Timing Analysis of a Linux-Based CAN-to-CAN Gateway

Michal Sojka[1], Pavel Píša[1], Ondřej Špinka[1], Oliver Hartkopp[2], Zdeněk Hanzálek[1]
[1]Czech Technical University in Prague
Technická 2, 121 35 Praha 6, Czech Republic
{sojkam1,pisa,spinkao,hanzalek}@fel.cvut.cz

[2]Volkswagen Group Research
Brieffach 1777, 38436 Wolfsburg, Germany
oliver.hartkopp@volkswagen.de

Abstract

In this paper, we thoroughly analyze timing properties of CAN-to-CAN gateway built with Linux kernel CAN subsystem. The latencies induced by this gateway are evaluated under many combinations of conditions, such as when traffic filtering is used, when the gateway is configured to modify the routed frames, when various types of load are imposed on the gateway or when the gateway is run on different kernels (both rt-preempt and vanilla are included). From the detailed results, we derive the general characteristics of the gateway. Some of the results apply not only for the special case of CAN-to-CAN routing, but also for the whole Linux networking subsystem because many mechanisms in the Linux networking stack are shared by all protocols.

The overall conclusion of our analysis is that the gateway is in pretty good shape and our results were used to support merging the gateway into Linux mainline.

1 Introduction

Controller Area Network (CAN) is still by far the most widespread networking standard used in the automotive industry today, even in the most recent vehicle designs. Although there are more modern solutions available on the market [1, 2] (such as FlexRay or various industrial Ethernet standards), CAN represents a reliable, cheap, proven and well-known network. Thanks to its non-destructive and strictly deterministic medium arbitration, CAN also exhibits very predictable behavior, making it ideally suited for real-time distributed systems. Because of these indisputable qualities, it is unlikely that the CAN is going to be phased out in foreseeable future.

The maturity of CAN technology means that there exist a huge number of CAN compatible devices on the market. It is therefore quite easy to build prototypes of complex systems by just connecting off-the-shelf devices and configuring them to do what it desired. Sometimes, however, there are devices (or complete subsystems) that are not compatible with each other. They may use different protocols or simply use fixed CAN IDs, that collide with other devices. Then, it is necessary to separate those devices using a gateway that ensures that only the traffic needed for communication with the rest of the system passes the gateway, perhaps after being modified to not collide with the rest of the system.

For this reason a widely configurable CAN gateway has been implemented inside the Linux kernel, which is based on the existing Linux CAN subsystem (PF_CAN/SocketCAN [3]) and can be configured via the commonly used netlink configuration interface. This gateway is designed for CAN-to-CAN routing and allows frame[1] filtering and manipulation of the routed frame content. Obviously, such a gateway must satisfy very strict real-time requirements, especially if it connects critical control systems. Therefore, the gateway had to undergo a set of comprehensive tests, focused on measuring latencies intruded by the gateway under various conditions.

[1]CAN uses the term "frame" for what other networks call packet or message.

The results of this testing are reported in this paper. The complete data set, consisting of gigabytes of data and more than one thousand graphs, as well as the source codes of our testing tools, are available for download in our public repositories [4, 5]. This allows other people interested in this topic to independently review our results and methods, as well as to use them as a base for their own experiments. Our methods and results are relevant not only for the special case of CAN-to-CAN routing but, since Linux networking subsystem forms the core of many other protocols, also for other networks including Ethernet, Bluetooth, Zigbee etc.

The paper is organized as follows: the next section describes the setup of our testbed and how we measured the gateway latencies. Section 3 summarizes the main results found during our testing. We give our conclusion in Section 4.

2 Testbed Setup

The testbed, used for gateway latency measurements, is depicted in Figure 1 and consists of a standard PC and the gateway. The PC is Pentium 4 running at 2.4 GHz with 2 GB RAM, equipped with Kvaser PCI quad-CAN SJA1000-based adapter. The gateway is an embedded board based on MPC5200B (PowerPC) microcontroller running at 400 MHz. There are two CAN buses that connects the PC with the gateway. The PC generates the CAN traffic on one bus and looks at the traffic routed via the gateway on the other bus. The gateway is also connected to the PC via Ethernet (using a dedicated adapter in the PC). This connection serves for booting the gateway via TFTP and NFS protocols, for configuring it via SSH, and also to generate Ethernet load to see how it influences the gateway latencies.

