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

Michal Sojka, Pavel Píša, Oliver Hartkopp, Ondřej Špinka and Zdeněk Hanzálek, *Member, IEEE*

***Abstract*—The aim of this paper is to briefly introduce an open-source Linux-based CAN-to-CAN gateway and to provide a thorough timing analysis and measurement of packet latencies, introduced by such a gateway into a real-time networked system. The testing methods and results are presented in detail and moreover, the complete data set, consisting of gigabytes of data and more than 600 graphs, as well as the source codes of our testing tools, are available for download in our public repositories. This allows other researches 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 also for the whole Linux networking system in general. The introduced gateway itself is a public available open-source component using the existing Linux Kernel networking infrastructure for the Controller Area Network protocol family (PF_CAN).**

I. 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 quantity, all merits most valued among the industrial community. Thanks to its non-destructive and strictly deterministic packet 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.

With the rapidly growing complexity and level of distribution of the electronic systems aboard modern vehicles [2], some kind of logical structure for the on-board networks comes into the focus. The separation of several vehicle networks can be used to isolate various subsystems, in order to minimize the likelihood of unwanted interactions. For example, it is highly desirable to isolate the critical systems (such as engine control, braking, Advanced Driver Assistance Systems (ADAS), power steering, airbags etc.) from the non-essential parts (such as air conditioning, entertainment system, control of power windows, mirrors and the like), and also one from the other (to a certain degree). This need will be pronounced with the upcoming advent of drive-by-wire vehicles and Inter-vehicle Communications (IVC) [3].

M. Sojka, P. Píša, O. Špinka and Z. Hanzálek are with the Department of Control Engineering, Faculty of Electrical Engineering, Czech Technical University, Karlovo náměstí 13, 121 35 Prague 2, Czech Republic. e-mail: sojkam1@fel.cvut.cz, pisa@cmp.felk.cvut.cz, spinkao@fel.cvut.cz, hanzalek@fel.cvut.cz

O. Hartkopp is with Volkswagen Group Research, Brieffach 1777, 38436 Wolfsburg, Germany. e-mail: oliver.hartkopp@volkswagen.de

Manuscript received December 31, 2010; revised January 31, 2010.

On the other hand, it is still necessary to maintain some level of communication among various subsystems. For instance, a vehicle diagnosis system must be able to extract data from all other units, but certainly should not be connected to every vehicle bus directly. This is important not only to prevent any unwanted interactions among the electronic systems themselves, but - for example - also to physically prevent maintenance personnel from a direct access to critical vehicle buses through the diagnosis interface.

Therefore, network segmentation [4] was introduced in order to cope with these issues. There are also other reasons for that; for example, segmentation allows to connect networks using different communicating protocols and/or physical layers through a common gateway. It is not uncommon that different buses are used for different purposes in a modern car - for example, critical ECUs (Electronic Control Units) are usually connected via CAN, while low bandwidth body electronics like power windows often runs on LIN (Local Interconnect Network), for cost reasons. Also, Internet protocols will undoubtedly find their way into passenger cars in the near future, allowing for direct audio and video streams into the entertainment system [5] [6], but also for constant monitoring of the car's health. For this purpose, an Ethernet-to-CAN gateway would be required.

Another important reason for a separate vehicle network is rapid prototyping and hardware reusability [7] [8]; it is convenient if one can quickly connect existing subsystems (possibly from different vendors), despite the fact that they were never designed to work together and possibly use incompatible communication protocols [9]. For that purpose, a CAN-to-CAN gateway is needed, capable of simple data conversions; for example, it might alter the packet IDs (CAN Identifier), do some data processing in the CAN data payload (like unit conversions), recalculate the CRC (because the IDs and/or data were altered) and so on. Obviously, such gateway must satisfy very strict real-time requirements, especially if it connects critical control systems.

For this and other purposes, a universal Linux-based gateway was developed in the course of the SocketCAN project [10] by Volkswagen Group Research in Wolfsburg. This gateway is designed for CAN-to-CAN and future Ethernet-to-CAN processing and allows packet filtering, data manipulation and checksum calculation for routed CAN data frames. The gateway is not only open-source under the GNU-GPL license; based on the existing PF_CAN infrastructure, it can be fully integrated into the Linux kernel, making it easily accessible for CAN developers. Since the gateway is designed for use in real-time systems, it had to undergo a set of comprehensive tests, focused on measuring packet latencies under various conditions. These extensive tests were recently performed by the Department of Control Engineering at the Czech Technical University in Prague, on behalf and in collaboration with the Volkswagen Group Research.

This paper briefly presents the gateway itself, but is primarily focused on the tests and testing methodology for precise packet latency measurements (which was inspired by [11] [12] [13] [14]). Our testing methods and results are presented in detail and moreover, the complete data set, consisting of gigabytes of data and more than 600 graphs, as well as the source codes of our testing tools, are available for download in our public repositories [15] [16]. This allows other researches 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 also for the Linux networking system in general, which basically covers everything from CAN to Ethernet, Bluetooth, Zigbee and other networks.

The paper is organized as follows: the next section introduces the gateway, while section III describes the methodology of the latency measurement tests. Section IV extensively summarizes the results, followed by conclusion.

II. Linux-Based Gateway

The CAN-to-CAN gateway, presented in this paper, is a software component using the CAN protocol family network infrastructure of the Linux kernel. The Linux CAN subsystem was contributed by the Volkswagen Group Research under GNU-GPL license in 2006 and consists of a new network protocol family (PF_CAN) and the idea to implement CAN hardware drivers using the network driver model known from ethernet devices. The protocol layer part and the network drivers have been finalized within the open-source project 'SocketCAN' [10] until it became part of the Linux mainline kernel in 2008.

