Timing Analysis of Linux CAN Drivers

Michal Sojka, Pavel Píša
Czech Technical University in Prague, Faculty of Electrical Engineering
Technická 2, 166 27 Prague, Czech Republic
sojkam1@fel.cvut.cz, pisa@cmp.felk.cvut.cz

Abstract

Controller-Area Network (CAN) is a communication bus widely used in industrial and automotive systems. There exists many CAN drivers for Linux and one of them – Socketcan – is being merged to the mainline kernel. Another often used driver is called LinCAN. The drivers differ in the internal architecture and in this paper we provide the results of several experiments that compare latencies introduced by this two drivers. The aim is to evaluate suitability of these drivers for use in real-time applications.

1 Introduction

CAN-bus (Controller-Area Network) is a communication bus widely used in industrial and automotive systems. Not only that it simplifies wirings, but CAN-bus offers deterministic medium access algorithm for which it is very suitable for certain type of time-critical applications. A typical CAN-based network consists of multiple microcontrollers and one or a few more complex nodes which implement higher-level control algorithms. These more complex nodes can run Linux OS so there is a need for CAN-bus drivers. There exist several projects which offer CAN drivers for Linux. Probably the most known driver is Socketcan, which is already included in the mainline kernel. Another option is LinCAN driver. The basic difference between these two drivers is that Socketcan is built around standard Linux networking infrastructure, whereas LinCAN is a character device driver with its own queuing infrastructure designed for specific needs of CAN communication.

The Linux networking infrastructure is designed with the goal of achieving highest possible throughput for IP and similar high-bandwidth traffic, which is almost opposite to the goals of CAN bus – low bandwidth (CAN messages can hold up to 8 bytes of data payload) but precise timing. One can naturally ask whether and how Socketcan's usage of Linux networking layer influences communication latencies.

We have developed a tool for measuring round-trip times (RTT) on CAN-bus and we have conducted several experiments to evaluate RTT under different system loads and other conditions with the two above mentioned drivers.

The results of this work are useful for the following reasons:

1. Users can approximately know timing properties of a particular CAN-on-Linux solutions.
2. Developed tools can be used for benchmarking other hardware to find suitability for intended applications.
3. Driver developers can profile and improve their drivers and find factors contributing to worst-case delays in RX/TX paths.
4. Overhead caused by the use of generic Linux networking infrastructure in Socket-CAN implementation was quantified. Our finding is that the difference is not negligible, but it is still suitable for many real-time applications.

1.1 CAN description

CAN is a serial bus introduced in 1986 [1] by Robert Bosh GmbH. It uses a multi-master protocol based on a non-destructive arbitration mechanism, which grants bus access to the message with the highest priority without any delays. The message priority is determined by the message identifier, which identifies the content of the message rather than the address of the transmitter or the receiver.

Physical layer has to support transmission of two states – dominant and recessive. If no node transmits dominant state, the medium should remain in recessive state. Whenever any node transmits dominant state, the medium should be in dominant state. This property (sometimes called “wired OR”) is used by the medium access control (MAC) layer to accomplish the non-destructive arbitration mechanism. All nodes transmit bits synchronously and during transmission of the message arbitration phase the node checks for a difference between transmitted and medium states. If a difference is found, the node backs off. Since the arbitration requires synchronous bit transmission and the signal propagation speed is finite, the length of the bus is limited. For 125 kbit/s the maximum bus length is 500 m, for the maximum speed 1 Mbit/s, it is 30 m.

The message identifier, which is transmitted in message arbitration phase is either 11 or 29 bit long depending on whether extended message format is used. The length of the data payload can be up to 8 bytes and every message if protected by 15 bit CRC. Every message is acknowledged by an ACK bit. Transmitter transmits a receive bit and all receivers with correctly received message transmit that bit as dominant. This way, if no receiver is able to receive the message correctly, the transmitter can signal error to the upper layers.

2 Internal architecture of the drivers

In this section we briefly describe internal architecture of the tested drivers, especially the TX and RX paths, which mostly influence the time needed to process transmitted and received messages.

