Managing Concurrent Queues for Efficient In-Vehicle Gateways

Miltos D. Grammatikakis*, Stelios Ninidakis, George Kornaros, Dimitris Bakoyiannis, Nikos Mouzakitis, and Alexis Staridas

Electrical and Computer Engineering, Hellenic Mediterranean University, Heraklion, Greece; Email: sninidakis@cs.hmu.gr (S.N.), kornaros@cs.hmu.gr(G.K.), d.bakoyiannis@cs.hmu.gr (D.B.), nmouzakitis@cs.hmu.gr (N.M.), astaridas@cs.hmu.gr (A.S.)
*Correspondence: mdgramma@cs.hmu.gr (M.D.G.)

***Abstract*—In modern vehicles, electrical control units communicate over multiple in-vehicle networks via domain- or zone-oriented gateway architectures. This work examines efficient frame transfer across incoming and outgoing Controller Area Network (CAN) interfaces at a gateway. In our embedded platform prototype, each CAN interface of a CAN-to-CAN gateway is controlled independently by a corresponding Portable Operating System Interface (POSIX) thread. Inter-thread communication and synchronization are implemented using shared data structures, either lock-free concurrent queues or traditional lock-based circular queues that embody single-producer single-consumer (SPSC) principles. Our experimental framework provides a realistic open automotive platform that integrates multiple CAN interfaces. An Odroid XU3 device acts as a gateway, providing two Universal Asynchronous Receiver-Transmitter (UART) to CAN interfaces that lead to Raspberry Pi 3B nodes (connected to CAN via Serial Peripheral Interface (SPI)). To increase communication reliability and performance by minimizing frame loss and improving the egress CAN frame rate, low-level parameters at the gateway are optimized, especially the inter-character delay of the UART-to-CAN interfaces. Our experiments show that the frame waiting time in the queue is shorter for the concurrent queue, while relaying messages is equally fast, reaching a maximum output rate of ~380 frames/s for large enqueue rates near saturation. Power consumption is almost four times higher for the concurrent queue implementation.**

***Keywords*—automotive gateway, CAN bus, concurrent queue, frame rate, in-vehicle network, producer-consumer, reliability**

I. Introduction

Newly emerging automotive architectures must support powerful ECUs and high-speed interfaces to handle complex tasks with an increasing communication bandwidth, but limited power consumption. Due to the increasing demand for supporting new functionalities, automakers connect multiple CAN buses via a central gateway. Automotive gateways aim to transfer data reliably and efficiently from one communication bus to another, possibly filtering (intercepting or altering messages) for security.

While in the old days, CAN-based communication was through an independent Electronic Control Unit (ECU), often called a Body Control Module (BCM), currently, in most vehicles, a Domain Centralized Electrical/Electronic (E/E) Architecture (central gateway) is used. This gateway allows for in-vehicle communication across many domain controllers, such as powertrain, transmission, chassis, interior, infotainment, telematics, and diagnostics.

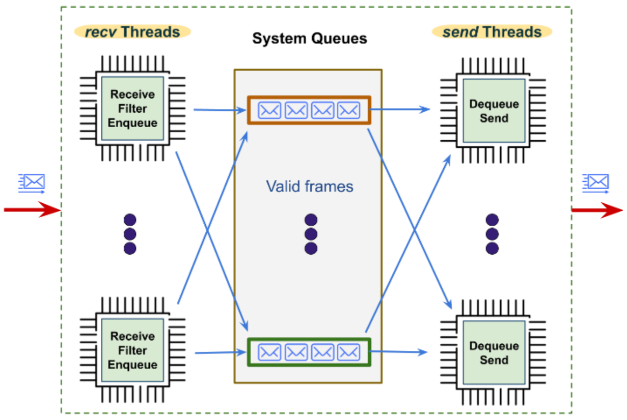
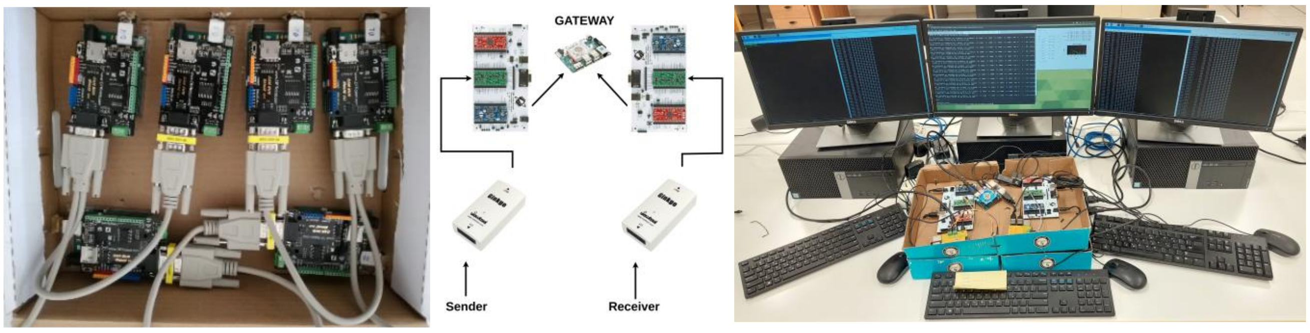