The SocketCAN approach is extensively using the Linux networking infrastructure, which is the main factor determining its inherent assets and shortcomings. The main asset of this approach is that SocketCAN is simple to use, making CAN communication no different to normal TCP/IP communication from the Application Interface (API) point of view. The very same functions are being used to handle the device and datapackets, the only notable difference being the data-holding structure, which includes the header of the CAN datapacket and is restricted to eight bytes of payload data. This approach also greatly simplifies routing packets among multiple networks, for example between CAN and Ethernet, which is a highly desirable asset (as explained in section I).

The drawback of this architecture is that it might be hard to achieve desired timing properties and reliability, especially under heavy bus load. SocketCAN is using the same data handling infrastructure as Ethernet and other networking devices (Bluetooth, Zigbee etc.), which makes it prone to negative influence by other communication (as was shown in [17]).

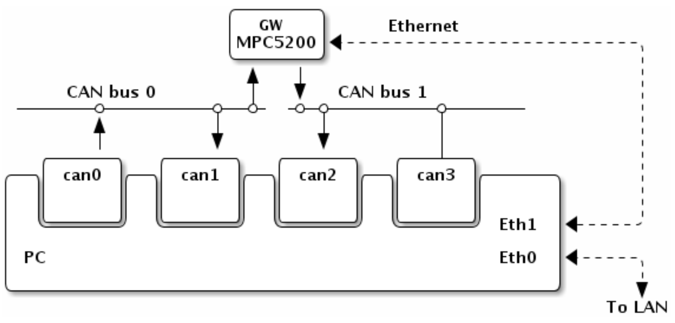


Fig. 1. Testbed configuration

The CAN gateway itself integrates seemless into the CAN subsystem code, and therefore runs in kernel-space. A CAN gateway generally can be run in user-space also, however, that approach has significant drawbacks (see section IV-B). The gateway allows point-to-point or multi-hop routing, packet filtering and modifying, and also enables packet routing among multiple network protocols. However, the last feature is experimental and requires further development. The regarded CAN gateway component is widely configurable via the common netlink protocol.

For the following measurements and tests a MPC5200 SoC (PowerPC) platform runnig several different Linux kernels (see section III-D) was used as gateway hardware.

III. Testing Methodology

A. The Testbed

The testbed, used for packet latency measurements, consists of a PC computer (a Pentium 4 at 2.4 GHz with hyper-threading, 2 GB RAM), equipped with Kvaser PCI quad-CAN SJA1000-based adapter, connected to both CAN interfaces of the gateway. The PC and the gateway are also connected via Ethernet (using a dedicated adapter on the PC), while the other Ethernet adapter of the PC is connected to a local network. The traffic on the second adapter could slightly influence measurement accuracy, therefore it is suspended while running the tests. The Ethernet connection between the PC and the gateway serves multiple purposes; it is used to boot the gateway (using TFTP and NFS services), configure it and download firmware updates (using SSH), and also to generate traffic during the tests. The connection is direct, using a crossed cable, to ensure that there are no switches or other networking elements in the way, which could otherwise influence the tests. The complete setup is shown in Figure 1.

The software configuration is kept as simple as possible, in order to expel any software tasks that could have any influence on the tests. On the gateway, only a Linux kernel and Dropbear SSH server run (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 (as will be explained in sections III-B and III-C) are assigned the highest real-time priority and their memory spaces are locked, in order to prevent swapping. SocketCAN was used on both the gateway and the PC as the CAN driver.

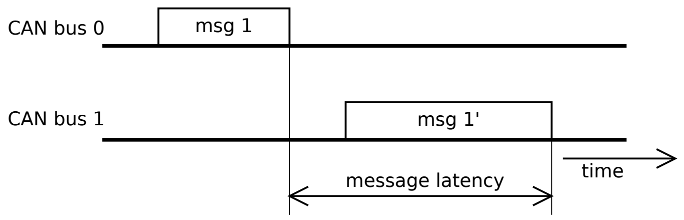


Fig. 2. Definition of packet latency

B. Measurement Methodology

CAN traffic is produced by the PC, using its *can0* interface. As can be seen in Figure 1, this interface is directly routed to the *can1* interface of the PC, as well as to the gateway. The *can1* interface is configured as passive and serves to determine exact time when each packet was actually transmitted. This is necessary in order to exclude various delays between the packet being stacked into the transmit queue and its actual transmission, which may be introduced on the PC side. When a packet is received by the *can1* interface, a high-priority interrupt is initiated and the packet is assigned a timestamp. This is done directly in the interrupt handler by the Linux kernel, hence there are no additional latencies associated with this process and the timestamp is very accurate.

The packets coming from the gateway are received by the *can2* interface of the PC. Incoming packets are assigned timestamps using the same mechanism described in previous article. The packet latency can then be determined as the difference between timestamps of the *can1* and *can2* interfaces for each packet. It is important to note that both timestamps are associated using the same clock, which ensures maximal accuracy, which could not be achieved if different clocks were used.

The definition of the packet latency is shown in Figure 2. The figure emphasizes the fact that the transmitted and received frames must not necessarily be the same, since the frame could have been manipulated by the gateway, as was explained in sections I and II. Therefore, it is necessary to exactly determine which received packet corresponds to which transmitted one, and this mechanism must be also able to cope with possible packet loss. For this purpose, the first two bytes of data are used to associate a unique number to each packet. This number serves as an index for a lookup table, in which the timestamps are stored. This also allows to easily detect packet losses (when the corresponding field in the lookup table contains just one timestamp after a certain timeout, which is set to 1 s by default). This mechanism is also immune to possible changes in the reception order, which might be introduced by the gateway (for example, when some packets are manipulated and others are not).