2.1 LinCAN

LinCAN driver presents CAN devices as character devices `/dev/canX`. Applications use the driver through the standard UN*X interface represented by the five functions: `open()`, `read()`, `write()`, `ioctl()` and `close()`. The heart of the driver is a flexible queuing infrastructure [2], which allows the driver to be used simultaneously from userspace, IRQ handlers and RTLinux programs. For every user of the driver (open file descriptor in userspace or an RTLinux application), there one queue for message transmission and one for reception.

The queue has preallocated a fixed number of slots to hold received messages or messages waiting for transmission. By default every queue can hold up to 64 messages. The queuing infrastructure does not support message ordering according to their priority, but there is (currently unused) support for having multiple queues for different priorities.

LinCAN is distributed as a out-of-tree driver and there are no plans to push it to mainline.

2.1.1 Transmit path

To transmit a message, the application calls `write()` system call on a `/dev/canX` file descriptor. This operation executes `can_write()` which stores the message in the queue and wakes the chip driver to send the messages in the queue. The driver simply sends out the message from the queue. If the chip is busy by sending the previous messages, nothing happens and the queue is processed later when TX completion IRQ arrives.

2.1.2 Receive path

Interrupt handler reads the message and copy it directly from the IRQ context to all the queues connected to the particular device (`canque_filter_msg2edges()`). Then any readers waiting in `read()` system call are woken up.

2.2 Socketcan

Socketcan is a driver built around Linux networking infrastructure. As of 2.6.31, most of it is already merged into the mainline. Besides the drivers themselves, Socketcan implements several higher-level protocols (BCM, ISO-TP and raw) and provides useful userspace utilities. Applications access the socketcan functionality through BSD sockets API which is commonly used for network programming not only in the UN*X world.

2.2.1 Transmit path

After the message is sent through `sendmsg()` or similar system call, it is passed to `can_send()`, which queues it in the device queue by calling `dev_queue_xmit()`. This function uses a queuing discipline (qdisc) to queue the message in the driver

TX queue. The default qdisc is `pfifo_fast` with the queue length of 10. When a message is enqueued `qdisc_run()` is called to process the queue. If the driver is not busy transmitting another message, it transmits it immediately, otherwise the qdisc's queue is appended (see `__netif_reschedule()`) to a per-CPU queue and processed later in `NET_TX_SOFTIRQ`.

2.2.2 Receive path

Interrupt handler reads the message[1] and calls `netif_rx()` which queues the message in per CPU `softnet_data` queue and schedules `NET_RX_SOFTIRQ` for later processing. It is important to know that this queue is shared for all kinds of network traffic, not only for CAN.

The softirq runs `net_rx_action()` which polls the `softnet_data` queue in `process_backlog()`, and calls `netif_receive_skb()` for every message there. Note there is also rate limiting in this softirq, so if there are many packets from other sources, our CAN messages are delayed by additional delays.

Within `netif_receive_skb()` the list of registered packet types is traversed and the message is passed to the handler, which, in case of socketcan, is `can_rcv()`.

`can_rcv()` traverses the list of sockets and delivers the message to all sockets interested in receiving the message. For raw sockets this is accomplished by calling `raw_rcv()` which clones the socket buffer and calls `sock_queue_rcv_skb()`, which copies the data to the socket receive queue by `skb_queue_tail()` and userspace is woken up in `sock_def_readable()`.

3 Testing

As one can see from the description of the driver internals in the previous section, TX and especially RX path is considerably more complicated in case of Socketcan. Given the fact, that CAN message is up to 8 byte in length, maximum bitrate is1 Mbit/s and deterministic timing is often required, the question which we try to answer in this paper is whether the Socketcan overhead is not too high for serious use.

3.1 Testbed description

To answer the question, we have set up a testbed to compare the drivers under the same conditions. The testbed was a Pentium 4 box with a Kvaser PCIcan-Q (quad-head PCI card based on SJA1000 CAN controller). The CPU is a single core and supports hyper-threading (HT).

In most experiments, we measured the round-trip time (RTT) i.e. the time it takes to send the message to the other end and back. To measure the RTT we used *canping* tool [2]. It send the messages according to command line switches and measures the time elapsed until receiving a response. The next message is sent after a response to the previous one is received. The measured times are statistically processed and histograms can be generated from them. To access the CAN driver, canping uses VCA (Virtual CAN API) library[3], which, which provides common API for different driver back-ends.