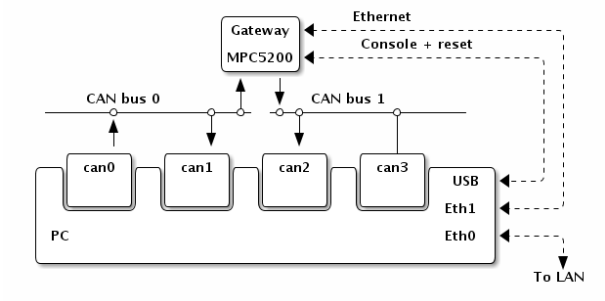

FIGURE 1: *Testbed configuration.*

The software configuration is kept as simple as possible in order to make the results not disturbed by unrelated activities. The gateway runs only a Linux kernel, a Dropbear SSH server and, obviously, the gateway itself. On the PC, a stripped-down Debian distribution is used. The tasks that generate the test traffic and measure the gateway latency are assigned the highest real-time priority and their memory is locked in order to prevent page-faults. SocketCAN was used on both the gateway and the PC as the CAN driver.

2.1 Measurement Methodology

To measure the gateway latency, we generate CAN traffic in the PC and send it out from *can0* interface. As can be seen in Figure 1, this interface is directly wired to the *can1* interface of the PC as well as to one interface of the gateway. The *can1* interface is used to receive the frames to determine the exact time when each frame actually appears on the bus[2]. This is necessary in order to exclude various delays such as queuing time in the *can0* transmit queue. When a frame is received on *can1* interface, it is timestamped by the driver in its interrupt handler. These timestamps are sufficiently precise for our measurements.

The frames routed through the gateway are received on *can2* interface of the PC. Again, these frames are timestamped the same way as was described in the previous paragraph. The total latency is then calculated by simply subtracting the timestamps measured on the *can2* and *can1* interfaces (see Figure 2). It is worth noting that both timestamps are obtained using the same clock (in our case timestamp counter register of the PC's CPU), which ensures that the results are not influenced by the offset of non-synchronized clocks.

To calculate the latency, we need to determine which received frame corresponds to which transmitted one, and this mechanism must be able to cope with possible frame losses or frame modifications in the gateway. For this purpose, the first two bytes of the data payload are used to store a unique number that is never modified by the gateway. This number serves as an index to a lookup table, which stores the timestamps relevant to the particular frame. This allows for easily detection of frame losses. When the corresponding entry in the lookup table contains just one timestamp after a certain timeout, which is set to 1 s by default, the frame is considered lost.

[2]We could also use kernel provided TX timestamps for this, but it somehow didn't work in our setup.

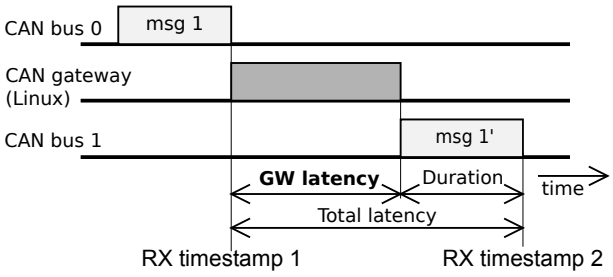

FIGURE 2: *Calculation of gateway latency.*

Since we are interested only in the latency introduced by the gateway (see GW latency in Figure 2), we subtract from the total latency the duration of the frame transmission, where we take into account the stuff bits inserted by CAN link layer.

The sources of our testing tools and the individual test cases can be found in our git repository [4] under `gw-tests` directory.