C. Traffic and Load Generators

Different traffic patterns were used in order to gain the broadest pool of comprehensive data. Let us look at each test case in detail.

Flood traffic is a test case focused on full bus utilization. The messages were sent as fast as possible, in order to check whether the gateway drops any messages under full load. In this case, the TX queue on the PC is full most of the time. To achieve true burst flow of the packets without any delays, it was necessary to increase the length of the Linux transmission queue from 10 to 200 messages. Without this, the `write()` system call returned `-ENOBUS` even though the `poll()` system call signalized that there was free space in the buffers, and therefore it was not possible to maintain burst packet flow.

In the test case called **50% bus load**, the packets were sent periodically, with period equal to the message transmission time multiplied by two. This case was meant to simulate some reasonable real-life bus load.

One message at a time is a case focused on precisely measuring packet latency under various controlled conditions in the gateway. In this mode the next packet was sent only after reception of the previous one. For the case of packet loss, a built-in timeout of one second was implemented.

The IDs and data lengths of the messages are configurable and were altered during the tests, in order to determine their influence (if any) on the packet latencies.

In order to determine the influence of processor load on packet latencies, a simulated computing load was imposed to the gateway. The load was generated using the `hackbench` tool [18].

To determine the influence of Ethernet traffic on packet latencies, the gateway was put under considerable Ethernet load in some tests. Ethernet traffic was generated by the PC using standard `ping` and `floodping` tools. The influence of these traffic generators on precision of the timestamps on the PC side is negligible, since the timestamps are assigned directly within the high-priority receive interrupt handler and no shared queues come into play.

D. SocketCAN and Kernel Versions

The tests were carried out with three different Linux kernels in the gateway. First, version 2.6.33.7 was used. The CAN subsystem mainline code was replaced by revision 1199 from the SocketCAN project SVN repository, the latest available at the moment. The second kernel was 2.6.33.7-rt29 (the latest available version with rt-preempt patchset at the time), along with the same SocketCAN revision. The motivation for this was to use the same kernel and SocketCAN versions, with and without rt-preempt patchset, in order to determine its influence. The last kernel was 2.6.36.2, the latest available at the moment.

The standard CAN gateway code from the SocketCAN SVN repository was slightly modified to allow for routing of a single packet multiple times as described in Section IV-E. Additionally, a small bug that caused occasional crashes with latest kernels was fixed in the gateway. This fix was distributed to the maintainer and co-author Oliver Hartkopp and is already implemented in the later SocketCAN SVN revisions.

All kernels used for testing are available in our git repository [19], in the branches called *cangw-test*, *cangw-test-rt* and *cangw-test-36*.

IV. Results

In this section, the most important results shall be presented and discussed. It should be noted that the results shown here represent only the most relevant ones to support our conclusions; the extensive set of 630 graphs and several gigabytes of data are available in [16].

Measured latencies of individual packets were processed statistically, and histograms (latency profiles) were generated from the data. A latency profile (see, for example, Figure 4) is a sort of backward-cumulative histogram with logarithmic vertical axis. The advantage of using latency profiles is that the worst-case behavior (bottom right part of the graph) is "magnified" by the logarithmic scale. 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 $m_1 - m_2$ messages have the latency in the range $(t_1 t_2)$. Additionally, the rightmost point (t_w, m_w) shows that there were m_w messages with the worst-case latency t_w.

Since the range of the packet latencies can significantly vary among the runs of a single test case, performed under different conditions, the horizontal axis (time) is scaled logarithmically too (in most cases). This allows to use the same scale for all graphs corresponding to a single test, making them easily comparable.

To evaluate the precision of the time measurements, we measured the time between sending the frame from the `can0` interface and its reception on `can1`. As these two interfaces are connected to the same bus, the delay only includes message transmission time and operating system overhead. It is important to determine this overhead and its dependency on the Ethernet traffic, since the same overhead affects the timestamp associated with the packet reception on `can2` interface.

The experiment was performed by measuring the latencies of 10000 consecutive packets and statistically evaluating the results. Same experiment was repeated with traffic generated on the Ethernet interface, in order to determine how much does this this traffic affect the timestamp accuracy.

It was determined that without the Ethernet load, for 99.9% of packets the delay was less than 10 μs. Only 0.1% of packets were received within additional 20 μs.

With the Ethernet load generated and received by the PC, 99.9% of packets fell within 100 μs. Remaining 0.1% of packets were received within 200 μs at most. Since the latency introduced by the gateway is much larger than that (as will be shown by following results), this precision is still well within acceptable limits.

A. Simple Gateway

The first test was performed on a simple gateway, configured to repeat all packets without any filtering or modifications. The block diagram can be seen in Figure 3.

The bus was fully utilized (burst traffic was generated by the PC). The results are shown in Figure 4. The latencies span from 60 μs for packets carrying two-byte data to 110 μs for eight-byte data packets. These results almost exactly match theoretical best times, given by the message transmission times (60 μs and 108 μs respectively, not taking bit-stuffing into account).

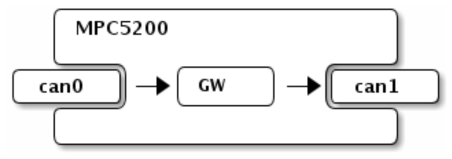


Fig. 3. Simple gateway configuration.