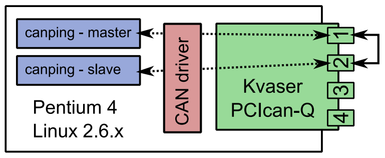

FIGURE 1: *Testbed setup*

We simply connected CAN card's output 1 with output 2 (see Fig. 1) and run two instances of canping on one computer – one in slave mode to only respond to messages generated by the second canping in master mode. Both instances was run with real-time priority and have locked all their memory (`mlockall()`).

In every experiment, we sent ten thousands messages. We have repeated all the experiments with several different kernels. For real-time application it is natural to measure the performance under rt-preempt kernels, but for other use, results for non-rt kernels can also be useful.

For all testes CAN baudrate was set to 1 Mbit/s and all messages carried 8 fixed bytes of data payload. The length of messages was measured on oscilloscope to be 112 μs, but every message is followed by 7 recessive end of frame bits and 3 bits of inter frame space. Therefore the shortest possible bus time needed for sending the message forth and back is $2 * (112 + 7) + 3 = 241\mu$s.

[1] This is true for `sja1000.c` driver. Other drivers, such as `mscan.c` read the messages entirely in softirq.
[2] http://rtime.felk.cvut.cz/gitweb/canping.git
[3] http://ocera.cvs.sourceforge.net/viewvc/ocera/ocera/components/comm/can/canvca/libvca/

The computer didn't run unnecessary services such as cron. Also irqbalance daemon was not running so that the interrupts went always to the first CPU (hyper-threading). Since there were some differences in results depending on whether hyper-threading was used or not, we also run the tests on kernels with `maxcpus=1` command line parameter. Such test are marked by `:maxcpus=1` suffix to kernel version.

All tests were run at full CPU clock speed, which was 2.4 GHz, as well as at the lowest one (300 MHz) by which we try to simulate an embedded system, where CAN is used a lot.

All tests used socketcan from trunk at SVN revision 1009. The exception is the kernel 2.6.31-rc7, for which the version in mainline was used. For LinCAN we used a version from CVS dated 2009-06-11.

3.2 Results

To graph the results, we use so called latency profiles. It is a cumulative histogram with reversed vertical axis displayed in logarithmic scale. This way we can see the exact number of packets with worst latencies at the bottom right part of the graph. You can see it in Figure 2, where we measured RTT on an unloaded system. It can be seen that the LinCAN has lower overhead that Socketcan, which was an expected result. Another observation is that worst-case driver overhead is approximately the same as message transmission time. For LinCAN the overhead is $450-241 = 209\,\mu$s, for Socketcan $525-241 = 284\,\mu$s.

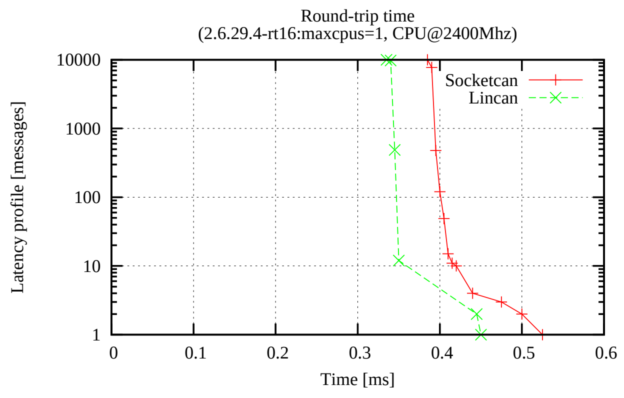

FIGURE 2: *Round-Trip Time on the unloaded system, 2.4 GHz*

If the CPU frequency is lowered to 300 MHz, then the overhead is much larger for both LinCAN and Socketcan (Fig. 3).