2.2 What Was Tested?

Our goal was to measure the properties of the gateway under a wide range of conditions. These included:

1. The **gateway configuration** such as frame filters, frame modifications, etc.
2. **Additional load** imposed on the gateway system. The following types of load were considered: no load; CPU load i.e. running `hackbench`[3] on the gateway; Ethernet load i.e. running `ping -f -s 60000 -q gw` on the PC with `gw` being the IP address of the gateway.
3. Type of **CAN traffic**. We tested the gateway with three kinds of traffic: *One frame at a time*, where the next frame was sent only after receiving of the previously sent frame from the gateway; *50% bus load*, where frames were sent with a fixed period which was equal to two times the transmission duration and finally, *100% bus load* (flood), where frames were sent as fast as possible.
4. Linux **kernel version** used on the gateway. The following versions were tested: 2.6.33.7, 2.6.33.7-rt29, 2.6.36.2, 3.0.4 and 3.0.4-rt14.

We run all experiments for all possible combinations of the above conditions which resulted in 653 experiments. The interested reader can find the full set of the results at [5]. The most important findings are discussed later in this paper.

2.3 Presentation of Results

In every experiment we measured the latency of multiple (in most cases 10000) frames. The results are presented below in a sort of histogram called "latency profile". Figure 3 shows how the latency profile (at the bottom) relates to the classical histogram (at top). In essence, latency profile is a backward-cumulative histogram with logarithmic vertical axis.

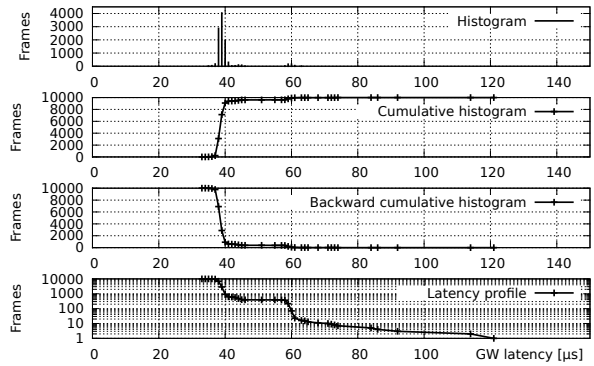

FIGURE 3: *How is latency profile constructed.*

The advantage of using latency profiles is that the worst-case behavior (bottom right part of the graph) is "magnified" by the logarithmic scale. More formally, the properties of the latency profile are as follows: Given two points (t_1, m_1) and (t_2, m_2) from a latency profile, where $t_1 < t_2$, we can say that $m_1 - m_2$ frames had the latency in the range (t_1, t_2). Additionally, the rightmost point (t_w, m_w) means that there were exactly m_w frames with the worst-case latency of t_w.

2.4 Measurement Precision

We conducted a few experiments to evaluate the precision of measuring the latencies with our setup. First, we measured the total frame latencies by two means: (1) by a PC as described above and (2) by an independent CAN analyzer by Vector[4]. In the other experiments we used only the method 1 (PC) as it allows for full automation of the measurement, whereas

[3]Hackbench repository is at http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git.
[4]We used analyzer called CANalyzer (http://www.vector.com/vi_canalyzer_en.html)

method 2 (CANalyzer) requires a lot of manual work to save and process the measured data. The results of the comparison can be seen in Figure 4. It can be observed that the time resolution of CAN analyzer is only 10 μs while the PC was able to measure data with far better resolution, thanks to the support of high resolution timers in Linux. Our histogram uses bins of 1 μs. The difference between the two methods is most of the time below 10 μs. Occasionally (for less then 0.01% of frames), we got a bigger difference. Such precision is sufficient for our experiments.

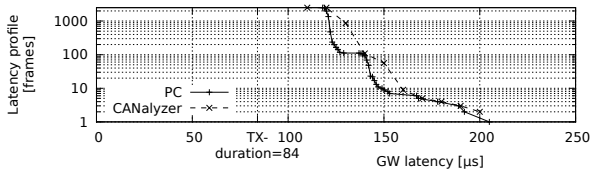

FIGURE 4: *Comparison of total latency measurement by PC and by CANalyzer, payload: 4 bytes.*