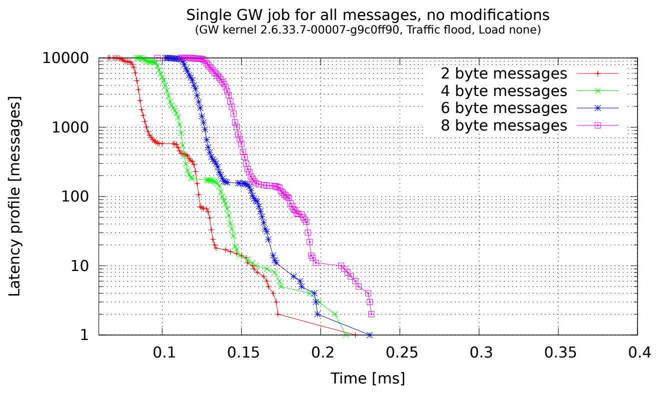


Fig. 4. Latency profile of a simple gateway

No packets were lost in unloaded case. Two packets (out of 10000) were lost under CPU load, while approximately 20% of packets were lost under full Ethernet load. This is caused by sharing RX queues among CAN and Ethernet packets in SocketCAN, as is explained in detail in our previous work [17].

B. User-Space Gateway

The previous "simple gateway" experiment was performed on the gateway running in kernel-space, as it is implemented by default (see section II). For the sake of completeness, this experiment was repeated with the gateway running in the user-space, in order to evaluate the difference.

With one-packet-at-time (the next packet is transmitted only after the reception of the previous one), the latency of the gateway increased by approx. 17 μs (Figure 5 above). In case of flood traffic, the latency of the user-space gateway increased dramatically (Figure 5 below) and packets became to be lost.

Please note that all other experiments were run with the kernel-space gateway.

C. Gateway with Packet Filtering

The most common application of a CAN-to-CAN gateway is packet filtering. In this mode, the gateway retransmits only packets satisfying certain filter specifications. Most commonly, bit masks are used to filter packets according their IDs.

1) Single-ID Filters: In this test, a set of 2048 different filters was constructed. Each filter matched exactly one packet ID, corresponding to the number of the filter. The complete filter set allowed only packets with IDs smaller than 2048 to pass. The filter set was deliberately made much more complex than necessary for that particular task, in order to impose some meaningful packet processing load on the gateway. In this

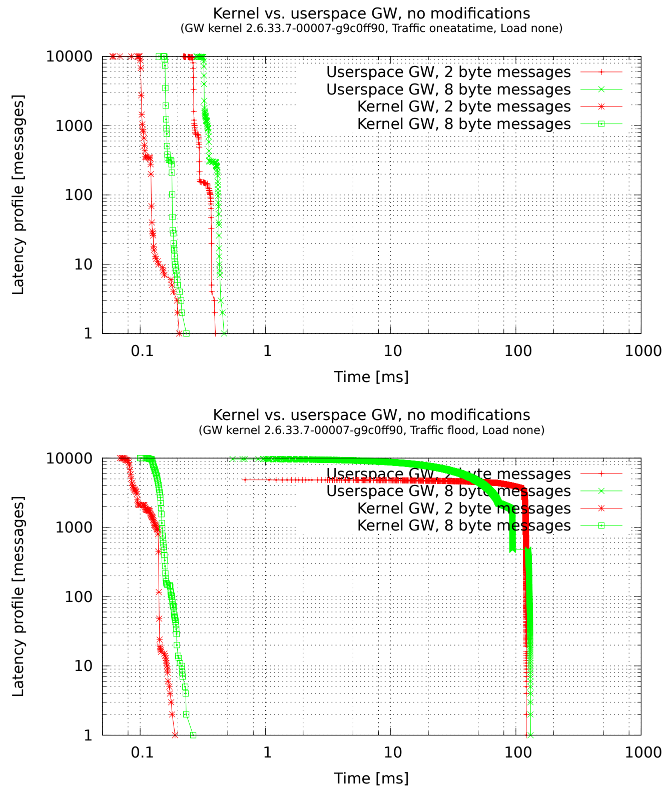


Fig. 5. Kernel-space versus user-space gateway – single message traffic and flood traffic.

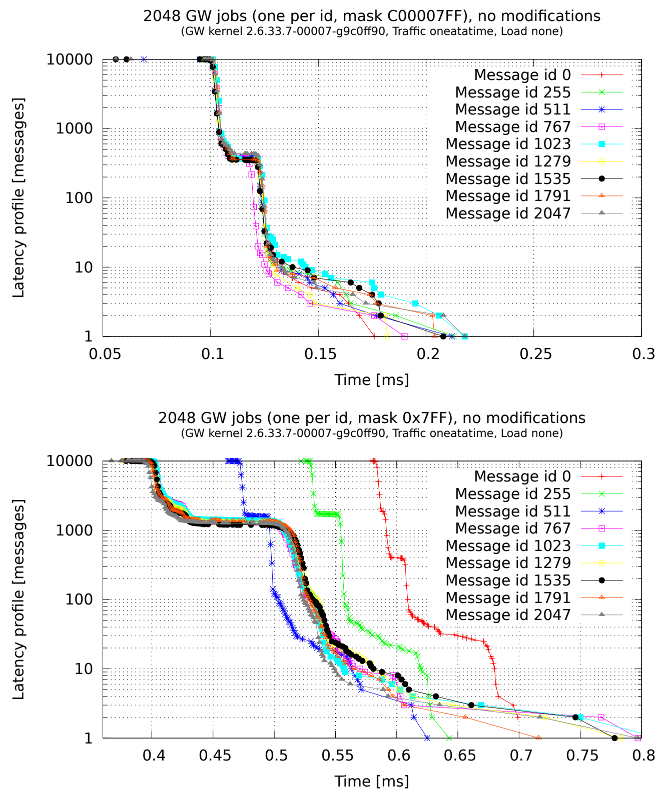


Fig. 6. Gateway with 2048 filters. **Above:** Filters matching only SFF packets **Below:** Filters matching both SFF and EFF packets

case, the filtering algorithm has to search through the filters to determine whether to route a given packet.