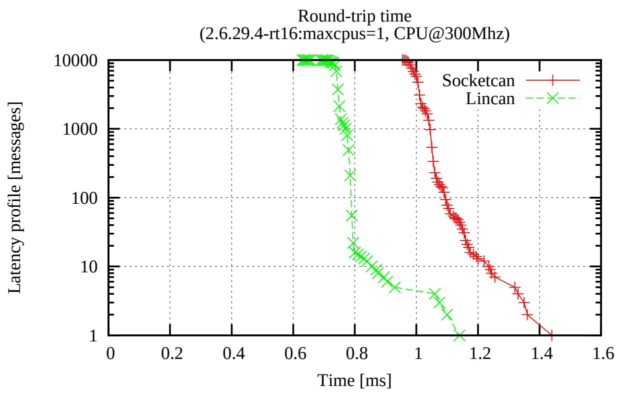

FIGURE 3: *Round-Trip Time on the unloaded system, 300 MHz*

Since Socketcan shares the RX path with the other parts of Linux networking stack, in the next experiment we tried to measure how other network traffic influences CAN latency. We generated Ethernet load to the CAN testbed from another computer. The used Ethernet card was Intel's `e100` (100 Mbps). We tried the following load types: flood ping, flood ping with 64 kB datagrams and TCP load (high bandwidth SSH traffic). It turned out that the worst influence was caused by flood ping with 64 kB datagrams (see Fig. 4). With default system configuration both drivers exhibit additional latency of approximately 300 μs.

However, under `-rt` kernels, most interrupt handlers run in thread context and it is possible to set the priority of those IRQ threads. Therefore we increased the priority of the IRQ thread handling our CAN card interrupts (the interrupt line was not shared with any other peripheral). The results can be also seed in Fig. 4 – lines marked as "boosted IRQ prio". As can be seed, LinCAN latency goes down to the same values as without any load. Socketcan is not influenced by this tuning at all.

Socketcan has another parameter which can be used to tune packet reception – `netdev_budget` sysctl. By this knob, one can tune how many received packets is processed in one run of `NET_RX_SOFTIRQ`. The default value of this parameter is 300 and as can be seen again in Fig. 4, changing this value has no effect on the Socketcan latency. Further investigation showed that with our tests there were rarely more that 10 pending packets and since softirqs are re-executed up to 10 times if they are pending at the time they finish execution, even lowering budget to 1 had no effect.

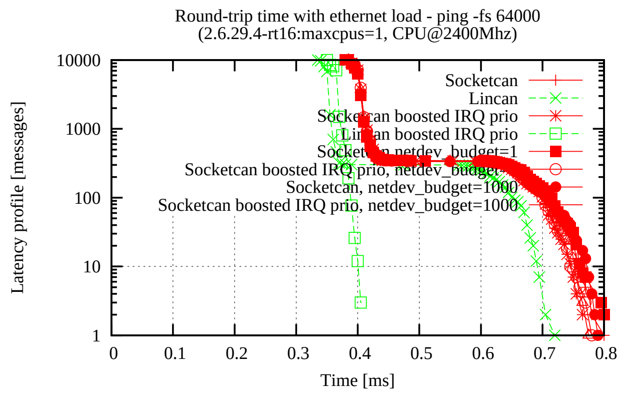

FIGURE 4: *Round-Trip Time with Ethernet load*

For the sake of completeness, we also tested whether Ethernet transmission influences CAN latency. As can be seen from Fig. 5, there is almost no influence by this type of load.

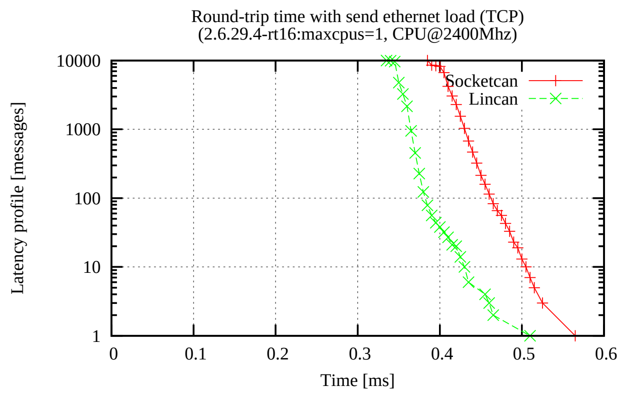

FIGURE 5: *Round-Trip Time with Ethernet transmission*

Also, there is almost no influence by non-realtime CPU load, which was generated by `hackbench` tool[4] (Fig. 6).