Since the PC not only timestamps the received frames but also generates the Ethernet load to artificially load the gateway, this additional activity influences the precision of the measurements. To see how big this influence is, we ignored the frames received on *can2* interface and calculated the latency l from RX timestamp on *can1*, the time before the frame was sent (in user space) to *can0* and TX duration (i.e. $l = t_{\mathrm{RX}} - t_{\mathrm{send}} - t_{\mathrm{duration}}$). The ideal result would be a vertical line at time 0, i.e. all frames were received immediately after being transmitted, but in reality (see Figure 5) we get a sloped line around 31 μs, because the measurement includes the overhead of sending and receiving operation. The second line in the graph shows, the generating the Ethernet traffic decreases the measurement precision by approximately 30 μs, if we ignore a few out-layers, or by about 200 μs if we do not ignore them. Both number are far below the increase of gateway induced latency (which is in order of milliseconds – see Section 3.3) and therefore, the precision is sufficient even when the PC generates the Ethernet load.

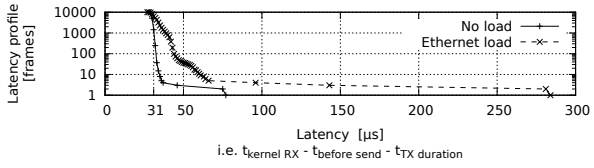

FIGURE 5: *Influence of Ethernet load generator on measurement precision (no GW involved).*

3 The Results

In this section we present the main results of the analysis. Some of the results gives a nice insight in how networking in Linux works and how Linux schedules various activities.

3.1 Simple gateway

In the first experiment we measured the behavior of the gateway which simply routes all frames from one bus to another without any modifications.

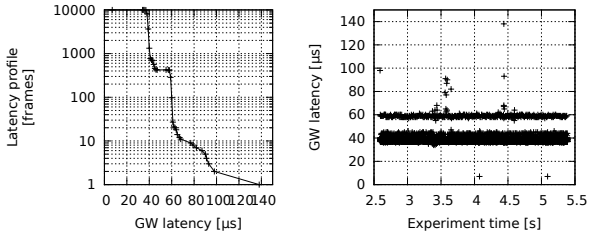

FIGURE 6: *Simple gateway – latency profile and the corresponding time chart. Conditions: GW kernel 2.6.33.7, traffic: oneatatime, load: none, payload: 4 bytes.*

The latency profile and the corresponding time chart is shown in Figure 6. It can be seen that the best-case gateway latency is about 35 μs and the worst-case is about 140 μs. Most of the observed latencies fall into two groups, which are split by a gap around 50 μs. We attribute this gap to timer interrupts which were triggered during processing the frame in the gateway. It can be seen that the cost of the timer interrupt is about 20 μs.

3.2 Batched Processing of Frames

Linux kernel processes the incoming CAN frames (or packets in Ethernet networks) in batches. Basically, when RX soft-irq is scheduled, it runs in a loop and tries to process all frames sitting in receive buffers (either in hardware or in software). The graph in Figure 7 shows nicely the effect of this. If we compare the latencies when the CAN traffic was generated with *one frame at a time* and *flood* methods, it can be seen that in the former case, the overhead of scheduling the RX soft-irq is always included (the latency profile starts at 35 μs), whereas in the latter

case, the overhead is reduced. Whenever the gateway receives a frame just when it finishes processing of the previous frame, it does not exit the soft-irq and continues processing the new frame. Therefore, the best-case latencies are much lower in that case (the latency of 0 is of course caused by measurement inaccuracies). The worst-case is about the same in both cases.

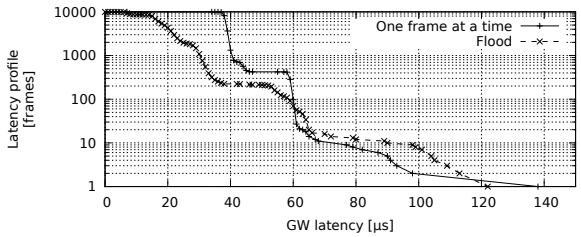

FIGURE 7: *The effect of batched frame processing. Conditions: GW kernel 2.6.33.7, load: none, payload: 4 bytes.*

3.3 Effect of Loading the Gateway

Figure 8 shows the effect of loading the gateway. The *No load* line is the same as in the graphs before. The *CPU load* line represents the case when the CPU of the gateway was heavily loaded by doing many inter-process communications in parallel (hackbench). This approximately doubles the worst-case latency from 140 μs to 250 μs. The Ethernet load (flood ping), however, influences the gateway much more significantly. As it was shown in [6], this is due to the shared RX queue for both CAN and Ethernet traffic. Therefore, processing of CAN frames has to wait after the Ethernet packets (in our case big ICMP echo requests) are processed.