Two test cases were set up. In the first one, filters satisfying the mask 0xC00007FF were constructed. These filters match only the Standard Frame Format (SFF) packet IDs. In this case, the SocketCAN keeps the filters in a static pre-allocated lookup table, the key being the message ID. That table has 2048 entries, so the filter can be located very quickly by indexing this table.

The other case is based on filters satisfying the mask 0x000007FF. That mask allows both SFF and EFF (Extended Frame Format) packets to pass. In that case, the filters are stored within a dynamic linked list (because of much greater number of possible packet IDs), which must be traversed linearly in order to find out whether the packet ID matches one of them or not. Obviously, this algorithm imposes much greater processing penalty than the lookup table indexing used in the first case.

The resulting performance is shown in Figure 6. It can be seen that the SFF-only filters had negligible impact and the packet latencies are almost the same as they were in the previous experiment (simple gateway). The second test case tells a different story - Once the filters were stored in a linked list, the performance deteriorated greatly. For example, the best-case latency in this case went down to $400\,\mu s$ (from $100\,\mu s$ in the SFF case).

2) Filters under Heavy Traffic: Since the time needed to process all 2048 filters in the list is higher than the transmission time of a packet, the gateway must drop some

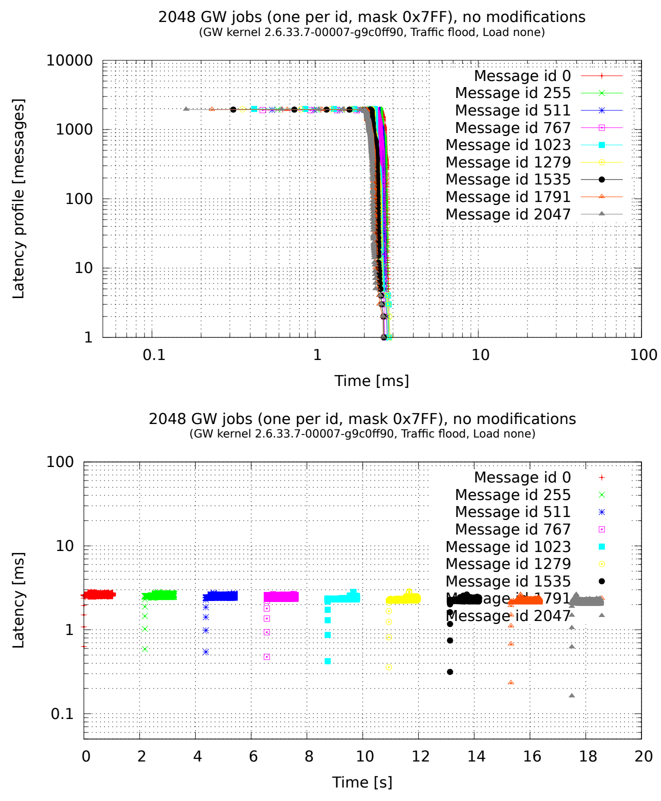


Fig. 7. Too many filters cause the packets to be dropped under heavy traffic

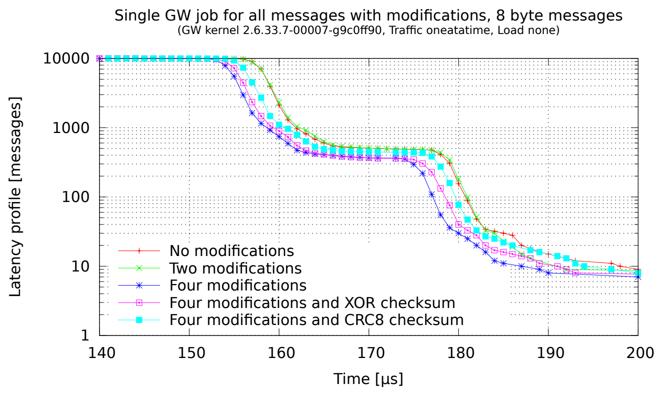


Fig. 8. Message modification comparison.

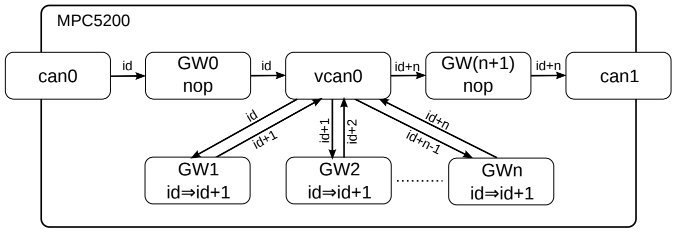


Fig. 9. Multi-hop gateway with a single virtual CAN interface (vcan0).

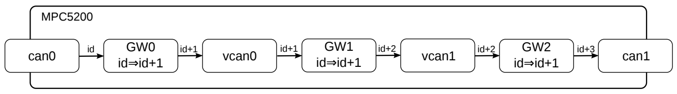


Fig. 10. Multi-hop gateway with two virtual CAN interfaces

of them under heavy load. In case of 100% bus utilization the drop rate was near to 80% (see Figure 7). In the time chart shown in Figure 7 below, the effect of the RX buffer is very obvious. MPC5200 has a buffer for four packests. The four isolated points at the beginning of each chunk correspond to RX buffer filling. Once the buffer is filled, packets start to be dropped.

D. Packet-Modifying Gateway

As was explained in section I, the gateway can also modify packets while routing them to the other CAN interface. It was determined that simple data and ID modifications have almost no performance impact, as can be seen in Figure 8.

Interestingly enough, the packet latencies in this case even tend to be slightly shorter than they were in the "simple gateway" experiment. This may be probably attributed to the way the C compiler optimizes the code. Packet modifications are implemented within a loop, which is present in both the "simple" and packet-modifying versions of the gateway. The only difference is that in the "simple" version, the condition within the loop always returns false, while in the packet-modifying version it returns true, save for the last iteration. Due to the "naive" branch prediction, the code is most likely optimized for the "true" condition, making it run slightly faster in that case, due to undisturbed instruction pipeline.

However, the authors must admit that this little mystery was not yet investigated in detail, since the difference is negligible and almost within the level of noise (although persistently present when the experiment was repeated over and over again). Therefore, it is not an issue and not on the top of our priority list, although it will be thoroughly inspected in the future.

E. Multi-Hop Gateway

The gateway can also be used as a multi-hop router, connecting multiple buses. Due to the hardware limitations of the MPC5200 platform, which has only two CAN interfaces, additional virtual CAN interfaces had to be used in this experiment.

Two test cases were prepared for this experiment. In the first one, a single virtual interface (denoted `vcan0`) was added to the existing two (denoted `can0` and `can1`). Each packet, received by the `can0` interface, was rerouted to the virtual interface `vcan0`. This interface was configured as a loopback, sending each packet back again (with simulated transmission delays). The packet ID was then incremented and the packet was sent back to `vcan0` to emulate another pass. This was repeated several times, and afterwards the packet was routed to the physical interface `can1` and sent back to the PC. This test case (Figure 9) was designed to evaluate the latencies introduced by the virtual interface. The packet was run through that interface several times in order to compute an "average latency", since each run may have been affected by some non-deterministic interruptions.

In the other test case, two virtual interfaces (`vcan0` and `vcan1`) were serially connected between the physical `can0` and `can1` interfaces (Figure 10). Each incoming packet was routed through that chain, its ID being increased when passing both `vcan0` and `vcan1` (in order to make the results comparable with the previous test case).

In both cases only the "fast" SFF-exclusive filters (mask 0xC00007FF) were used (as explained in section IV-C1).

The results are presented in Figure 11. In order to make them comparable, the first graph shows a test case when each packet was run two times through the single `vcan0` interface. It can be seen that the additional delay in the first test case was around $20 - 25\,\mu s$, while in the second case it was slightly longer, about $30\,\mu s$. These numbers give us an estimate of approx. $11\,\mu s$ per hop in the first case and $14\,\mu s$ per hop in the second. This slight difference was expected and can be easily explained by greater memory demands of the two interfaces, resulting in increased cache misses.

When both test cases were run with flood traffic, packet losses started to occur with increasing number of hops. In the case with a single virtual interface, packet losses started at eight hops, while in the second case with multiple virtual interfaces, it was already at six hops. This is consistent with the slightly increased overhead represented by the two separate virtual interfaces.

F. Gateway Load

All experiments shown so far were repeated under different CPU and Ethernet loads, in order to investigate their influence

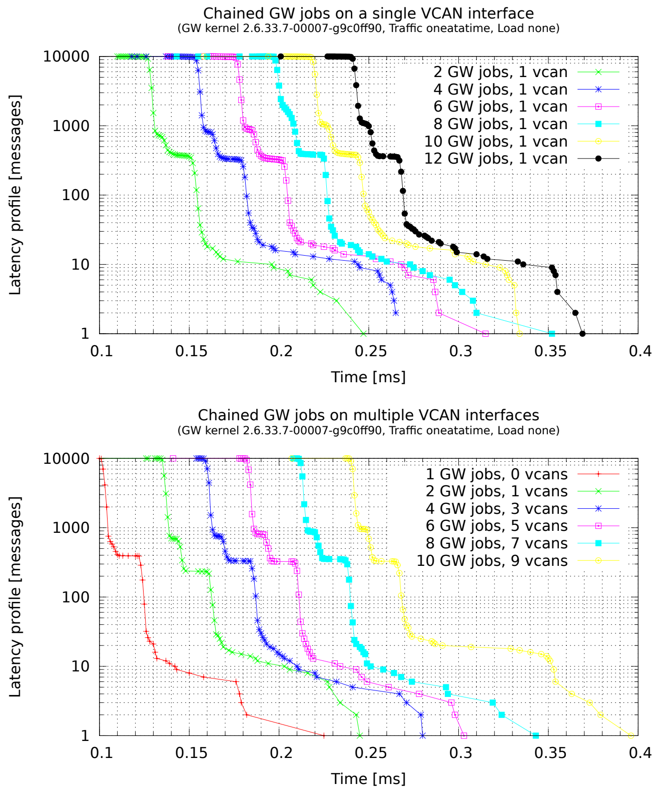


Fig. 11. Multi-Hop gateways. **Above:** Single virtual interface **Below:** two virtual interfaces