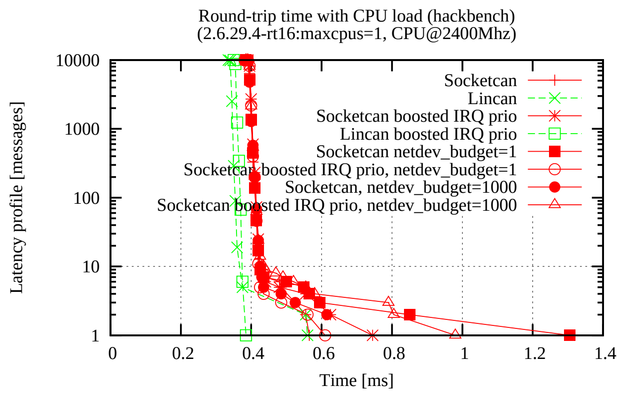

FIGURE 6: *Round-Trip Time with CPU loaded by* ***`hackbench -pipe 20`***

While testing the drivers on non-rt kernels, the worst-case latency was usually very high as can be seen in Fig. 7. In this particular experiment, Socketcan experienced one 22 ms delay, but in other experiments this happened even to LinCAN. We didn't investigated the source of this latency in detail, but it was probably caused by `ath5k` driver because after unloading this driver, this latency "peaks" disappeared.

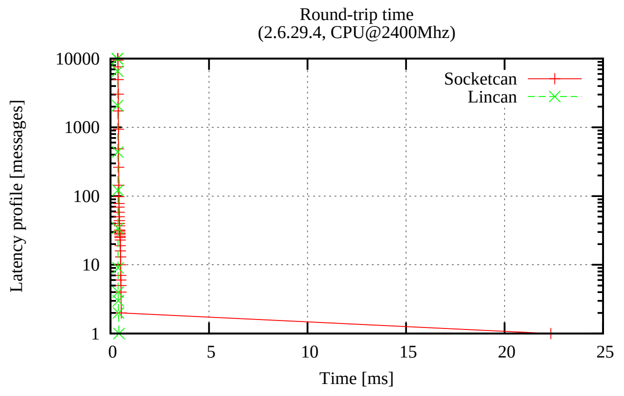

FIGURE 7: *Round-Trip Time on the unloaded system, 2.4 GHz, non-rt kernel*

When we unloaded all unnecessary modules from the kernel, the latencies went down for LinCAN (see Fig. 8). The reason is that LinCAN does all the RX processing in hard-IRQ context, whereas Socketcan employs a soft-IRQ which is shared by all network devices.

[4]http://devresources.linux-foundation.org/craiger/hackbench/

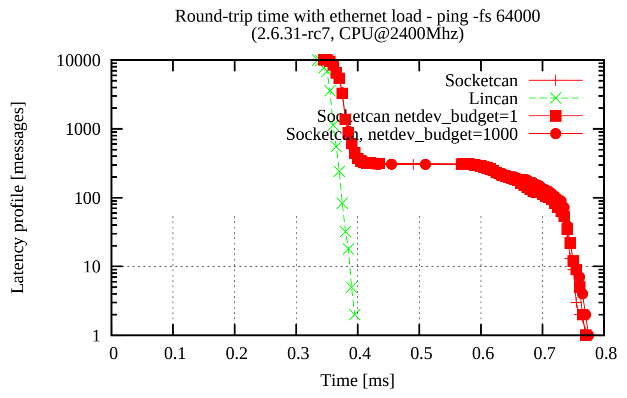

FIGURE 8: *Round-Trip Time with Ethernet load, 2.4 GHz, non-rt kernel, no unnecessary modules*

Another interesting results result is performance of virtual CAN drivers. This driver can be used to test CAN applications on places where real CAN hardware is not available. It seems there is some problem with how Socketcan implements this. Under Debian kernel 2.6.26-2-686, which is configured with `PREEMPT_NONE=y`, Socketcan latencies raised up to 550 ms (Fig. 9).

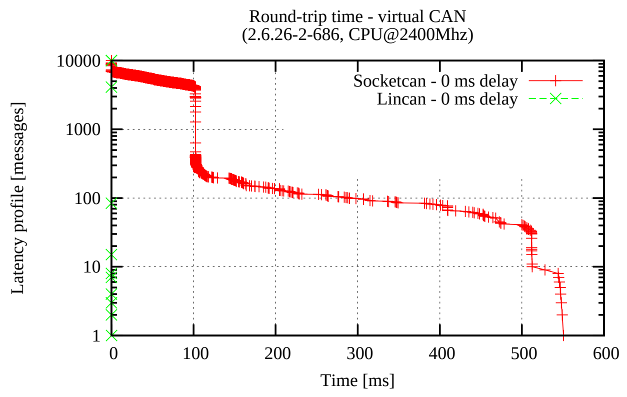

FIGURE 9: *Round-Trip Time on virtual CAN driver*

On kernel 2.6.31-rc7, the results seem also strange for Socketcan (Fig. 10). For all other kernels the results were as expected – worst-case latency around several hundreds microseconds.

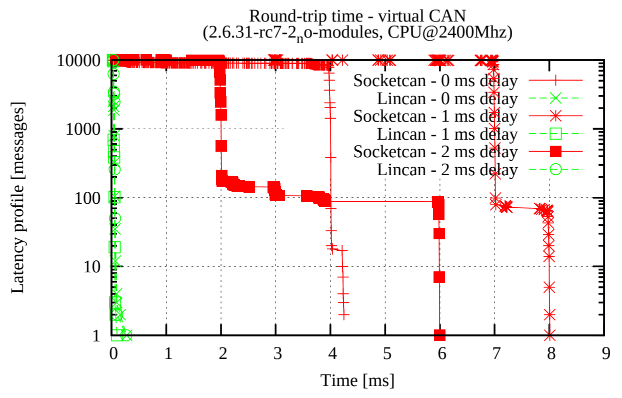

FIGURE 10: *Round-Trip Time on virtual CAN driver*

Last but not least, people are continuously improving Linux network stack and under kernel 2.6.31-rc7, the performance of Socketcan in only slightly worse that of Lincan (see Fig. 11).

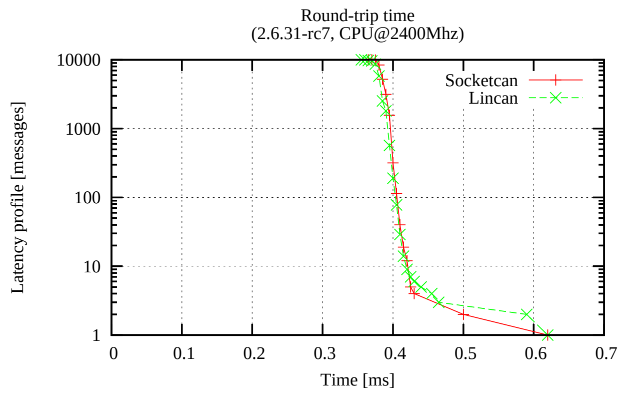

FIGURE 11: *Round-Trip Time on recent kernel and unloaded system*

The complete results of all tests for several different kernels can be found on our web page[5]. If somebody would like to reproduce our results all the code, configurations etc. is available from Git repository[6].

4 Conclusion

The experiments described in this paper show the differences between two distinct approaches to writing Linux CAN driver – character device and network device. In most cases the character device approach

[5]http://rtime.felk.cvut.cz/can/benchmark/1/
[6]http://rtime.felk.cvut.cz/gitweb/can-benchmark.git

has lower overhead but it seems that recent improvements in Linux kernel decrease the overhead of the network device approach.

One area where character based drivers still win is when the system is loaded by receiving non-CAN traffic. Under preempt-rt kernels, one can increase the priority of the IRQ thread which handles the CAN hardware and latencies go down with character device approach, whereas network device approach exhibits higher latencies regardless of IRQ thread priority. When compared to the case with without non-CAN traffic, the worst-case latency is approximately doubled. This values can be still suitable for most real-time applications. If this latency is not acceptable, one would need to split the queue between hard- and soft-IRQ to several queues of different priorities with CAN messages going to the high-priority queue and all other traffic to queues with lower priorities.

Finally, we found that timing of Socketcan's `vcan` driver is quite odd under some kernels. It seems, it has some connection to CPU scheduling and it needs to be further investigated and fixed.

References

[1] CiA, "CAN history," 2001. [Online]. Available: http://www.can-cia.org/index.php?id=161

[2] P. Píša, "Linux/RT-Linux CAN driver (LinCAN)," 2005. [Online]. Available: http://cmp.felk.cvut.cz/ pisa/can/doc/lincandoc-0.3.pdf