls

We applied the method described in Section 3.1 to a varying number of frames and plotted the measured times to the graph in Figure 4.1.

The left graph shows the performance of a PC, the right graph of the MPC5200-based (embedded) system. On the PC, reading 50 thousand of frames with `read()` takes 15 ms, whereas using a single `recvmmsg()` call takes about 14 ms. Sending the same number of frames with `write()` takes 40 ms and with `sendmmsg()` 35 ms. For all system calls the time depends almost linearly on the number of frames. The `recvmmsg()` system call is 7% faster than `read()`, `sendmmsg()` is 12% faster than `write()`.

On MPC5200, the situation is similar but the difference is bigger. `recvmmsg()` is 19% faster and `sendmmsg()` even 35% faster.

Note that this method measures only a fraction of full RX and TX paths and especially that the measured RX latency comprises only the very last step of the full RX path. See Figures 2.1 and 3.1 to compare the difference between measured and full paths.

Furthermore, we looked whether the `*mmsg()` system calls can have higher overhead than `read()`/`write()` for a small number of frames. The result is that `*mmsg()` system calls are always slightly faster even for a single frame.

4.2 Comparison of gateway implementations

We measured the latency of the gateway as described in Section 3.2. In each experiment we sent 3 200 frames. The measured latency represents the time needed by the gateway for reception and transmission of a single frame. In the graphs below, we show medians of the measured latencies.

The results for low CAN traffic, i.e. a new frame was only sent to the gateway after the previous one was received from it, are depicted in Figure 4.2.

The lowest latency of 15 μs is achieved with the RTEMS-based gateway. The second lowest value (49 μs) belongs to the kernel-based gateway. These two values are provided only for comparison and the properties of these gateways are outside the scope of this paper.

Classical *read-write* gateway performs rather badly with its 180 μs, similarly as all implementations that use some form of blocking, i.e. *mmap-write*, *mmap-mmap* and

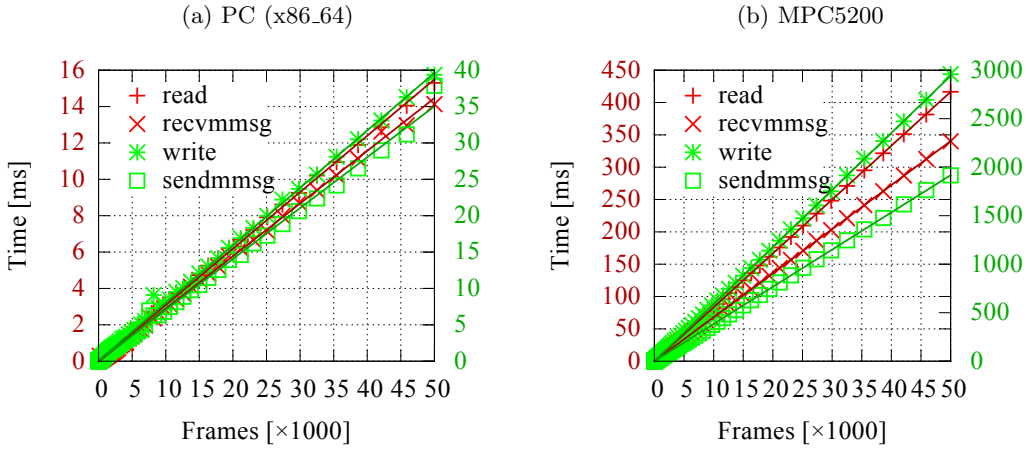

Figure 4.1: Comparison of `read()`/`write()` and `recvmmsg()`/`sendmmsg()` performance on a virtual CAN (`vcan`) interface. The straight lines show linear fit of the measured data. Note that reception times correspond to the left axis and send times to the right one.

mmsg-mmsg. A slightly better latency around 100 μs can be achieved if some form of busy waiting is employed.