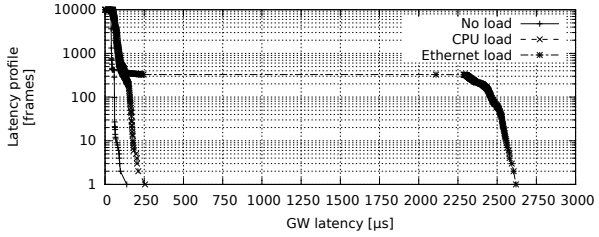

FIGURE 8: *The effect of loading the gateway. Conditions: GW kernel 2.6.33.7, traffic: one frame at a time, payload: 4 byte.*

[5]Extended Frame Format
[6]Standard Frame Format

3.4 Frame Filtering

The SocketCAN gateway allows for filtering the frames based on their IDs. There are two kinds of filter implementations. First implementation (used for all EFF[5] frames) puts the filtering rules into a linked list. Whenever a frame is received, this list is traversed and when a match is found, the frame is routed to the requested interface. The second implementation is optimized for matching single SFF[6] IDs. Since there is only 2048 distinct SFF IDs, the filter uses the frame ID as an index to the table and the destination interface if found without traversing a potentially long list.

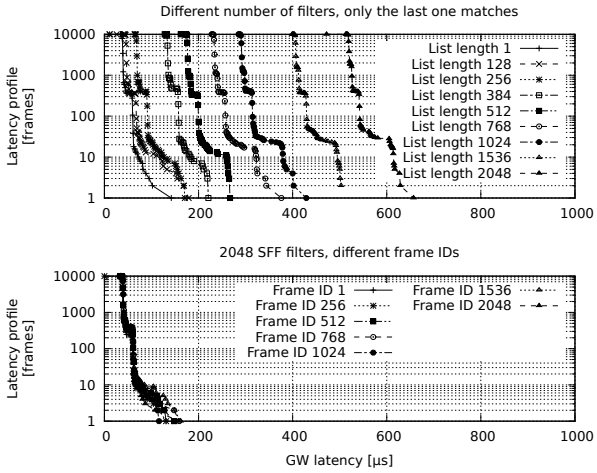

FIGURE 9: *Gateway with filters. Top: latency for different number of EFF filter, bottom: single SFF ID filters. Conditions: GW kernel 2.6.33.7, traffic: one frame at a time, load: none.*

The top graph in Figure 9 show the cost of having a different number of EFF filters in the list and only the last one matches the sent frames. The gateway latency obviously increases with the number of filters. From a more detailed graph [5] it was nicely visible when the list started to be bigger than CPU cache and the latency started to increase quicker. In our case this boundary was hit for about 80 filters.

Additionally, when the filter list is too long and CAN frames arrives faster, the gateway is no longer able to handle all of them and starts dropping them. This is visible in the appropriate graph at [5].

The bottom graph in Figure 9 shows that the single ID SFF filters perform the same for all frame IDs even when there is 2048 distinct filtering rules.

3.5 Frame Modifications

Besides routing the frames, the gateway is also able to modify them. There are different operations which can be applied to the frames: AND, OR, XOR, SET and two checksum generators. The graph in Figure 10 shows the cost of modifying the frames. In essence, most of the cost comes from copying the socket buffer before modifying it. The difference between different modifications is negligible.