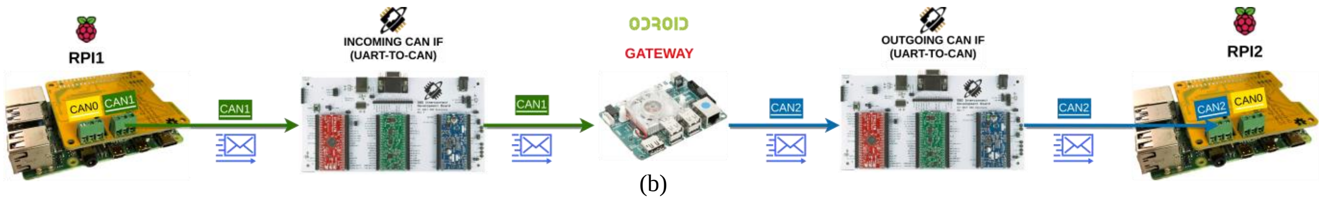
Figure 1. Inter-thread communication in the gateway.

Recently proposed high-end automotive solutions supported by Bosch require a centralized E/E architecture [1]. A central gateway (many-core processor) is connected via On-Board Diagnostics (OBD) with many zone controllers via 1000BASE-T1 Ethernet, enabling extensive telematics services and smart Cloud-based solutions [2-4]. Zone controllers act as smaller in-vehicle gateways to distribute data based on the physical location of ECUs that control vehicle sensors and actuators. This high-end approach, although not fully implemented, reduces wiring costs, and enables safer direct communication from high-bandwidth sensors and actuators to central components. In addition, the proposed architecture could use the central gateway to perform vehicle functions in the cloud ecosystem, including vehicle maintenance, firmware and software updates, security control, and third-party infotainment applications.

Manuscript received August 15, 2022; revised September 20, 2022; accepted October 30, 2022.



(a)



(b)

Figure 2. a) Pictures of two initial platform incarnations used for emulation and debugging (top left/center), and the final open platform (top right), and b) schematic details of the final distributed embedded platform for experimenting with gateway architectures. ECU components are based on Raspberry Pi with Canberry SPI-to-CAN interfaces. The gateway is an Odroid XU3 with UART-to-CAN interfaces (via OBD development kits).

In both current and future automotive solutions discussed above, the efficiency of in-vehicle gateways is important. Our study focuses on an in-vehicle gateway architecture that supports efficient CAN message exchanges across multiple interfaces. CAN is the most popular in-vehicle network [5]. It is a robust serial communication bus designed by Bosch in 1980 for applications that perform in harsh environments. Since 1993, CAN is ISO 11898 standard. At the physical layer, it enables a two-wire analog data connection (CAN high/low signals), reducing cable wiring and weight, and increasing reliability. ECUs interact with the physical layer via a transceiver (e.g., Microchip MCP2551 or high-performance Philips TJA1050). CAN physical layer operates at data rates from 125 Kbps to 1Mbps and covers a maximum distance of 40m (120m with high-performance transceivers). The data link layer, implemented by the CAN controller, e.g., Microchip MCP2515, is responsible for sending/receiving packets (known as frames). CAN implements a Carrier Sense Multiple Access with Collision Avoidance policy (CSMA/CA) using ID priority. This means that although all connected nodes have the right to access the bus, only the one with the highest priority (lowest ID) is authorized to broadcast. The remaining nodes switch to the receiving state to listen to the broadcast message.

As shown in Fig. 1, the proposed software architecture uses a thread to control independently each CAN interface, i.e., each recv thread receives and filters frames arriving from an incoming interface, before enqueuing valid frames into a Queue, while each send thread dequeues frames from the Queue and sends them via an outgoing interface.

Lock-free data structures can reduce synchronization overheads of shared memory accesses compared to classical implementations synchronized using mutual exclusion constructs (locks, semaphores, or barriers). Hence, inter-thread communication is implemented using parallel shared memory data structures, namely, lock-free concurrent queues using Cameron implementation [6] (called CQ), or traditional lock-based circular queues. In the latter case, two different implementations are considered that embody single-producer single-consumer principles: a) Dijkstra's SPSC implementation [7, 8] (called List), and b) Lamport [9].