The simplest form of busy waiting is to use non-blocking sockets as in the *readnb-write* implementation. The increased performance is caused by the fact that the application thread never blocks and when a new frame arrives, there is no overhead of switching the context from the idle thread back to the application and perhaps also for waking the CPU from some power saving mode.

Other busy waiting implementations perform similarly. The *readbusy-write* implementation (a.k.a. low-latency sockets) is discussed separately in Section 4.3.

It was determined that having `ftrace` compiled in the kernel (but disabled at runtime) caused about 20% performance degradation. Therefore, `ftrace` was not compiled into the kernels used in this paper.

The situation is different for heavier traffic (see Figure 4.3) when frames were sent every 120 μs. All `write()`-based implementations have very long latencies (caused by queueing delays in overflowed socket queues) and also loose up to one third of frames. Implementations that use `mmap()` or `sendmmsg()` perform much better and when `mmsg`- or `mmap`-based operations are used for both reception and transmission no packet is lost. Surprisingly, no frame is lost even for *readnb-mmap* gateway.

The reason for good performance of `mmsg`- or `mmap`-based operations is that it is not required to perform a system call for every frame. Receiving side can be implemented completely without system calls as in *mmapbusy-** implementations. Transmission side requires a system call, but one `send*()` can (and in our implementations does) transmit several frames.

To evaluate how various system calls scale to higher bandwidth, we measured how

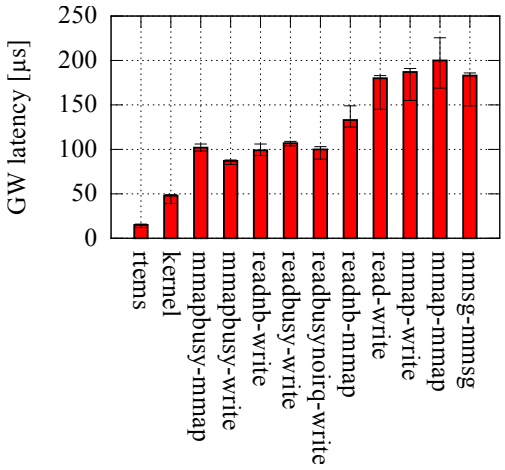

Figure 4.2: Latencies of various gateway implementations on MPC5200. A new frame was sent to the gateway only after receiving the previous one from the gateway. Bars show median, error bars span from the minimum value to 90$^{\text{th}}$ percentile.

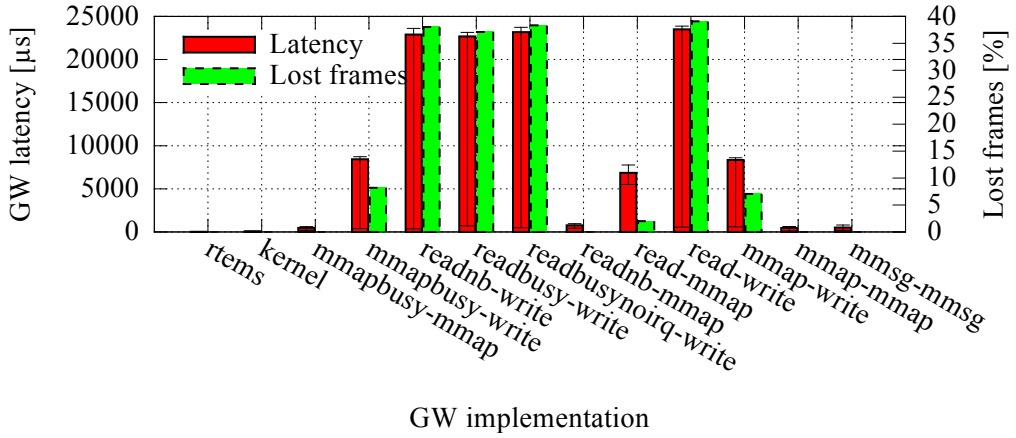

Figure 4.3: Latencies of various gateway implementations on MPC5200 when frames are sent with the period of 120 μs. Bars show median, error bars span from the minimum value to 90$^{\text{th}}$ percentile. The exact height of small bars can be seen at the bottom of Figure 4.4. The high latencies (e.g. 23 ms) are caused by queueing delays when the gateway was overloaded.