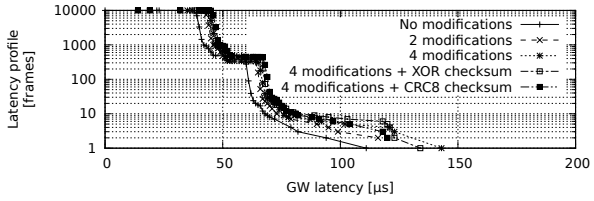

FIGURE 10: *The Cost of Frame Modification. Conditions: GW kernel 2.6.33.7, traffic: one frame at a time, load: none, payload: 8 bytes.*

3.6 Differences between Kernels

Figure 11 shows the differences between different kernel versions. We tried to keep the configs[7] of the kernels as similar as possible by using `make oldconfig`. From the graph, we can observe two things. First, with increasing non-rt versions, the latency increases as well. The difference between 2.6.33 and 2.6.36 is about 10 μs, the difference between 2.6.36 and 3.0 is smaller – about 2 μs.

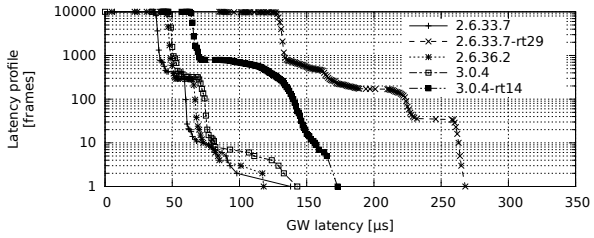

FIGURE 11: *Different kernels. Conditions: traffic: one frame at a time, load: none, payload: 4 bytes.*

Second, the latency of rt-preempt kernels is higher than any of non-rt kernels. This is obvious, because the preemptivity of the kernel has its costs. Interestingly, 3.0-rt is much better in this regard than 2.6.33-rt. Compared to non-rt kernels, 3.0-rt increases latencies by about 20 μs, whereas 2.6.33-rt increases them by almost 100 μs.

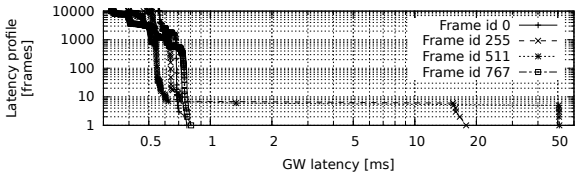

FIGURE 12: *Huge latencies in rt-preempt kernel. Conditions: 2048 EFF filters, GW kernel: 3.0.4-rt14, traffic: one frame at a time, load: none, payload: 2 bytes.*

There seems to be a bug in rt-preempt kernels. For certain workloads, we get the latencies as big as 50 ms. This can be seen in Figure 12. Interestingly, we get such a high latency regularly, exactly every one second. This behavior appears in both -rt kernels tested and we will try to find the source of the latency later. Additionally, kernel 3.0.4-rt14 hangs with heavy Ethernet load, which is also something we want to look at in the future.

3.7 User-Space Gateway

The SocketCAN gateway is implemented in kernel space. It is interesting to see the differences when the gateway is implemented as a user space program. The user space gateway was executed with real-time scheduling policy (SCHED_FIFO) with priority 90.

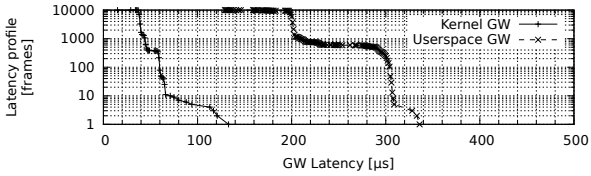

FIGURE 13: *Kernel-space vs. User-space gateway. Conditions: kernel: 2.6.33.7, load: none, traffic: one frame at a time, payload: 2 bytes.*

In Figure 13 can be seen that the time needed to route the frame in user-space is about three times bigger then with the kernel gateway.