We claim that these data structures are important for designing robust, high-performance, energy-efficient automotive gateways. However, there is limited prior research that relates to automotive systems or in-vehicle gateways. More specifically, the performance of blocking and non-blocking SPSC queue implementations has been examined only with standalone test benches involving multiple threads, but rarely on scientific or industrial problems, e.g., [10, 11].

In addition, with the development of intelligent connected vehicles, recent research on efficient in-vehicle CAN-to-CAN gateways focuses mainly on automotive security. Thus, for example, intrusion detection/protection systems, including authentication and encryption methods, and firewall protection (with packet filtering, DoS protection, and access control) are examined mostly using spreadsheets, and sometimes using automotive subsystems or emulation systems using embedded platforms [12-15].

However, only rarely CAN-to-CAN gateway performance or energy is examined. In [16], a CAN-to-CAN bridge is proposed that provides selective frame retransmission. In [17], data transfer policies across CAN network subsystems are examined by running a benchmark from the Society of Automotive Engineers. In [18], dimensioning of a CAN-to-CAN gateway is considered via analytical queueing models and simulation, i.e., assuming two interconnected CAN systems with different transfer rates, the effect of reducing buffer capacities to frame loss is modeled. Finally, in [19], the authors theoretically calculate worst-case latency analysis for frame arrival in a CAN-CAN gateway that uses real-time scheduling.

The paper has the following structure. Section II focuses on explaining the gateway implementation using a realistic open embedded platform prototype. Section III concentrates on the experimental framework, including the use case, platform configuration, detection metrics, and results. Finally, Section IV provides conclusions and discusses future work.

II. Exprerimental Framework

A. *Automotive Platform - Hardware*

To evaluate the gateway architecture with different queue data structures, an open, distributed embedded platform prototype that integrates multiple CAN networks is implemented. The platform went through several phases of integration and testing while examining different CAN node architectures, before reaching its final configuration.

The top left and center graphs of Fig. 2(a) show preliminary platform designs that consist of multiple interconnected Arduino AVR boards (with DFRobot CAN shield), and Linux single board computer (Odroid XU3) connected to CAN via Viewtool's Gingko UART-to-CAN sniffer/adapter interface. These initial platform incarnations were used for concept validation and debugging in early validation tests; however, these systems are not able to support a gateway architecture with multiple CAN interfaces.

The top right of Fig. 2 (a-b), show our final distributed embedded platform emulating a simplified automotive ecosystem on which gateway performance will be examined. Although our proof-of-concept platform instance consists of two CAN networks and three CAN nodes (including the gateway), it can be extended to include 4 CAN networks (using IndustrialBerry's Canberry Dual v2.1 shield) and tens of CAN nodes. In our final platform, two Raspberry Pi 3B nodes (RPI1 sender, and RPI2 receiver), running 2019-04-08-Raspbian (Linux kernel 4.9 with preempt_rt patch), are used as end nodes.

An Odroid XU3 single-board computer serves as the gateway. The ARMv7 board has a 2GB DDR3 and uses the Samsung Exynos 5422 chipset with Big-Little CPU architecture (Cortex A15 quadcore, and low-power Cortex A7 quadcore). It runs Ubuntu 18.04 LTS (Linux kernel 4.14 with preempt_rt patch). The preempt patch targets embedded systems with latency requirements in the microsecond to the millisecond time range. The board connects to the Raspberry end nodes (sender and receiver) using two incoming/outgoing UART-to-CAN interfaces provided by two Scantool OBD development kits. Each kit enables a one-way CAN interface (incoming or outgoing). It consists of three removable modules (Power, CAN Transceiver, and STN-based CAN Controller) and multiple breakout headers, configuration jumpers, and test points providing access to DB15, RX/TX, and CAN High/Low signals. The power module offers fast sleep/wake-up mechanisms (max 20ns), important for instant suspension of interface action.

B. *Automotive Platform Ecosystem - Software*

As discussed before, our platform interconnects multiple Raspberry Pi 3 nodes to an Odroid XU3 that serves as a gateway. For the Raspberry PI 3B nodes, our embedded software toolchain uses Linux CAN-utils tools (commands cansend, canplay, candump, cangw, cansniff, etc., see [20]) to send, receive or control packets via the Canberry shield. Several options are also available, e.g., the "candump -t a" flag provides the time of arrival.

As shown in Fig. 3, the gateway protocol stack (~3K lines of C code) first initiates two functions to configure the gateway's incoming and outgoing UART-to-CAN message interfaces (Rx/Tx signals) using STN2120's microcontroller ELM327 AT and ST (teletypewriter (TTY)) commands. These interfaces connect to the end nodes (RPI1 sender on CAN1 and RPI2 receiver on CAN2). Besides the terminal commands tcgetattr and tcsetattr to set options, write is used to a) set the CAN protocol to either RAW CAN (or ISO 15765 OBD) protocol via "STP 31" (resp., "STP 33"), b) to permanently set the serial baud rate to 2Mbp/s via "STSBR 2M" and "STWBR"; this enables a CAN rate of 500K frames/s (identical to that of the Raspberry nodes), and c) set receiver/sender mode, flow control, and block filters (e.g., "ATMA" prepares the CAN interface for receiving frames, "ATSHA is used to change CAN ID, and "ATAL 00" is used to disable long frames).