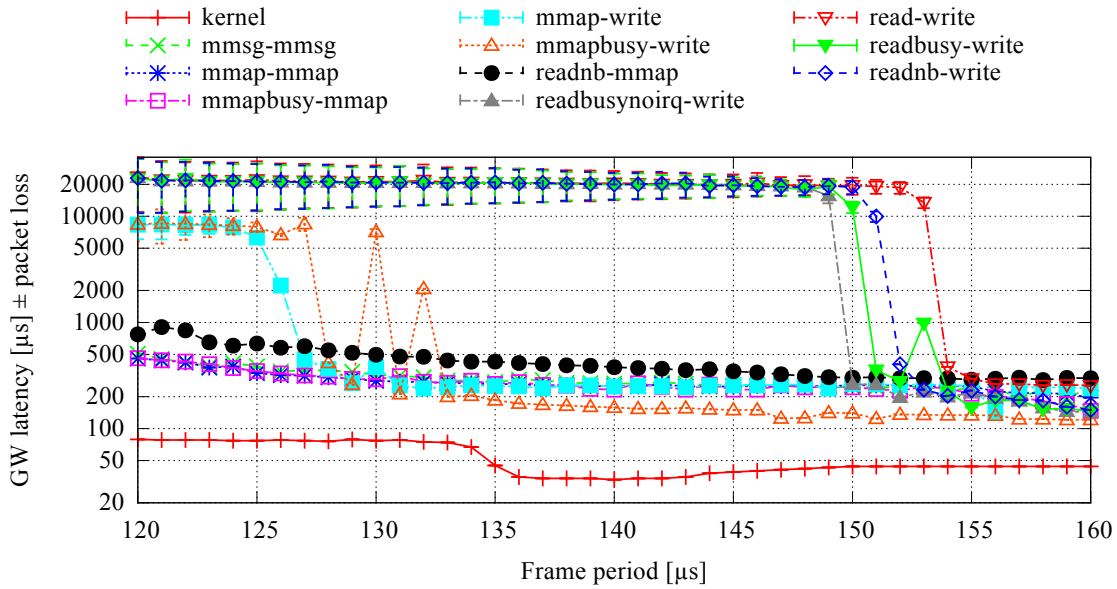

Figure 4.4: Latency and packet loss depending on the frame rate. Measured on MPC5200. The latency is the median of 3 200 measurements. Packet loss is drawn as error bars. For the sake of readability, error bars are magnified ten times.

does the gateway latencies and lost frames depend on the frame rate. The result is in Figure 4.4.

As it was already discussed, *read*-write* implementations cannot sustain higher frame rates (lower frame periods). The gateway throughput is saturated for periods between 150 and 155 μs. Only kernel-based gateway and `mmap`/`mmsg`-based gateways can sustain the higher frame rates.

Both *mmap*-write* gateways drop off around 127 μs. By comparing *mmap-write* with *readnb-mmap*, one can conclude that `write()` has a bigger overhead than `read()`. This matches with the results in Section 4.1, where the write is 3 to 6 times longer than read.

Gateways *mmap*-mmap* and *mmsg-mmsg* perform almost exactly the same. They all benefit from the fact that they can handle multiple frames per system call. Interestingly *readnb-mmap*, survives the highest frame rate[1] even though it needs to use one system call for each received frame. Probably frame coalescing on transmission side is sufficient here.

Although the kernel gateway is not evaluated in this paper, it is interesting to see the increase in latencies around period of 135 μs. This is caused by a TX interrupt on the output CAN interface (see RX timestamp 2 in Figure 3.3) that happens to interrupt the processing of the next received frame on the input CAN interface.