[7]The configs of our gateway kernels can be found at https://rtime.felk.cvut.cz/gitweb/can-benchmark.git/tree/HEAD:/-kernel/build/shark

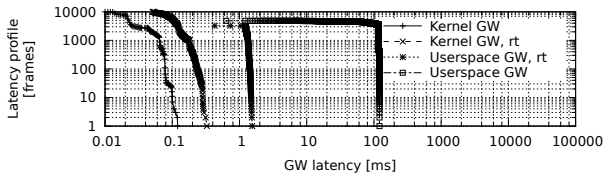

FIGURE 14: *Kernel-space vs. User-space gateway under heavy traffic. Conditions: kernel: 2.6.33.7 (non-rt and rt), load: none, traffic: flood, payload: 2 bytes.*

Figure 14 shows the gateway latencies with flood CAN traffic. One can see that the user-space gateway under non-rt kernel drops some frames (the gap at the top) and exhibits latencies up to 100 ms. The latencies are caused mainly by queuing frames in receiving socket queues. Since the user space had no chance to run for a long time, the queue becomes long and eventually some frames are dropped. The kernel simply has "higher priority" than anything in the user space. With -rt kernel, the situation is different. The priorities of both user and kernel threads can be set to (almost) arbitrary values, which allows to reduce latencies of the user-space gateway down to 2 ms.

3.8 Multihop Routing

In the last experiment, we modified the kernel gateway to allow routing a single frame multiple times via virtual CAN devices. This allows us to split the overall latency into two parts. The first part is the overhead of interrupt handling and soft-irq scheduling and the second part is the processing of the frame in CAN subsystem. The latter part can be derived from Figure 15, by looking at the difference between consecutive lines (and dividing it by two). We get that CAN subsystem processing takes about 10 μs. The rest (from ca. 60 μs to 130 μs) is the overhead of the rest of the system.

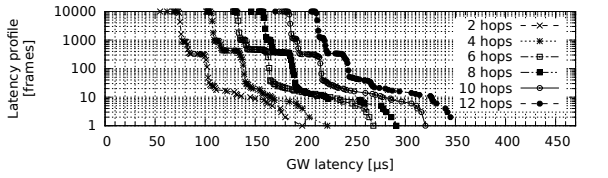

FIGURE 15: *Latencies of multi-hop routing via virtual CAN interfaces.*

It is interesting to compare these graphs for different kernel versions. From that, one can see where the increase of overhead comes from. The interested reader is referred to our web site [5].

4 Conclusion

This paper presented the timing analysis of a Linux-based CAN-to-CAN gateway and studied influence of various factors (like CPU and bus load, kernel versions etc.) on frame latencies. The results indicate that the gateway itself introduces no significant overhead under real-life bus loads and working conditions and can reliably work as a part of a distributed embedded system. Our results were used to support merging the gateway into Linux mainline. The gateway should appear in Linux 3.2 release.

On the other hand, it must be noted that especially excessive Ethernet traffic or improperly constructed frame filters can lead to significant performance penalties and possible frame losses. The CAN subsystem, which forms the core of the examinated CAN gateway, is inherently prone to problems under heavy bus loads, not only on CAN bus, but also on other networking devices, as was already demonstrated in our previous work [6]. Nevertheless, the described gateway is a standard and easy-to-use solution, integrated in Linux kernel mainline, and therefore represents the framework of choice for most developers.

It was also clearly demonstrated that the kernel-space solution works much better than the user-space solution, and that it can be beneficial to use standard non-rt kernels (providing that the gateway runs in kernel-space). This allows to avoid greater overhead and resulting performance penalty of rt kernels, providing that the standard kernel is properly configured.

Finally, our benchmarks revealed a few problems in -rt kernels. We will investigate these problems as our future work.

References

[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 communi-

cation systems," *Proceedings of the IEEE*, vol. 93(6), pp. 1204–1223, 2005.

[3] "The SocketCAN poject website," http://developer.berlios.de/projects/socketcan.

[4] M. Sojka, "CAN Benchmark git repository," 2010. [Online]. Available: http://rtime.felk.cvut.cz/gitweb/can-benchmark.git

[5] M. Sojka and P. Píša, "Netlink-based CAN-to-CAN gateway timing test results," 2011. [Online]. Available: http://rtime.felk.cvut.cz/can/benchmark/2.1/

[6] M. Sojka and P. Píša, "Timing analysis of Linux CAN drivers," in *Eleventh Real-Time Linux Workshop*. Homagstr. 3 - 5, D-72296 Schopfloch: Open Source Automation Development Lab, 2009, pp. 147–153.