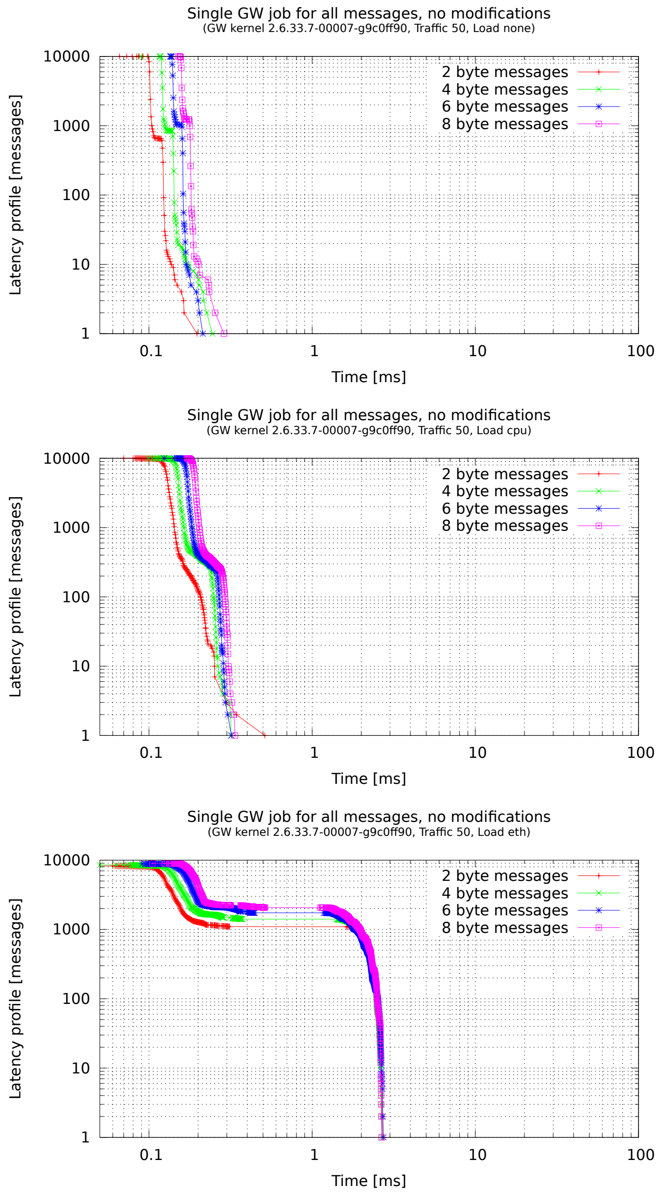


Fig. 12. Performance under different loads – no load, CPU load and Ethernet load

on packet latencies. Figure 12 shows the results for the "simple gateway" experiment, shown in section IV-A, under different loads. The CPU load adds approximately $100\,\mu s$ to the processing time. The influence of the Ethernet load is much worse; the worst-case latency increased ten times from $0.3\,ms$ to almost $3\,ms$. From the bottom graph, it seems likely that processing of a single 60 kB ping packet takes approximately $2.5\,ms$ and this time is simply added to the latency, experienced by the CAN message. The reason for this is that the Linux kernel processes all incoming packets in the same soft-IRQ in a non-preemptive manner, as is explained in [17].

The differences observed in other experiments run under different loads were similar.

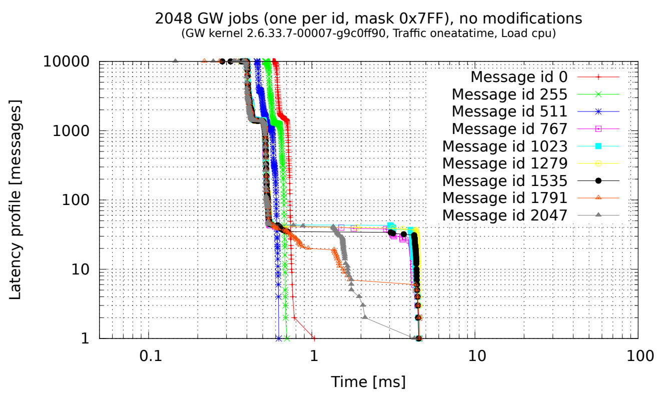
G. Differences between Linux Kernels

All tests were also repeated with three different kernel versions (2.6.33.7, 2.6.33.7-rt and 2.6.36.2), in order to determine especially the difference between standard and rt kernels.

The difference between 2.6.33 and 2.6.36 was very small - in all experiments the latencies of the 2.6.36 kernel were approximately $10\,\mu s$ higher, which can be attributed to the evolution of the Linux networking stack, aimed to optimize overall data throughput rather than latencies of individual packets.

Much more interesting is the comparison between standard and rt 2.6.33 kernels. Oddly enough, the rt kernel actually exhibited worse behavior than the standard one, when the gateway was running in kernel-space. In this case, the packet latencies in the rt kernel were about $70\,\mu s$ longer compared to the non-rt version. This could be attributed to somewhat greater overhead of the rt kernel (which, on the other hand, leads to greater predictability and stability of the latencies), but the worst-case behavior of the rt kernel shows latencies as long as 50 ms (as can be seen in Figure 13). This indicates that there is probably something seriously wrong with this particular version of the rt patch, since our previous experience and results of our previous work [17] show visible improvement of the rt version over the standard one in case of older kernels.

The 2.6.33-rt kernel showed significant improvement when the gateway run in user-space. In that case (refer to Figure 14) the worst-case latency experienced by the rt kernel was about 2 ms, while the non-rt version exhibited delays as long as 130 ms (Figure 5). Interestingly, the performance of the



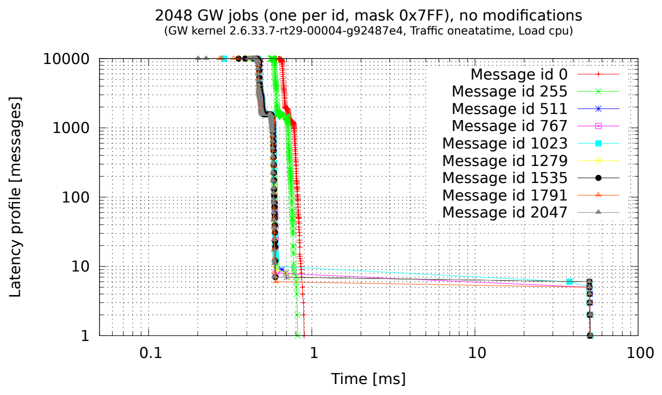


Fig. 13. Standard (above) and rt (below) 2.6.33 kernels under CPU load

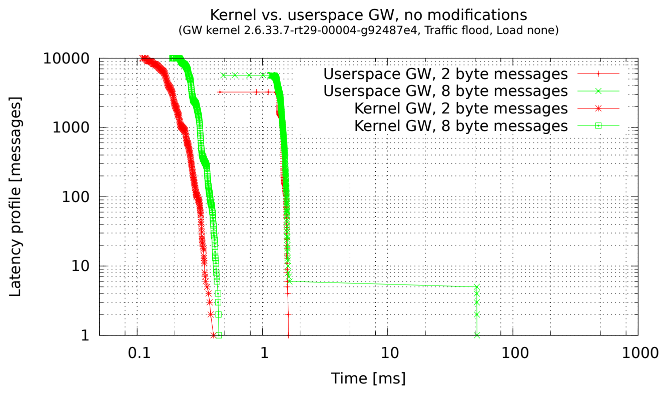


Fig. 14. User-space gateway under 2.6.33.7-rt kernel.

gateway in user-space under the rt kernel was better than in kernel-space, which, again, indicates some serious problem with this particular version of the rt patch.

V. Conclusion

This paper presented the timing analysis of a Linux-based CAN-to-CAN gateway and studied influence of various factors (like CPU bus load, kernel versions etc.) on packet 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.

On the other hand, it must be noted that especially excessive Ethernet traffic and improperly constructed packet filters can lead to significant performance penalties and possible packet 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 [17]. On the other hand, it is a standard and easy-to-use solution, integrated in Linux kernel mainline, and therefore forms 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 and CPU load is not excessive.

Our future work will focus primarily on Ethernet-to-CAN routing, since such system will be required in the near future, with the advent of Ethernet and Internet protocols in cars. This implies that special attention must be given to the solution of current issues affecting CAN packet delays under heavy Ethernet load.

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

[3] L. Wischhof, A. Ebner, and H. Rohling, "Information dissemination in self-organizing intervehicle networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 6(1), pp. 90–101, 2005.

[4] M.-Y. Lee, S.-M. Chung, and H.-W. Jin, "Automotive network gateway to control electronic units through MOST network," in *IEEE International Conference on Consumer Electronics (ICCE), Los Angeles, USA*, 2010, pp. 309–310.

[5] L. Guglielmetti, "Standardizing automotive multimedia interfaces," *IEEE Transactions on Multimedia*, vol. 10(2), pp. 76–78, 2003.

[6] K. Sato, T. Koita, and S. McCormick, "Design and implementation of a vehicle interface protocol using an ieee 1394 network," *The EUROMICRO Journal of Systems Architecture*, vol. 54(10), 2008.

[7] Y. Yacoub and A. Chevalier, "Rapid Prototyping with the Controller Area Network (CAN)," in *SAE World Congress, Detroit, USA*, 2001.

[8] J. Stroop and R. Stolpe, "Prototyping of automotive control systems in a time-triggered environment using flexray," in *IEEE International Symposium on Intelligent Control, Munich, Germany*, 2006, pp. 2332–2337.

[9] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli, "Automotive virtual integration platforms: why's, what's, and how's," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, Freiburg, Germany*, 2002, pp. 370–378.

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

[11] H. Zeng, M. D. Natale, P. Giusto, and A. Sangiovanni-Vincentelli, "Statistical Analysis of Controller Area Network Message Response Times," in *IEEE Symposium on Industrial Embedded Systems*. Lausanne, Switzerland: Ecole Polytechnique Federale de Lausanne, 2009.

[12] L. Almeida, J. Fonseca, and P. Fonseca, "A flexible time triggered communication system based on the controller area network," in *Proceedings of the FeT'99Fieldbus Systems and their Applications Conference*, Germany, 1999.

[13] J. L. Campos, J. J. Gutierrez, and M. G. Harbour, "CAN-RT-TOP: Real-Time Task-Oriented Protocol over CAN for Analyzable Distributed Applications," in *In 3rd International Workshop on Real-Time Networks (formerly RTLIA)*, Catania, Sicily (Italy), 2004.

[14] T. Nolte, M. Nolin, and H. Hansson, “Real-time server-based communication for CAN,” *IEEE Transactions on Industrial Informatics*, vol. 1(3), pp. 192–201, 2005.

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

[16] M. Sojka and P. Píša, “Netlink-based CAN-to-CAN gateway timing test results,” 2010. [Online]. Available: http://rtime.felk.cvut.cz/can/benchmark/2/

[17] 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), Trento, Italy*, 2010.

[18] “The hackbench tool project website,” 2009. [Online]. Available: http://devresources.linux-foundation.org/craiger/hackbench

[19] M. Sojka and P. Píša, “Linux kernel git repository for SocketCAN tests,” 2010. [Online]. Available: http://rtime.felk.cvut.cz/gitweb/shark/linux.git