One may also wonder why the *mmapbusy-write* curve jumps up and down around

[1]Our test bed cannot generate higher frame rates reliably.

130 μs. We do not have the exact explanation, but in general, close to the point where the gateway throughput get saturated the measurements are quite unstable. The reason is that at this point the main factor that determines the measured latency changes from processing delay to queuing delay. We tried to find a high enough number of measurements to average out this instability, but we have to make a compromise between the number of measurements and the duration of the experiment.

4.3 Busy polling of CAN sockets

Busy polling, previously known as low-latency sockets, is a feature that was added to Linux recently, in version 3.11. It was designed for latency sensitive applications communicating over Ethernet. The question that we try to answer in this section is how beneficial is this feature for CAN bus networking. It is shown, that for most CAN applications, there are no big advantages of using this feature.

When busy polling is enabled for a socket (e.g. with `setsockopt(SO_BUSY_POLL)`) then the behavior of `read()` or `poll()` system calls is changed if the socket receive queue is empty. Instead of blocking the caller, the device driver is invoked directly from the application context to poll for new packets (without busy polling enabled, such polling is only invoked from hard or soft IRQ context). This happens in a busy loop until either a packet is received or a specified timeout elapses. When a new packet arrives, it is pushed to the network stack (protocol layer in Figure 2.1) as in the standard situation and the application finds new data in the socket queue when leaving the busy loop. Device independent functionality of busy poll is implemented in `include/net/busy_poll.h`[2].

For CAN networking, busy polling does not work out of the box. It is necessary to add support for it to the network stack as well as to device drivers. We implemented support for CAN raw sockets[3] and for the `mscan` device driver[4].

As can be seen from Figure 4.2, the performance of *readbusy-write* implementation is not very different from other non-blocking implementations, namely from *readnb-write*, which also uses busy polling but in the user space instead of in the kernel. The reason is that when the application busy-polls for the frames in the `read()` call, receive IRQs are not disabled and the system still needs to handle an IRQ when a frame arrives. Soft-IRQs are disabled when busy-polling but since every MSCAN interrupt service routine (ISR) schedules a soft-IRQ, the soft-IRQ is always invoked after the busy-polling exits and a frame was received. Usually, this soft-IRQ has no work to do because all work was already performed during the busy-polling. The difference in the sequence of operations during blocking and busy-polling read is depicted in Figure 4.5 a) and b).

Since it is not necessary to handle receive interrupts while polling, we were interested in whether eliminating the interrupt overhead brings significant advantage. We imple-

[2] `http://lxr.free-electrons.com/source/include/net/busy_poll.h?v=3.12`

[3] `https://rtime.felk.cvut.cz/gitweb/shark/linux.git/commitdiff/154a6ad59e506835b6b812a3128f16e5786291e5?hp=f6d51c347e2b3568061990600062240 1bf5f6cc5`

[4] `https://rtime.felk.cvut.cz/gitweb/shark/linux.git/commitdiff/d40550253b8ea0434100b496528b35759eff07e5?hp=154a6ad59e506835b6b812a3128f16e5786291e5`

```
read() {
  block()
  :
  hard IRQ {}
  soft IRQ {
    poll_for_frames() {
      push_to_stack() {
        wake_up()
      }
    }
  }
  copy_to_user()
}
```

(a) Blocking read

```
read() {
  poll_for_frames()
  poll_for_frames()
  poll_for_frames()
  hard IRQ {}
  poll_for_frames() {
    push_to_stack()
  }
  soft IRQ {
    poll_for_frames()
  }
  copy_to_user()
}
```

(b) Read with busy-poll

```
read() {
  irq_disable()
  poll_for_frames()
  poll_for_frames()
  poll_for_frames()
  poll_for_frames() {
    push_to_stack()
  }
  irq_enable()
  copy_to_user()
}
```

(c) Read with busy-poll, no IRQ

Figure 4.5: Comparison of blocking and busy-poll reception path.

mented a patch[5] that disables hard-IRQs during polling. The results on the gateway with this patch applied are named *readbusynoirq-write*. Detailed examination of Figure 4.2 reveals that the latency is 7 μs smaller when compared to *readbusy-write*, which is not a significant improvement. Also in Figure 4.4, although the *readbusynoirq-write* line drops off at the highest frame rate (150 μs) among all *read*-write* implementations, the difference is not significant.