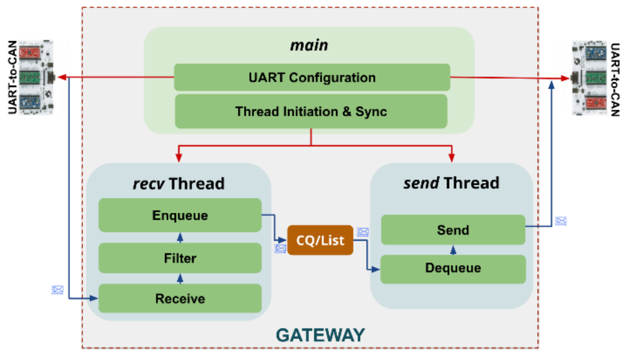


Figure 3. The gateway software stack architecture. The recv and send threads compete for the shared buffer; the architecture is scalable (i.e., extendible) to many (incoming/outgoing) interfaces.

After configuration, the gateway software architecture enables two POSIX threads to manage incoming (and outgoing) CAN interfaces. This code extends an open-source serial TTY terminal [21] with multithreaded code that manages the corresponding serial incoming/outgoing interfaces to receive or send CAN frames. More specifically, the recv thread receives CAN frames from CAN1, then filters and stores valid ones in the CQ (or circular buffer, in the case of List). Similarly, the send thread periodically dequeues frames from the CQ (resp. the List) and transmits them to CAN2.

The threads communicate and synchronize using either a concurrent queue or a classic single-producer single-consumer (bounded buffer) synchronization pattern. In the latter case, the threads utilize two POSIX semaphores (initialized in the main function) to control access to the shared circular queue (bounded buffer) that stores pending frames. In both cases, the operation follows a concurrent producer-consumer pattern, i.e., the recv thread fills, while the send thread concurrently empties the CQ (or List). The queue size is selected during configuration to store many CAN frames (header and data). Proper operation is confirmed by blinking Light-Emitting Diodes (LEDs) and notifications on the console.

```
send thread - index1: 553
    new_msg: 554
    A15: 0.808476W ← Power on Cortex-A15 quadcore
    A7:   0.027540W ← Power on Cortex-A7 quadcore
    T0: 54 C ← Temperature on Cortex-A15 tiles: 0 to 3
    T1: 53 C
    T2: 59 C
    T3: 58 C
```

Figure 4. Visualizing processing at the gateway enables viewing the number of frames (received or sent), power consumption, and thermal data on each Cortex-A15 tile.

As shown in Fig. 4, gateway notifications identify arriving/departing frames, power consumption for Cortex-A15 and Cortex-A7, and thermal zones for all Cortex-A15 tiles; the latter are obtained from integrated INA231 and thermal sensors by accessing registers via Inter-Integrated Circuit bus (I2C).

C. *Traffic Patterns*

The experiments use actual engine traffic. More specifically, the open Korean Hyundai YF Sonata engine trace dataset [22] has been scaled to cover a broad range of injection rates and injected into CAN1 by RPI1 via command canplay (or via Viewtool's Gingko UART-CAN). The exact timing log of more than 988,987 frames makes this emulation very realistic.

The traffic includes time-periodic frame requests, e.g., related to dashboard display metrics (e.g., speed, RPM, temperature, etc.), as well as responses from the engine control unit, and other electrical control units.

D. *Performance Metrics*

Using our experimental framework, the efficiency of frame exchanges at the gateway is examined, when dynamically changing input parameters, such as injection rate (ingress frame rate). Different performance metrics are evaluated, such as frame loss, output rate, queue waiting time, and power consumption metrics related to Cortex-A15 tiles. For a fair comparison, in our tests, all other CAN parameters remain identical, such as cable length, bus load, frame density, and CAN bit rate.

III. Experimental Results

Platform performance is evaluated using different experiments, considering standalone queue performance, reducing frame loss rate using system parameters (i.e., UART-to-CAN inter-character delay), and evaluating egress frame rate, waiting time, and energy consumption. Next, each case study is described separately.

A. *Queue Performance on the Odroid XU3*

In a preliminary study, stand-alone queue performance is evaluated on the Odroid XU3 gateway when two threads (sender and receiver) compete to simultaneously enqueue and dequeue integer data. In this case, CAN interfaces are not in place, the sender and receiver threads are pinned to different Cortex-A15 cores, and the buffer size is set to 8K elements. In addition, failed operations are not counted but immediately retried. Our queue integrates 3 implementations: Lamport, Cameron (called CQ), and List. Another one, BatchQueue [23] generated segmentation faults for large injection rates.

Fig. 5 compares the Average Waiting Time for all three queues for different Injection Rates (all rates are below saturation). Observe that the Lamport queue is better for small rates, CQ is better for medium rates, and List (classic Single Producer – Single Consumer synchronization) is the most efficient for large injection rates.

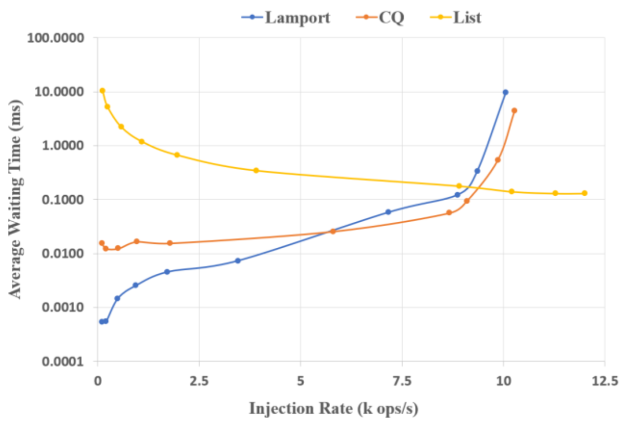


Figure 5. Standalone queue performance at Odroid XU3 vs injection rate.

In general, it is not apparent how the average waiting time behaves in a complex gateway, and how it is affected by network limitations and interface delays. Thus, next in Section III.B, the performance of the two most viable competitors (List and CQ) is examined, when the gateway operates, and CAN frames are exchanged.

B. *Optimizing the Inter-Character Delay of UART Sender*

Each UART controller is independently configurable with many low-level parameters, such as baud rate, data bit length, bit ordering, number of stop bits, and parity. This subsection focuses on reducing frame loss by optimizing the inter-character delay of the UART-to-CAN outgoing gateway interface (leading to the RPI2 end-node). Notice that frame loss leads to at least a one-bit error in the transmitted CAN frame. Frame loss is measured at the end node by sending pre-specified, full-length diagnostic frames (8-byte data including length) that the end node knows how to handle.

To handle reliability issues, a constant inter-character timeout is introduced to effectively limit the time to propagate serial data from the internal buffer to the physical CAN interface.

Since for a speed of 500k frames/s, the serial port is set to 2 Mbps (with 8N1: 8 Data bits, 1 start and 1 stop bit, and no parity), the total number of bits to send a character is 10 bits and the time required to transfer one character is (10 bits / 2 Mbits/s) × 1M (us/s) = 5us. Therefore, inter-character timeout must be larger than 5us.

Fig. 6 examines frame loss at the outgoing link of the Odroid XU3 gateway for a wide range of UART inter-character delays. Both CQ and List are examined, for two relatively large injection rates: 61.5 frames/s and 192 frames/s; the former corresponds to a 16ms delay between consecutive frames, while the latter corresponds to a smaller ~5ms delay.

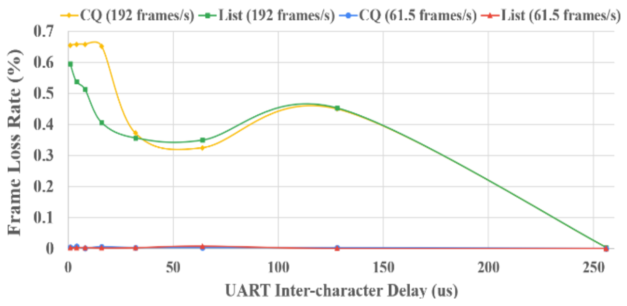


Figure 6. Frame loss rate at the gateway vs UART inter-character delay.

Notice that for the larger injection rate of 192 frames/s (upper two curves for CQ and List), a very high frame loss is detected at RPI2 for small inter-character delays, especially below 5us; however, for large UART inter-character delays above 256us, frame loss decreases abruptly to almost 0 (less than 0.1%). In addition, for the smaller frame injection rate of 61.5 frames/s (lower two curves), frame loss is always near zero (less than 0.1%) for both queues.

Hence, to limit frame loss, the optimal UART inter-character delay is above 5us, as predicted. However, as the UART inter-character delay increases further, real-time functionality in the CAN network is violated, since this extra delay causes the frame waiting time in the queue to vastly increase. For instance, since the total number of frame bits, including up to 24 bits for bit stuffing, is at most 134, a UART inter-character delay of 256 us causes each frame to be transmitted from the queue in ~34ms, a prohibitively large value for real-time in-vehicle communications.

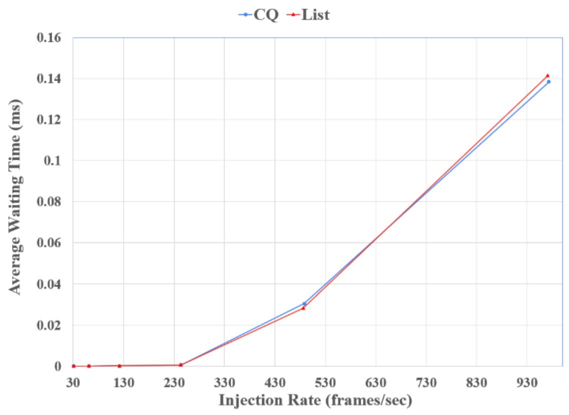


Figure 7. Average waiting time vs injection rate for CQ and List. The inter-character delay is set to 8us.

C. Queue Performance on the Odroid XU3 Gateway

We now focus on examining queue characteristics, by considering queue performance indicators, such as length, and waiting time. While queue length behaves very similarly for list and queue, it is interesting to examine the queue waiting time.

Hence, in this experiment, the average waiting time of CAN frames in the queue is analyzed for medium to large injection rates, from 30 to 973 frames/s. The waiting time is calculated by measuring the elapsed time interval from enqueuing to dequeuing a frame when CAN-to-CAN communication is in place at the gateway. In all experiments, UART inter-character delay is set to 8us.

As shown in Fig. 7, the average waiting time for both CQ and List have similar behavior in low and medium frame injection rates. However, the CQ implementation is only slightly better (~2.2%) for rates higher than 730 frames/second. This extra slack time can be useful in performing slightly more sophisticated frame filtering.

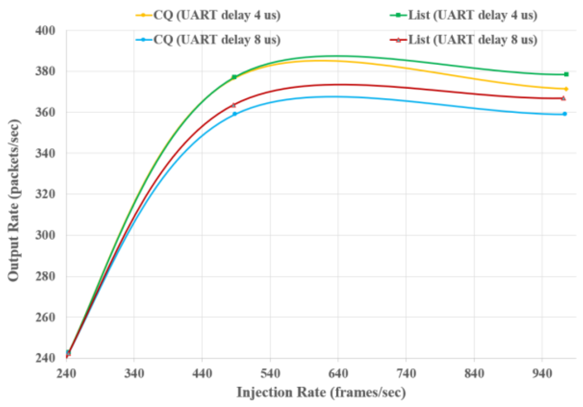


Figure 8. Output rate vs injection rate. For both CQ and List, two different inter-character delays are examined: 4us or 8us.

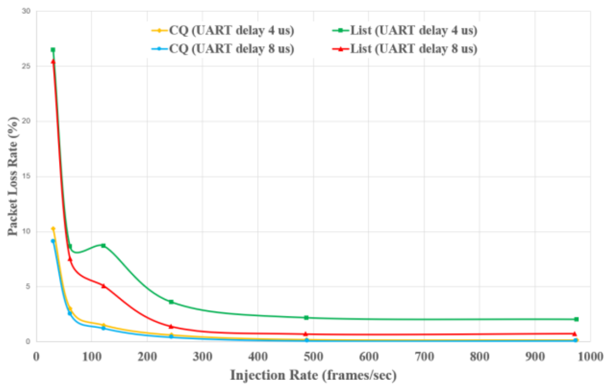


Figure 9. Frame loss vs injection rate. For both CQ and List, two different inter-character delays are examined: 4 us or 8 us.

Fig. 8 shows the output vs. the injection rates (for large injection rates), for two different UART inter-character delay values: 4 us and 8 us. Notice the following.

- Before saturation, both List and CQ perform well, reaching a maximum rate of ~380 frames/s for 4 us (resp. 340 frames/s for 8 us).
- Within the saturation region (too large injection rates, above 500 frames/s), List slightly outperforms CQ (by ~2.7%).

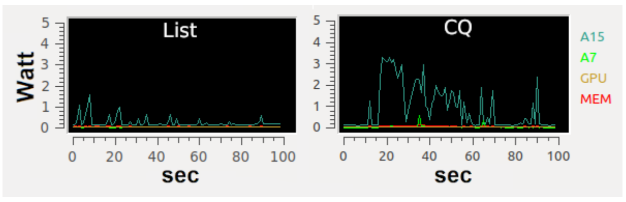
Finally, Fig. 9 illustrates the frame loss vs the injection rate (for large injection rates), for two different UART inter-character delays: 4 us and 8 us. Notice that

- frame loss is very small for both CQ and List (less than 0.1%) when the UART delay is 8 us, but
- quite large especially for CQ (2.0% to 26.5%) when the UART delay is set to 4 us.

D. Power Consumption on the Odroid XU3 Gateway

For power estimation, the Odroid XU3 gateway integrates four TI INA231 current-shunt and power sensors equipped with Analog Digital Converter (ADC) circuits to obtain power data from A15 cores, A7 cores, memory, and GPU. The INA231 sensors have high accuracy with a maximum error of 0.5% ppm/°C for temperatures ranging from -40°C to +125°C which is realistic for this work. The I2C driver is configured so that on-chip sensor configuration and data registers can be sampled from within our CAN Gateway application via sysfs file system entries. The I2C driver supports an update rate of 200ms (or higher depending on configuration), however, by averaging instant values, an average power consumption for each cluster (big or little) is obtained once per second. It is also possible visualize all on-chip sensors (including thermal) by adapting the architecture-independent Qt-based Energy Monitor [24]; another compatible power tool that enables detailed per-process power profiling is ALEA [25].

Figs. 10 (a) and (b) show CPU, memory, and GPU power consumption for List (left) and CQ (right graph). Notice that only A15 power consumption is significant since threads recv and send execute on two Cortex-A15 and access shared registers and cache. In addition, observe that the CQ is much more power intensive (~3.1 Watt/s) than List (~0.8 Watt/s), due to more complex synchronization mechanisms; in contrast, List only uses two semaphores (locks) to synchronize accesses. Data locality also plays an important role in power efficiency; however, these issues must be further investigated.



(a) (b)

Figure 10. Typical power profiles from the Energy Monitor for a) List (left) and b) CQ (right), assuming a UART inter-character delay of 8 us.

A large drop in power consumption can be accomplished using application watchdog monitors that leverage the OBD dev kit power module's ability to throttle, sleep, or shut down traffic with just 20ns activation/deactivation time.

IV. Conclusions and Future Work

The proposed design of gateway architectures efficiently transports frames among in-vehicle subsystems using traditional lock-based producer-consumer or lock-free concurrent queues. This work indicates that both queue implementations support high performance with dequeue rates reaching ~380 CAN frames/s. Furthermore, although lock-free concurrent queues have a slightly smaller waiting time, they have much higher energy requirements.

The developed open embedded gateway platform prototype is based on inexpensive, mostly general-purpose hardware (Raspberry Pi 3B, Odroid XU3, and OBD Development Kit). The platform can help in teaching about in-vehicle subsystems. The platform can be expanded with programmable ECU devices (e.g., Scantool ECUsim2000), and many other physical or virtual interfaces [26, 27]. With hands-on experience, in analyzing, visualizing, or controlling in-vehicle messages on clusters of ECU nodes and gateways, Computer Engineering and Electrical Engineering students can understand better the structure of efficient, reliable, and secure automotive communications in real vehicles. Moreover, students can improve their knowledge of operating systems, parallel and distributed systems, data structures, networking, embedded systems, and databases, since efficient communication and synchronization strategies, such as the producer-consumer problem (central in this study), transcend these computer science/engineering domains [28].

The open software architecture and implementation, including an extensive troubleshooting guide, will be made available to the community via SourceForge/GitHub soon (in Q2/2023).

Besides CAN bus, automotive makers deploy other in-vehicle network technologies that operate with different communication protocols and baud rates. Thus, future research can also focus on hybrid gateway architectures that interconnect multiple in-vehicle subsystems that operate at different rates, e.g., CAN, Local Interconnect Network (LIN), Media Oriented Systems Transport (MOST), Flexray, CAN FD, and Wireless. In this case, the gateway architecture can leverage multiple-producer multiple-consumer (MPMC) queues or appropriately adapted single-producer single-consumer solutions using buffered multistage networks. Finally, Linux RT policies can help improve the real-time performance of our gateway architecture.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Conceptualization of the idea, and supervision by MDG, and GK; Research, implementation, and analysis during different integration stages of the platform by SN, MDG, AS, DB, NM. MDG and SN wrote the paper; all authors reviewed the paper and approved the final version.

Funding

This research was partially supported from EU H2020 projects AVANGARD (Contract No. 869986) and more recently FLUIDOS (No. 101070473).

References

[1] B. Mobility. (2022). Vehicle-centralized, zone-oriented E/E architecture with vehicle computers. [Online]. Available: https://www.bosch-mobility-solutions.com/en/mobility-topics/ee-architecture

[2] S. Brunner, J. Roder, M. Kucera *et al.*, "Automotive E/E-architecture enhancements by the usage of Ethernet TSN," *Workshop on Intelligent Solutions in Emb. Syst. (WISES)*, 2017, pp. 9-13.

[3] D. Wang and S. Ganesan, "Automotive domain controller," *Int. Conf. Comput. Info Tech. (ICCIT)*, 2020, pp. 1-5.

[4] V. Bandur, G. Selim, V. Pantelic *et al.*, "Making the case for centralized automotive E/E architectures," *IEEE Trans. on Vehic. Tech.*, vol. 70, no. 2, 2021, pp. 1230—1245.

[5] R. Bosch, "CAN Specification v2.0," Stuttgart, Germany, 1991.

[6] Cameron. (2013). A single-producer, single-consumer lock-free queue for C++. [Online]. Available: https://github.com/cameron314/readerwriterqueue

[7] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, 1967, pp. 341—346.

[8] Producer-Consumer, Dijkstra's solution, Wikipedia. (Sept. 24, 2022). [Online]. Available:https://en.wikipedia.org/wiki/Producer-consumer_problem

[9] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, 1983, pp. 190-222, 2022.

[10] M. Meneghin, D. Pasetto, H. Franke *et al*., "Performance evaluation of interthread communication on multicore architectures," *Int. Symp. High-Perf. Parallel and Distributed Comput.*, 2013, pp. 131-132.

[11] V. Maffione, G. Lettieri, and L. Rizzo, "Cache-aware design of general-purpose Single-producer—single-consumer queues," *J. Softw. Pract. Exp.*, vol. 49, 2019, pp. 748—779.

[12] U. E. Larson, D. K. Nilsson, and E. Jonsson, "An approach to specification-based attack detection for in-vehicle networks," in *Proc. IEEE Intelligent Vehicles Symp.*, 2008, pp. 220—225.

[13] W. Wu, R. Li *et al*., "A survey of intrusion detection for in-vehicle networks," *IEEE Trans. Intelligent Transp. Systems*, vol. 21, no. 3, 2020, pp. 919-933.

[14] M. D. Pesé, J. W. Schauer, J. Li, and K. G. Shin, "S2-CAN: Sufficiently secure controller area network," in *Proc. Computer Security Appl. Conf. (ACSAC)*, 2021, pp. 425-438.

[15] L. Teri and R. Bolboaca, "A stateful firewall and intrusion detection system enforced with secure logging for controller area network," in *Proc. ACM European Interdisciplinary Cybersecurity Conf. (EICC)*, 2021, pp. 39-45.

[16] H. Ekiz, A. Kutlu, and E. Powner, "Design and implementation of a CAN / CAN bridge," in *Proc. Int. Parallel Arch., Algorithms, and Networks (I-SPAN)*, 1996, pp. 507—513, 1996.

[17] H. Ekiz, A. Kutlu, and E. Powner, "Implementation of CAN/CAN bridges in distributed environments and performance analysis of bridged CAN systems using SAE benchmark," in *Proc. IEEE Engineering*, 1997, pp. 185- 187.

[18] J. Sommer and R. Blind, "Optimized resource dimensioning in an embedded CAN-CAN gateway," in *Proc. Int Symp. Industrial Embedded Systems*, 2007, pp. 55-62.

[19] G. Xie, H. Gong, Y. Han *et al*., "A real-time CAN-CAN gateway with tight latency analysis and targeted priority assignment," in *Proc. EEE Real-Time Systems Symp. (RTSS)*, 2020, pp. 141-152.

[20] M. K. Budde, "SocketCAN: The official CAN API of the linux kernel," in *Proc. CAN Conf.*, 2012, pp. 5—17.

[21] Serial Console. SourceForge. (Dec. 24, 2022) [Online]. Available: https://sourceforge.net/projects/serialconsole/files/serialconsole

[22] Car-Hacking dataset, HCRL. (Dec. 12, 2022). [Online]. Available: https://sites.google.com/a/hksecurity.net/ocslab/Datasets/CAN-intrusion-dataset

[23] T. Preud'homme, J. Sopena, G. Thomas *et al*., "Batchqueue: Fast and memory-thrifty core to core communication," in *Proc. Int. Symp. on Computer Arch. and High-Perf. Comput.*, 2010, pp. 215-222.

[24] C. Imes, L. Bergstrom, and H. Hoffmann, "A portable interface for runtime energy monitoring," in *Proc. 24th Int Symp. Found. Soft. Engin.*, 2016, pp. 968-974.

[25] L. Mukhanov, D. S. Nikolopoulos, and B. R. De Supinski, "ALEA: fine-grain energy profiling with basic block sampling," *Int. Conf. Parallel Arch. and Compilation (PACT)*, 2015, pp. 87-98.

[26] S. McCall, C. Yucel, and V. Katos, "Education in cyber-physical systems security: The case of connected autonomous vehicles," in *Proc. IEEE Global Engin. Education Conf. (EDUCON)*, 2021, pp. 1379-1385.

[27] C. Liu and F. Luo, "A co-simulation-and-test method for CAN Bus system," *Journal of Communications*, vol. 8, no. 10, 2013, pp. 681-689, 2013.

[28] Y. Luo and X. Wang, "The study of the classic producer-consumer problem in a series of IT courses," in *Proc. Conf. Info. Tech. Education (SIGITE20)*, 2020, pp. 162–-16