The reason why busy-polling sockets are useful for Ethernet and not for CAN (or at least for the MSCAN device) is that Ethernet devices implement so called interrupt coalescing. In this mode, the device does not interrupt the host with every received packet but enforces minimum inter-IRQ time. For example Intel's `ixgbe` driver supports 125, 50 and 10 μs as minimum inter-IRQ time. When a packet arrives at the beginning of the no-IRQ interval, the application might end up waiting for it until the end of the interval. With busy-polling sockets, this extra latency can be eliminated at the price of higher CPU utilization.

[5] `https://rtime.felk.cvut.cz/gitweb/shark/linux.git/commitdiff/busy_poll_noirq`

5 Conclusion

From our results, it can be clearly seen that kernel interfaces that can handle multiple CAN frames per invocation are superior to the simple `read()`/`write()` interface. Those interfaces are PF_PACKET sockets and `recvmmsg()`/`sendmmsg()` and both perform almost the same. The shiny new "low-latency sockets" are not very interesting for CAN bus. CAN traffic processing in the Linux kernel is about 3 times slower than in RTEMS and when the processing is done in the Linux user space, it can be even 13 times slower.

As for the future work, it might be interesting to propose a new interface that would be the combination of SO_BUSY_POLL (i.e. polling the driver directly from the task context) and memory mapped sockets. With such an interface, it might be possible to bypass the Linux networking stack completely and achieve very good receive performance. Such interface would be suitable for logging of big number of CAN interfaces even on low-end hardware.

Acknowledgment This work was financially supported by Volkswagen AG. The authors would like to thank Oliver Hartkopp for his support.

Bibliography

[1] T. Nolte, H. Hansson, and L. L. Bello, "Automotive communications-past, current and future," in *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, 2005, pp. 992–1000.

[2] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93(6), pp. 1204–1223, 2005.

[3] M. Sojka, P. Píša, M. Petera, O. Špinka, and Z. Hanzálek, "A comparison of Linux CAN drivers and their applications," in *5th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2010.

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028073

[5] C. Miller and C. Valasek, "Adventures in automotive networks and control units," Tech. Rep., 2013, http://illmatics.com/car_hacking.pdf.

[6] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "CANAuth — a simple, backward compatible broadcast authentication protocol for CAN bus," in *ECRYPT Workshop on Lightweight Cryptography 2011*, 2011.

[7] O. Hartkopp, C. Reuber, and R. Schilling, "MaCAN - message authenticated CAN," in *10th Embedded Security in Cars Conference (ESCAR), Proceedings of*, 2012.

[8] B. Glas, J. Guajardo, H. Hacioglu, M. Ihle, K. Wehefritz, and A. Yavuz, "Signal-based automotive communication security and its interplay with safety requirements," in *10th Embedded Security in Cars Conference (ESCAR), Proceedings of*, 2012. [Online]. Available: http://www4.ncsu.edu/~aayavuz/2012-11-28_ESCAR_SecICCv2.pdf

[9] M. Sojka, P. Píša, O. Špinka, O. Hartkopp, and Z. Hanzálek, "Timing Analysis of a Linux-Based CAN-to-CAN Gateway," in *Thirteenth Real-Time Linux Workshop*. Schramberg: Open Source Automation Development Lab eG, 2011, pp. 165–172. [Online]. Available: http://lwn.net/images/conf/rtlws-2011/proc/Sojka.pdf

[10] "Rtems (real-time executive for multiprocessor systems)," http://www.rtems.org/.

[11] J. Baudy. (2014) Linux packet mmap. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt