EFFICIENT GATEWAY IMPLEMENTATIONS FOR CYBER-PHYSICAL SYSTEMS

by

STYLIANOS G. NINIDAKIS

Computer Science B.Sc., University of Crete, 2013

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2022

Approved by:

Major Professor
Miltos D. Grammatikakis

Abstract

Smart devices with numerous sensors and actuators connected to servers via heterogeneous networks form an increasing technology trend. The transmission, storage, and manipulation or analysis of big data raise many issues which the research community should address. A common practice to lighten a server's workload is adding an intermediate device (gateway) to (1) synchronize various networks, (2) gather, filter, and preliminarily analyze the data from the sensors, (3) control any local actuators and (4) send the extracted data to the server.

Since gateway devices are often portable embedded systems with restricted processing capability to easily function close to sensors, software optimization plays a crucial role in ensuring data integrity and real-time functionality. Parallel execution and concurrent buffers could easily contribute toward that direction. In this study, we investigate the way that concurrent buffers implemented as wait-free single-producer single-consumer queues perform on two different use cases: (1) a Bluetooth-to-Wireless healthcare gateway transferring biosensor (ECG) data, and (2) a CAN-to-CAN in-vehicle gateway transmitting CAN frames.

In the Bluetooth-to-Ethernet gateway, we focus on enhancing the performance of biosensor data flows across the gateway by examining lock-free concurrent queues and traditional lock-based circular FIFOs that embody single-producer single-consumer principles. Our results show that for large biosensor rates, the waiting time of the concurrent queue is smaller. Large rates, above 256 pulses/s, are not supported by the biosensor but are examined by designing a self-similar, digital twin process.

In the CAN-to-CAN gateway, we examine efficient frame transfer across incoming and outgoing CAN interfaces at the gateway. In our embedded platform prototype, each CAN interface is controlled independently by a corresponding POSIX thread. Inter-thread communication is implemented using shared data structures, namely lock-free concurrent queues, and traditional lock-based circular FIFOs. Our experimentation relies on increasing communication reliability and performance by minimizing frame loss and improving the egress CAN frame rate.

Our experiments show that the frame waiting time in the queue is smaller 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 higher for the faster concurrent queue implementation.

Περίληψη

Οι έξυπνες συσκευές με πολλούς αισθητήρες (sensors) και ενεργοποιητές (actuators) σε ετερογενή δίκτυα οι οποίοι παράγουν και απαιτούν δεδομένα, αντίστοιχα, αποτελούν μία όλο και πιο σύγχρονη τάση στην τεχνολογία και συγκεντρώνουν μεγάλο επιστημονικό και ερευνητικό ενδιαφέρον στις μέρες μας. Ωστόσο, η μετάδοση, η διαχείριση και η ανάλυση αυτού του μεγάλου όγκου δεδομένων (big data) στο διακομιστή εγείρουν πολλά ζητήματα, τα οποία απασχολούν την ερευνητική κοινότητα, στα οποία καλείται να προτείνει λύση. Μια κοινή πρακτική για τη μείωση του φόρτου εργασίας ενός κεντρικού διακομιστή είναι η προσθήκη μιας (ή περισσοτέρων) ενδιάμεσης συσκευής (πύλη-gateway) για: (1) το συγχρονισμό ετερογενών δικτύων, (2) τη συλλογή, τη διαλογή και την ανάλυση των δεδομένων από τους αισθητήρες, (3) τον έλεγχο των ενεργοποιητών και (4) την αποστολή των εξαγόμενων δεδομένων στον κεντρικό διακομιστή. Δεδομένου ότι οι συσκευές πύλης (gateway devices) είναι συνήθως φορητά ενσωματωμένα συστήματα με περιορισμένες επεξεργαστικές δυνατότητες και ικανότητα να λειτουργούν εύκολα κοντά στους αισθητήρες λόγω του μεγέθους και των ενεργειακών τους αναγκών, η βελτιστοποίηση του λογισμικού παίζει κυρίαρχο ρόλο διασφαλίζοντας την ακεραιότητα των δεδομένων και τη λειτουργικότητα σε πραγματικό χρόνο. Η παράλληλη εκτέλεση και η χρήση δομών (buffers) που επιτρέπουν τον ταυτοχρονισμό (concurrency) θα μπορούσε εύκολα να διευκολύνει τη διαδικασία προς αυτή την κατεύθυνση. Στην παρούσα μελέτη, διερευνήσαμε τον τρόπο με τον οποίο οι παράλληλες δομές χωρίς αναμονή ενός παραγωγού και ενός καταναλωτή (wait-free single-producer single-consumer queues) αποδίδουν σε δύο διαφορετικές περιπτώσεις χρήσης: (1) ένα σύστημα μετάδοσης δεδομένων ηλεκτροκαρδιογραφήματος (ECG) μέσω BT-Wireless gateway, και (2) σε ένα εσωτερικό δίκτυο μεταγωγής πακέτων σε ένα σύγχρονο όχημα (in-vehicle CAN-to-CAN gateway).

Στην πύλη ECG, εστιάζουμε στη βελτίωση της απόδοσης της ροής δεδομένων ενός βιοαισθητήρα (biosensor) κατά μήκος μιας πύλης Bluetooth-to-Ethernet, εξετάζοντας ταυτόχρονες ουρές χωρίς κλειδαριά (lock-free concurrent queues) και παραδοσιακές κυκλικές FIFO που βασίζονται σε κλειδαριές αμοιβαίου αποκλεισμού (traditional lock-based circular FIFOs) που ενσωματώνουν αρχές μονού παραγωγού μονού καταναλωτή (single-producer single-consumer). Τα αποτελέσματά μας δείχνουν ότι για υψηλού ρυθμού μετάδοσης δεδομένων ενός βιοαισθητήρα, ο χρόνος αναμονής της παράλληλης ουράς (concurrent queue) είναι μικρότερος.

Οι μεγάλοι ρυθμοί, πάνω από 256 παλμούς/δευτ., δεν υποστηρίζονται από το βιοαισθητήρα, αλλά εξετάζονται μέσω μιας πανομοιότυπης (self-similar) διαδικασίας ψηφιακού διδύμου (digital twin process).

Στην δεύτερη περίπτωση χρήσης που αφορά στην πύλη CAN, εξετάζουμε την αποτελεσματική μεταφορά πλαισίων (frame transfer) μέσω εισερχόμενων και εξερχόμενων διασυνδέσεων CAN (incoming and outgoing CAN interfaces). Στο πρωτότυπο (prototype) της ενσωματωμένης πλατφόρμας μας, κάθε διεπαφή CAN ελέγχεται ανεξάρτητα από ένα αντίστοιχο νήμα POSIX (corresponding POSIX thread). Η επικοινωνία των νημάτων (Inter-thread communication) υλοποιείται χρησιμοποιώντας κοινόχρηστες δομές δεδομένων (shared data structures), πιο συγκεκριμένα ταυτόχρονες δομές χωρίς κλειδαριές αμοιβαίου αποκλεισμού (lock-free concurrent queues), και παραδοσιακές κυκλικές ουρές βασισμένες σε κλειδαριές αμοιβαίου αποκλεισμού (mutual exclusion lock-based queues).

Η πειραματική μας διαδικασία βασίζεται στην αύξηση της αξιοπιστίας και της απόδοσης της επικοινωνίας, ελαχιστοποιώντας την απώλεια πακέτων (frame loss) και βελτιώνοντας το ρυθμό μετάδοσης πακέτων της εξόδου CAN (egress CAN frame rate). Τα πειράματά μας δείχνουν ότι ο χρόνος αναμονής της μετάδοσης των πακέτων στην ουρά είναι μικρότερος για την ταυτόχρονη ουρά (concurrent queue), ενώ η αναμετάδοση μηνυμάτων είναι εξίσου γρήγορη, φτάνοντας ένα μέγιστο ρυθμό εξόδου περίπου 380 πακέτα/δευτερόλεπτο, για μεγάλους ρυθμούς εισόδου κοντά σε κορεσμό (saturation). Τέλος, η κατανάλωση ρεύματος είναι υψηλότερη στην περίπτωση ταχύτερης υλοποίησης ταυτόχρονης ουράς (faster concurrent queue implementation).

Table of Contents

List of Figures viii
Acknowledgments x
Preface xii
Chapter 1 – Introduction 1
1.1 Gateway Applications 3
1.2 Gateway Architecture 6
1.2.1 Bluetooth-to-Ethernet Gateway Architecture 7
1.2.2 CAN-CAN / CAN-Ethernet Gateway Architecture 9
1.3 Concurrent Queues 11
1.4 Objectives 12
1.6 Thesis Outline 13
Chapter 2 - Efficient Gateway for Biosensors 14
2.1 Driver Architecture 15
2.2 Methodology and Optimizations 16
2.2.1 Optimizations 17
2.2.2 Digital Twin Concept 17
2.3 Experimental Results 18
2.3.1 Gateway Metrics: Average Waiting Time vs BGW ECG Rate 18
2.3.2 File Server Application Metrics: Instant Pulse Rate vs Time 20
2.3.3 File Server Application Metrics: Cumulative Average Pulse Rate vs Time 22
2.3.4 Energy Consumption 24
Chapter 3 - Efficient In-Vehicle Gateway 25
3.1 Software Stack 26
3.2 Methodology & Optimizations 29
3.3 Performance Evaluation 30
3.3.1 Detecting the Optimal Inter-Character Delay of UART Sender 30
3.3.2 Queue Performance on the Odroid XU3 Gateway 32
3.3.3 Energy Consumption 34
Chapter 4 – Conclusions 37
Chapter 5 - Future Work 39
Bibliography 40

List of Figures

Figure 1.1. IoT stack 1
Figure 1.2. Edge gateway diagram [2] 4
Figure 1.3. Interconnection among in-vehicle different modules [3] 5
Figure 1.4. The architecture of healthcare monitoring systems in Smart Intensive Care Units [5] 6
Figure 1.5. IoT gateway architecture. [6] 7
Figure 1.6. The overall architecture of the Bluetooth gateway system [7] 8
Figure 1.7. Domain-based Architecture [8] 9
Figure 1.8. CAN-CAN/Ethernet Gateway architecture overview 10
Figure 1.9. Inter-thread communication in the gateway. 10
Figure 1.10. Queue performance at the Odroid XU3 gateway vs injection rate. 12
Figure 2.1. The gateway architecture with a complex BT-to-Wireless protocol stack. 16
Figure 2.2. Waiting time in the CQ or the list (circular FIFO) vs the BGW rate, both running simultaneously on the same device. 19
Figure 2.3. Waiting time in the CQ or the list (circular FIFO) vs the BGW rate, both running on different devices. 20
Figure 2.4. The instant arrival rate of ECG samples at the server for CQ and list, when (a) BGW is set at 128 pulses/s (up), or (b) 256 pulses/s (down). 21
Figure 2.5. The instant arrival rate of ECG samples at the server, for CQ and list, is shown; higher BGW rates (1024, 2048, or 4096 pulses/s) are based on a normalized digital twin model. 22
Figure 2.6. The cumulative average arrival rate of ECG samples at the server for CQ and list, when BGW is set at 128 pulses/s, or 256 pulses/s. 23
Figure 2.7. The cumulative arrival rate of ECG samples on the server for CQ and list. When using the normalized digital twin concept, the BGW rate is set to (1024, 2048, or 4096) pulses/sec. 23
Figure 2.8. Energy consumption (in Watts) for CQ and list at selected rates. 24
Figure 3.1. A prototype of our open distributed embedded platform. ECU components are based on Raspberry Pi with Canberry SPI-to-CAN interfaces, while the gateway is an Odroid XU3 with UART-to-CAN interfaces (via OBD development kit) [46] 27
Figure 3.2. The embedded software stack at the gateway. Threads compete for the shared buffer; the architecture is extendible to many (incoming/outgoing) interfaces. 29
Figure 3.3. Packet losses on different UART inter-character delay values for two different rates (192 and 61.5 frames/sec). 31
Figure 3.4. Average packet waiting time at various injection rates for CQ and List. The inter-character delay is set to 8us 32
Figure 3.5. Correspondence between injection and output rate for each implementation 33
Figure 3.6. Percentage of packet losses vs injection rate. 34
Figure 3.7. Typical power profiles from the Energy Monitor for a) List (left) and b) CQ (right), assuming a UART inter-character delay of 8 us. 35

Acknowledgments

I am grateful to my supervisor, Professor Miltos D. Grammatikakis, who assigned me the topic of this MSc Thesis and he was always present with his continuous support and guidance. I sincerely thank him because he believed in me, and he always helped me by suggesting a solution to all the problems and difficulties I faced. Moreover, I would like to thank my friends and colleagues Vaggelis, Alexandros, Giorgos, Haris, Savvas and Antonis, who supported and encouraged me during my MSc studies. I would also like to thank Magda, who has always been there for me, supporting and helping me in all possible ways.

Finally, I am grateful to my parents Giorgos and Eleni for their unconditional love and support. Thank you for bearing with me and being always by my side through my studies and life.

Preface

The comparative study among four concurrent queue implementations on the Odroid XU3/XU4 board, which is described in Section 1.4 was performed by Alexios Staridas at the Artificial Intelligence and System Engineering Laboratory (AISE Lab) of the Hellenic Mediterranean University (HMU).

Parts of the results of Chapter 2 were presented at ApplePies2022 Conference. This paper will be published in the Springer Lecture Notes in Electrical Engineering book series, under the title “Towards Efficient Gateways and Servers for Biosensors”.

Preliminary results from Chapter 3 are under review in the DATE 2023 Conference in a paper entitled “Towards Efficient In-Vehicle Gateways”.

Chapter 1 – Introduction

The Internet of Things (IoT) [1] is ever-expanding, playing a significant role in real life. The extensive use of IoT systems in the industry, health, energy, and smart living has increased productivity, energy saving, and better living in general. Since smart things make use of many sensors/actuators over different networks while they are connected to the internet, several issues have risen related to security, reliability, responsiveness, and big data management. To address these issues, embedded gateway devices are widely used as they can host implementations that enable the design of more complex systems.

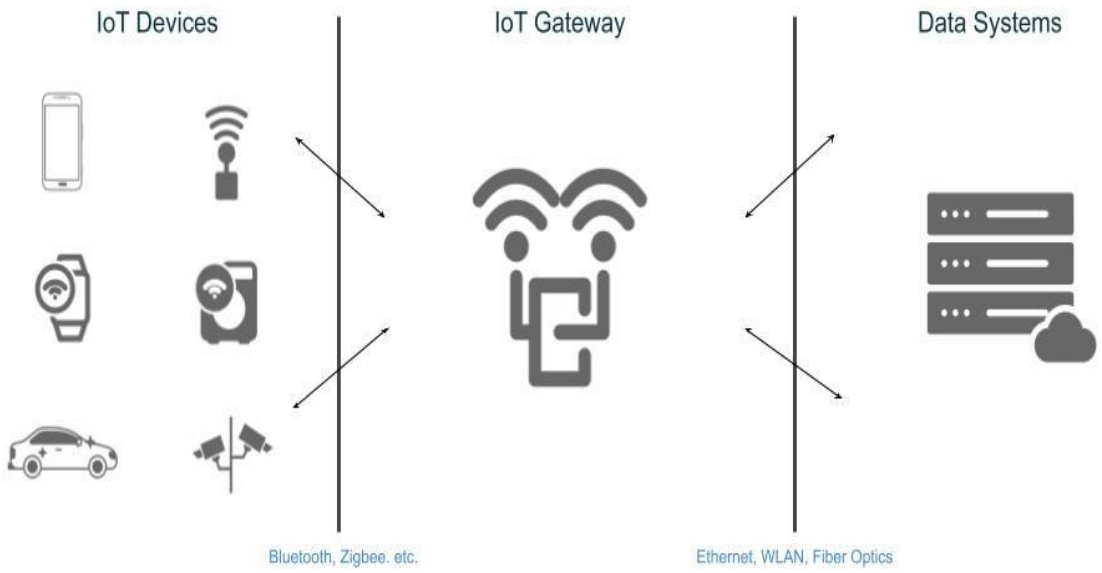


Figure 1.1. IoT stack

Gateways form a 'bridge' between legacy and next-generation devices to the IoT idea, as they are called to fulfill a plethora of operations as an intermediate device between the Things and the Cloud. More specifically, third generation IoT gateways are critical system components as they:

1. integrate lightweight narrow-bandwidth device networks (RS232, RS485, Bluetooth, wireless, etc.) to communicate with sensors, actuators, or other subsystems, and broadband heavyweight networks (e.g. the Internet) that transmit sensor data to server or

cloud systems for further analysis and decision making, while also ensuring their integrity,

2. manage the storage of the data, filter, prioritize, and commonly apply analytics on them,
3. receive or send intelligence from/to the Cloud and - in some cases - make autonomous decisions,
4. facilitate data flow securely between edge devices and the Cloud, as Figure 1.1 shows.

1.1 Gateway Applications

Due to their capabilities, third generation gateways are applied to a huge number of systems nowadays. Ostensive and extensive usage applications of gateways can be found in the automotive industry, healthcare systems, IoT and Industrial IoT (IIoT) ecosystems, and sensor networks.

In the case of IoT systems, the use of an intermediate gateway could reduce the total workload of the cloud by filtering data gathered from the sensors, extracting analytics, and making decisions.

For instance, modern smart homes are equipped with many smart devices, including multiple sensors/actuators to monitor and perform smart operations. The sensors are connected to a local IoT microcontroller over wired networks (on-board or peripherals). The microcontroller communicates with the cloud or directly to users' smartphones through one or more edge devices over Wi-Fi or ethernet protocols. Let us consider a smart washing machine· its sensors monitor temperature, water flow, pressure, vibration, dirtiness, door status, and load of washing. Some possible actuators may focus on switching on/off the device, starting, pausing, or stopping the washing process, and opening the door. The sensors periodically or (ideally in real-time) give their status change to the microcontroller, which processes the data and - in conjunction with the user's selections - calls the actuators to perform an action and informs the cloud/user. If any malfunctions are detected, the microcontroller decides to prohibit or block some operations and instantly informs the user. Figure 1.2 shows a diagram of an edge IoT gateway communicating with numerous smart devices.

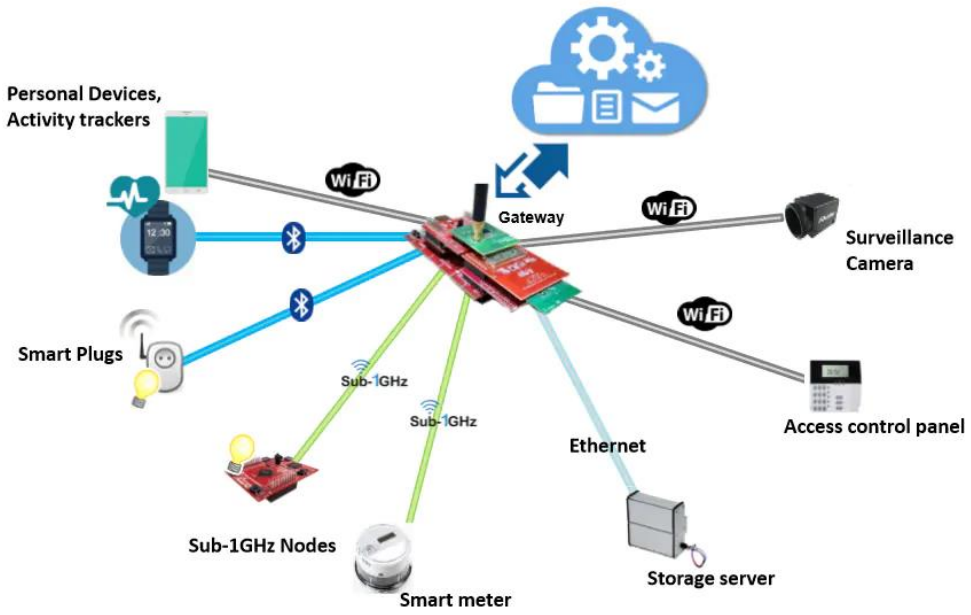


Figure 1.2. Edge gateway diagram [2]

Unlike IoT, Real-Time Embedded Systems (RTES), Cyber-Physical Systems (CPS) and Wireless Sensor Networks (WSN) related to healthcare or automotive require more autonomy that plays a crucial role, since a malfunction may threaten human life. In those cases, sensors/actuators are rated by criticality and more critical sensors have a significant priority compared to the rest (multimedia, humidity sensors, etc.). When priority is strictly considered, different network protocol synchronization and prioritization assurance among the messages of critical sensors/actuators and the gateway become important, as well as possible latencies.

For example, a proximity sensor on a car detects an object approaching it, but the driver takes no action (slowing or braking). The gateway must be instantly informed and actuate the hydraulic brakes to slow down. Any latencies from network congestion or CPU load may prove to be disastrous since the threshold for this critical operation is 50ms or less.

Automotive systems are composed of soft and hard real-time embedded subsystems. Ensuring synchronization among network protocols and message dependencies and ordering using static priorities makes the design of these systems even more complex. A gateway is a common solution to manage critical and non-critical in-vehicle networks (see Figure 1.3).

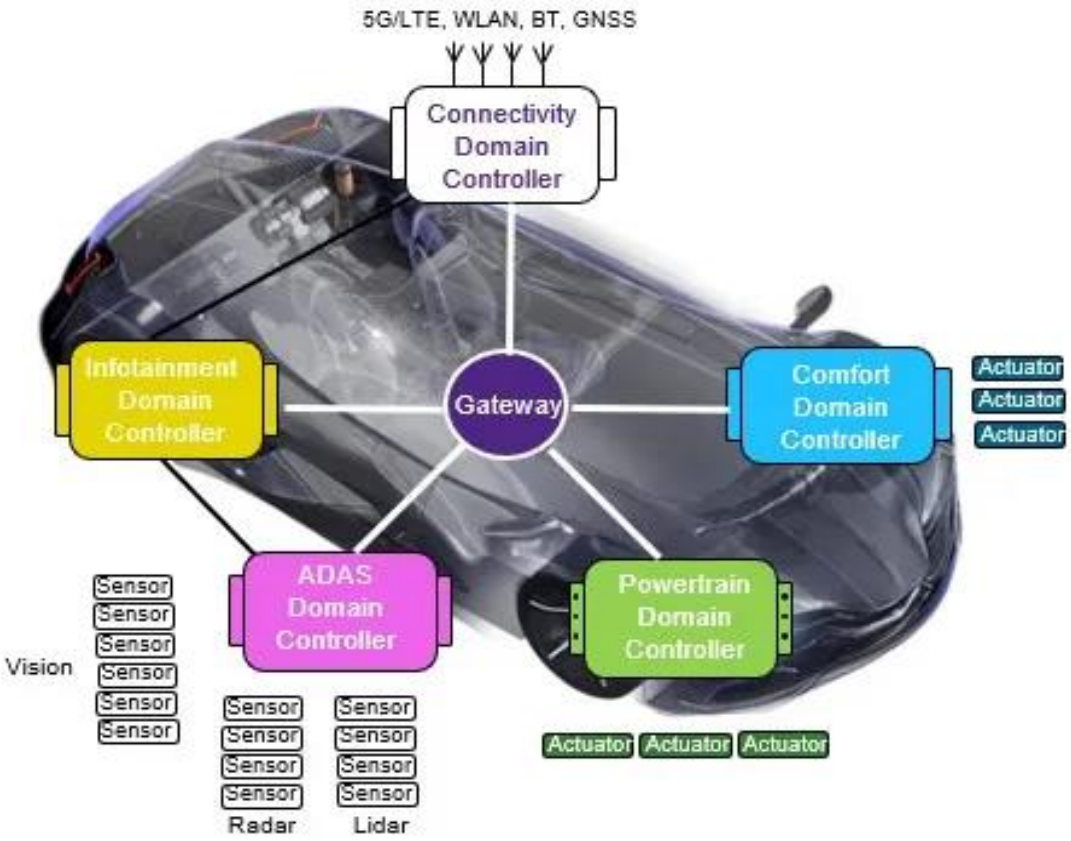


Figure 1.3. Interconnection among in-vehicle different modules [3]

The most common network in automobiles is the Controller Area Network (CAN bus) [4] by which multiple devices and sensors intercommunicate with different prioritizations. Other network protocols are LIN, FlexRay, MOST, and Ethernet which provide wider bandwidth and higher speed. On top of them, other wireless networks (Wi-Fi, Bluetooth, and Cellular Networks) could also be combined to extend automobile functionalities, such as mobile phone Bluetooth connection, Cloud services, embedded phone features, etc.

Embedded systems have also many applications in the healthcare industry. They are used for monitoring vital signs, amplifying sounds in electronic stethoscopes, and nearly every kind of imaging system, including PET scans, CT scans, and MRIs. Glucose monitors, pacemakers, CPAP machines, and a variety of biomedical sensors also rely on embedded systems. With biomedical applications, embedded systems allow doctors to remotely monitor patients' health and make diagnoses and treatment decisions through telemedicine and other remote systems.

For the reasons above, gateways may have to manage a huge amount of data steaming from several sensors, which is mandatory to be kept for further analysis and patient profiling. In

some cases, such as remote surgery, real-time responsiveness also plays an important role. Figure 1.4 shows an example of the healthcare system topology.

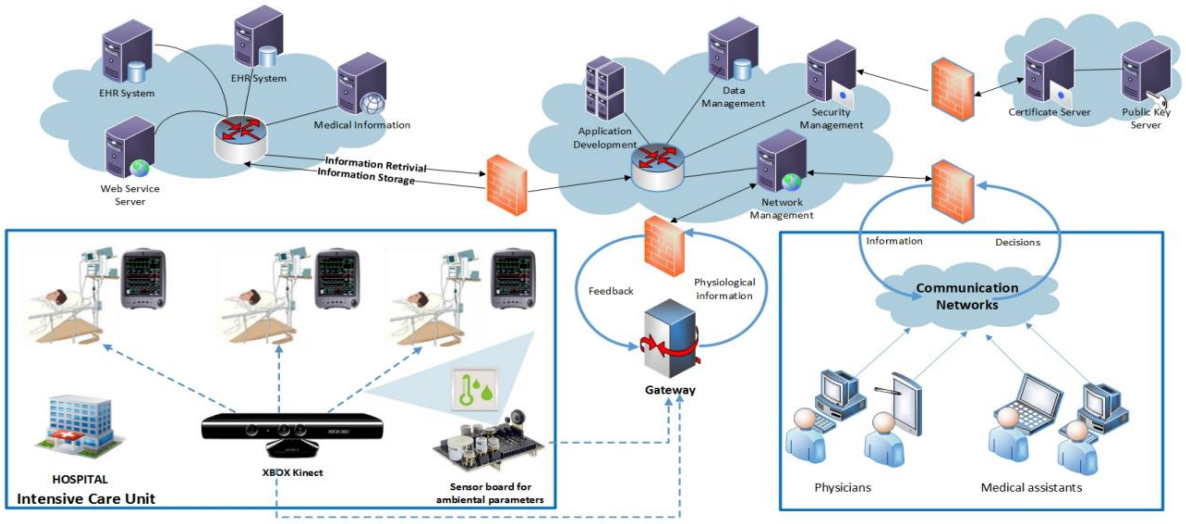


Figure 1.4. The architecture of healthcare monitoring systems in Smart Intensive Care Units [5]

Regardless of the context, IoT, CPS, etc., recent technologies tend to integrate many sensors, actuators (as peripherals or on-board), and other peripheral devices, with mixed criticality levels, different topologies, and different communication interfaces. Peripheral devices transmit data over networks like CAN Bus, Bluetooth, Ethernet, WLAN, 4G/5G, etc., while the onboard ones use internal buses. All these parameters make managing, scheduling, and proper operation even more complex for the gateway device since it is called to process many requests at a time.

1.2 Gateway Architecture

Modern IoT gateways consist of several components arranged in different layers. They are responsible for a specific reason, e.g., network management, sensor stacks, etc. Thus, depending on the aim of the gateway, different components can be present in the architecture.

Figure 1.5 shows an IoT gateway architecture that contains several components separated into five layers. The two base layers include, on the one hand, the integrated hardware (CPU, RAM, storage modules, connection modules, etc.), and on the other hand, an operating system (commonly lightweight) to manage the available hardware. On top of them, a variety of protocols, stacks, and microservices are used to synchronize sensors, ensure security, and enable

communication between edge devices remotely accessed, managed, or updated (FoTA - Firmware over The Air).

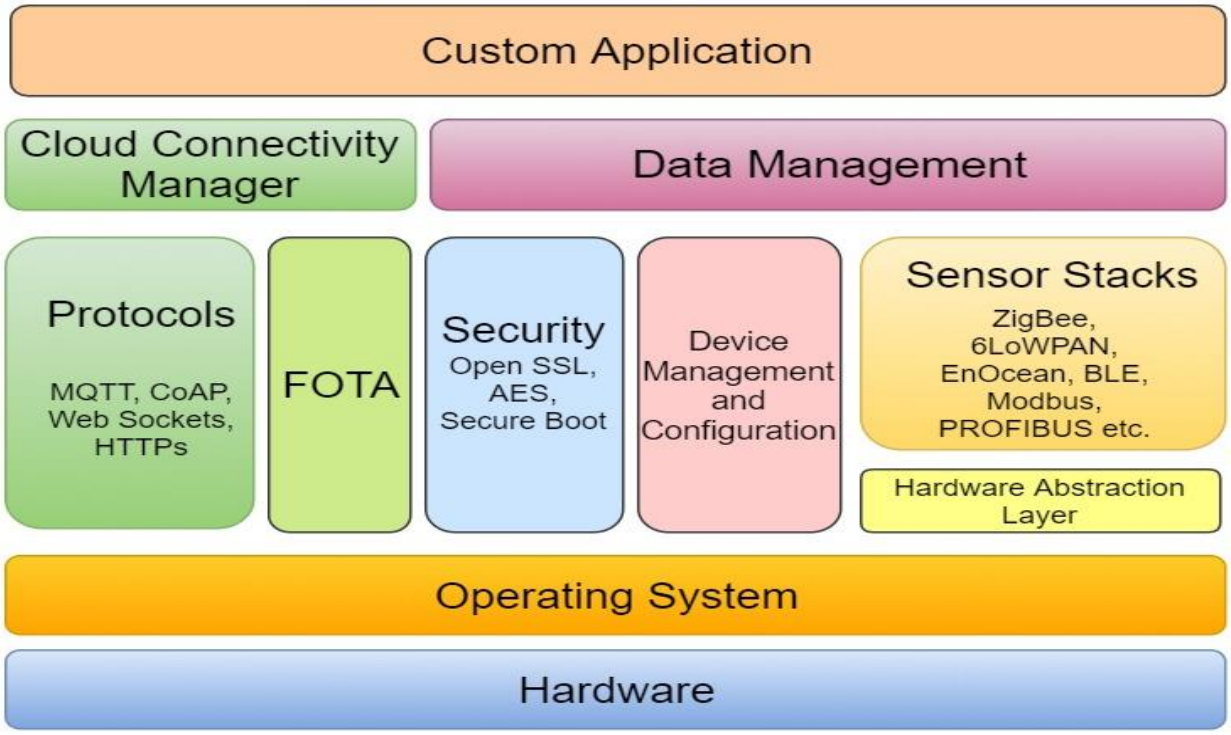


Figure 1.5. IoT gateway architecture. [6]

In this thesis, we deal with Bluetooth-to-Ethernet and CAN-to-CAN gateway architectures. These architectures are further detailed in the following subsections.

1.2.1 Bluetooth-to-Ethernet Gateway Architecture

Bluetooth is a short-range wireless communication technology standard. It uses short-wavelength UHF radio waves of a frequency range between 2.4 and 2.485 GHz. Bluetooth enables one to create an area network wherein multiple devices talk to each other wirelessly via Bluetooth or to a central control system - a typical usage is the hospital monitoring of patients with cardiovascular diseases. For example, all electronic sensors in a hospital ward can be connected to a central control system via Bluetooth, and the central control system is controlled over the Internet.

Gateways can play an important role in a large-scale architecture of transferring data from different areas to the Cloud over the Ethernet/WLAN. They could receive data from sensors over Bluetooth and send them synchronized and serialized to the Cloud in real time. Extending the example of the previous paragraph, an individual gateway can be used for collecting data from a group of sensors (e.g., in a ward or a few rooms). All gateways transfer the data to the Cloud over the Ethernet network to a remote server.

As Figure 1.6 shows, a Bluetooth module manages to receive data from several remote sensors. Once received, data are cached in a central queue (or a dedicated queue for each sensor). A Wi-Fi/Ethernet module is responsible for gathering data from the queue and sending them to a remote server over the Internet. The Queue Management and Feedback Mechanism assure correct functionality.

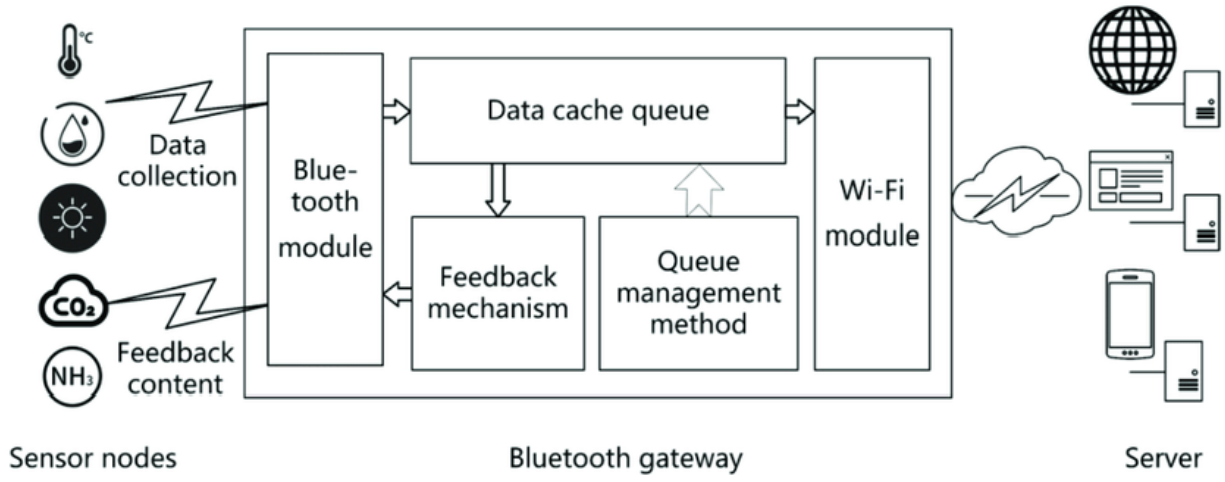


Figure 1.6. The overall architecture of the Bluetooth gateway system [7]

1.2.2 CAN-CAN / CAN-Ethernet Gateway Architecture

The classic in-vehicle network architecture (presented in Figure 1.3) facilitates a central gateway that is responsible to communicate and route data to/from a variety of ECUs. Because of the increased communication overhead as more ECUs and devices are added in modern vehicles, a domain-based architecture was developed, where various functional domains with a domain controller are interconnected through the central gateway. According to this model, most of the data communication occurs within these functional domains, reducing the communication load on the gateway. Furthermore, the architecture is scalable to allow adding more functional domains. The domain-based architecture is depicted in Figure 1.7.

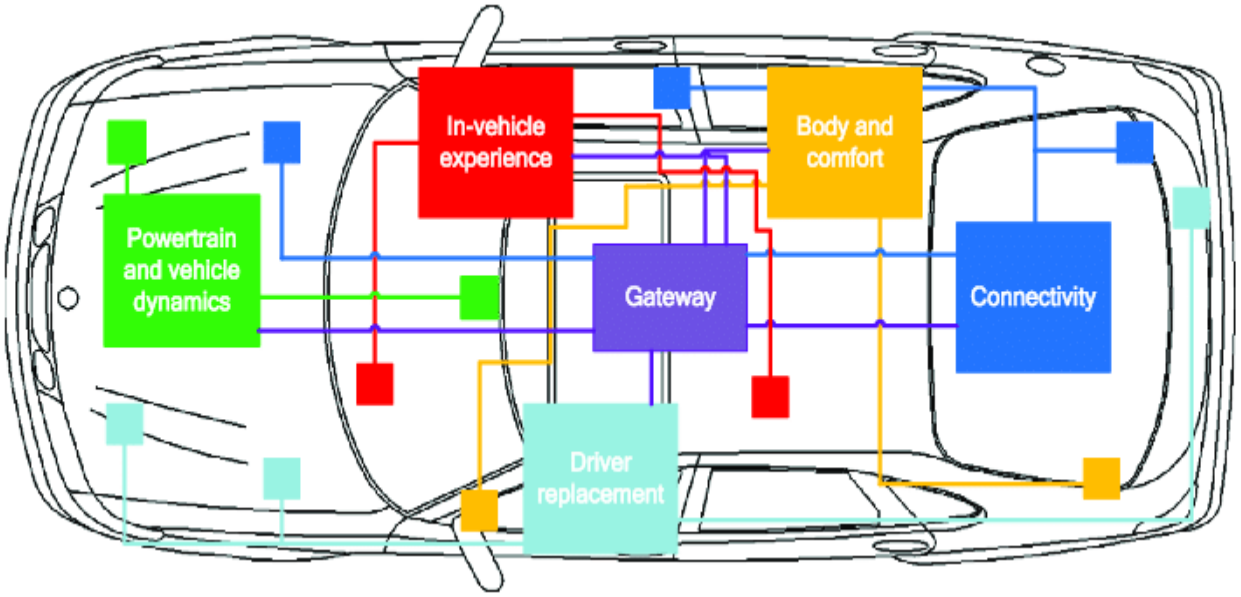


Figure 1.7. Domain-based Architecture [8]

As mentioned in Section 1.1, CAN is the widely used protocol in vehicular communications to provide and control various functionalities. It enables critical real-time communications for engine management, brake control, airbags, body systems control, etc. The CAN protocol uses inter-connected buses to transmit CAN frames between various ECUs inside a vehicle network.

A gateway (or domain controller) is responsible for connecting domain devices to other domains or the central gateway. Let's assume a case where all devices of a specific domain are

interconnected via a CAN bus, and there is a domain controller on that bus, which communicates with the central gateway over another CAN bus or Ethernet (Figure 1.8).

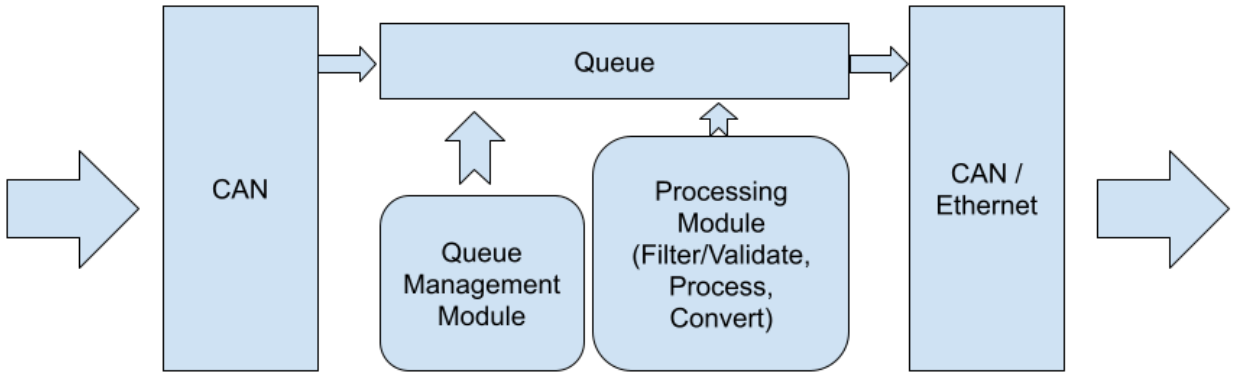


Figure 1.8. CAN-CAN/Ethernet Gateway architecture overview

The most common architecture found on such a gateway includes two connection modules (CAN and CAN, or Ethernet), a queue (or a system of queues), and software modules to either synchronize accesses into the Queue or process the data (validation, filtering, security check, analysis, conversion, etc.). The communication modules (receiver and sender) and the Queue enqueues/dequeues are controlled by a set of threads dedicated on each side.

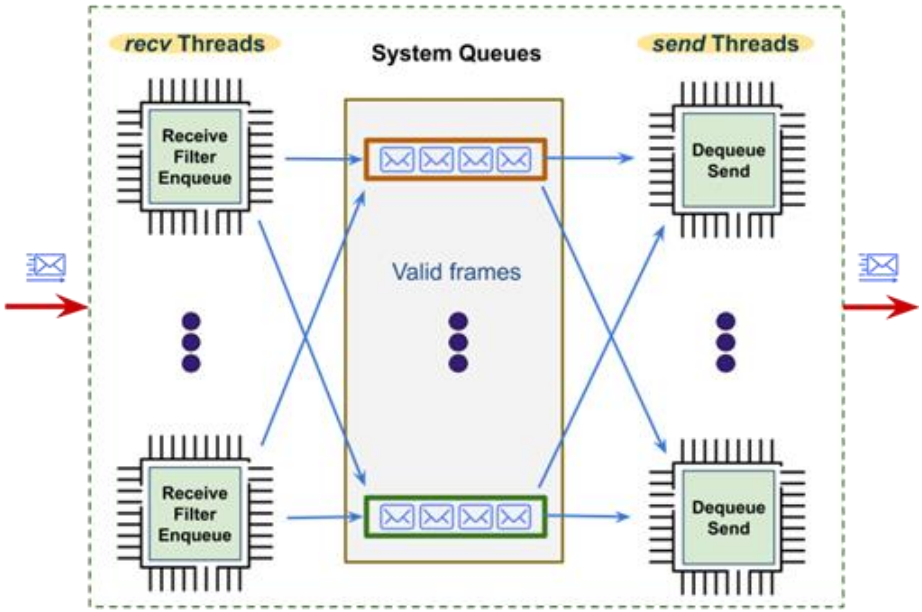


Figure 1.9. Inter-thread communication in the gateway.

Figure 1.9 examines inter-thread communication in the gateway. According to the figure, `N recv` and `N send` threads are responsible to control independently each CAN/Ethernet interface, i.e., `recv` threads receive and filter frames arriving from the incoming interface before enqueuing valid frames into a Queue. Simultaneously, `send` threads dequeue frames from the Queue and send them further via the outgoing interface. This system of queues facilitates inter-thread communication, and it can be implemented using shared parallel data structures.

1.3 Concurrent Queues

Traditional data structures use critical sections (on shared memory) for supporting concurrency, which is synchronized by mutual exclusion locks. Locks are widely used as they offer simple programming abstractions although they suffer from several weaknesses in a (possibly heterogeneous and faulty) distributed computing environment: (1) whenever a thread is delayed inside the critical section, all the other threads cannot make any progress, (2) only one thread can access the critical section at any given time, and (3) when there is a cyclic dependency between one or more threads, which causes them to wait for one another to finish, thus blocking all operations involved (deadlock) [9, 10].

To address these issues, lock-free data structures were proposed to allow concurrent access to threads. Lock-free synchronization was first introduced by Lamport in [11] to solve the concurrent readers' and writers' problem (Dijkstra's producer-consumer problem [12]). Numerous lock-free algorithms have been proposed for the most frequently used data structures (e.g., linked lists [13], queues [11, 14, 15, 16], sets [17, 18], heaps [19], and binary search trees [20, 21]).

While lock-free data structures assure system-wide progress (non-blocking algorithms), they do not guarantee per-thread progress, as well. Wait-free algorithms, proposed in the early 90s [22], provide solutions on how a thread can complete any operation in a finite number of steps, regardless of the execution speeds of the other threads.

For the current work, we compared three different concurrent queue implementations, in terms of average waiting time versus average injection/insertion rate (a traditional lock-based circular queue, a wait-free version of Lamport's single-produce single-consumer implementation, and Cameron's wait-free queue implementation) on an Odroid XU3 board to

select the most appropriate for our single-producer single-consumer concept. The results are shown in Figure 1.9.

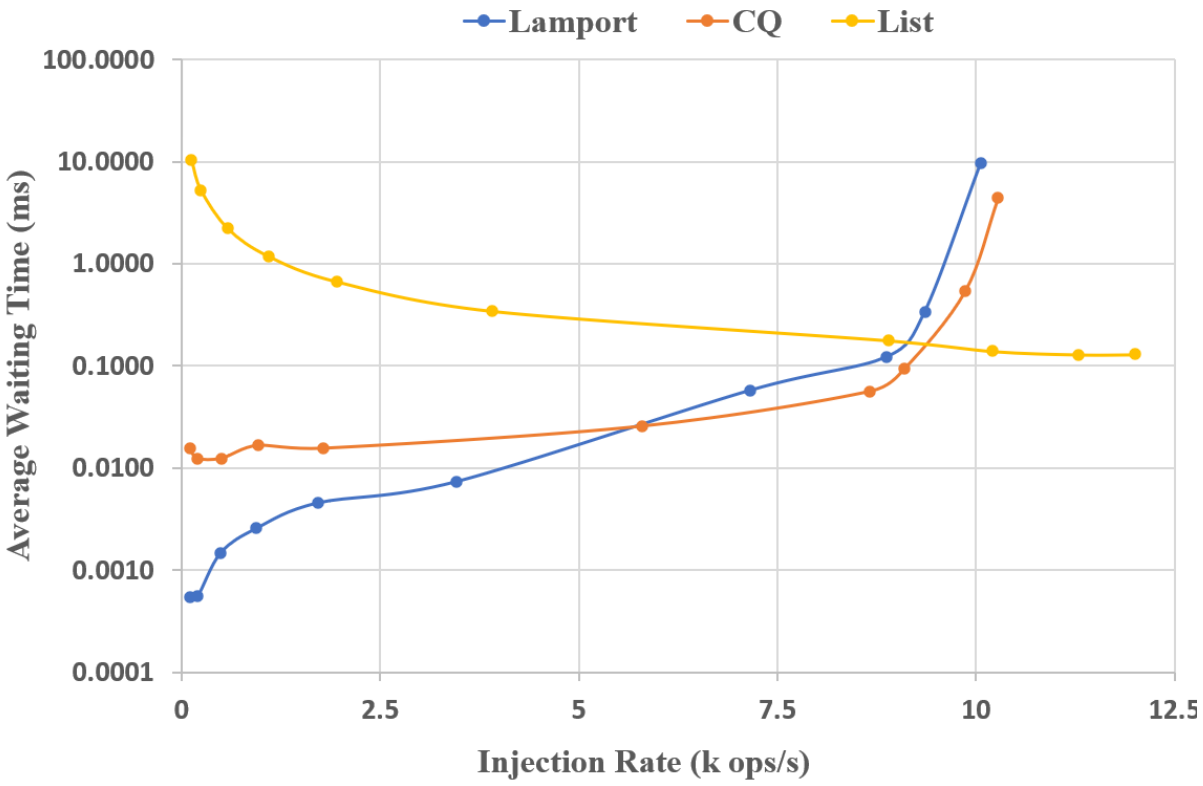


Figure 1.10. Queue performance at the Odroid XU3 gateway vs injection rate.

Lamport queue appears better for small rates, the Cameron queue (CQ) is better for medium rates, and List (traditional Single Producer – Single Consumer synchronization) is the most efficient for large injection rates. In general, it is not apparent how the average waiting time that an element stays in system queues of a complex gateway is affected by network limitations and interface delays. Thus, in our gateway implementations, we mainly focus on how the most viable competitors (List and CQ) fare when the gateway operates, and packets are exchanged.

1.4 Objectives

This work focuses on methods and implementations to optimize gateway scheduling in terms of (1) performance and latencies/rates, (2) scalability, (3) reliability, and (4) energy consumption using concurrent queues. For these purposes, we design optimized gateway

architectures for two applications: a healthcare gateway (Bluetooth-Ethernet) and a vehicular gateway (CAN-CAN). Both gateways aim to minimize latency (and other overheads, such as packet loss) using concurrent data structures, maximizing the use of hardware multi-core features and extending scalability. On top of the above, we try to make the system more reliable and energy efficient. For achieving those goals, we make use of a fast, single-producer, single-consumer wait-free queue implementation [23] to replace existing traditional circular FIFOs.

1.6 Thesis Outline

The rest of this document is organized as follows· Chapter 2 describes and evaluates our gateway implementation for transmitting and processing electrocardiogram data (ECG) from Bluetooth to Ethernet. In Chapter 3, we present a CAN-CAN gateway implementation in an automotive system along with its experimental framework. In Chapter 4, we state our conclusions, while we discuss some future directions in Chapter 5.

Chapter 2 - Efficient Gateway for Biosensors

Tiny wearable sensors monitoring physiological parameters assist the proliferation of e-health services [24]. Most of these sensors send biometric signals to a remote server or cloud center. The recipient stores, processes and analyzes patient data, typically with the assistance of a caretaker with medical expertise. Recent market research shows that more than 50% of existing commercial heart rate recorders such as BodyGateway (BGW) and BodyGuardian [25, 26, 27], CardiBeat [28], Bittium Faros 360 [29], D-Heart [30] and EC-12RM [31] support Bluetooth wireless communication, which consumes less power than Ethernet. Therefore, we need to support gateways that facilitate the exchange of information between sensors, gateways, and servers.

As mentioned before, lock-based data structures usually suffer from synchronization overheads. On the other hand, lock-free (or even better, wait-free) data structures have less synchronization overhead for shared memory accesses than lock-based implementations. Therefore, in this work, we investigate the feasibility of designing robust and energy-efficient real-time gateway architectures implemented either by conventional circular FIFOs or concurrent queues.

More specifically, our research focuses on gateway architectures that manage complex BT-to-Ethernet protocol stacks. In particular, the gateway uses `rfcomm` to communicate with BGW impulse devices over Bluetooth, send command packets to program BGW devices, provide reliable data collection of raw BT packets, and capture critical signals and send it over Ethernet to the server.

Our gateway emulates a single-producer single-consumer bridge between Bluetooth and Wireless. Its implementation uses either a lock-based circular FIFO (called list) or a concurrent queue (CQ) to store received BT packets. These data structures are important in terms of performance but have not been investigated in previous studies.

Experiments with high-rate ECG (up to 4096 pulses/s) focus on file servers to collect data before performing ECG monitoring, analysis, or visualization. These experiments show that we sustain soft real-time behavior at the server; note that higher rates above 256 pulses/s are not supported by the biosensor but are probed by the self-similar digital twin sensor process. Further

experiments examine the CQ or list waiting time delay at the gateway for different biosensor rates.

Existing state-of-the-art implementations related to efficient ECG signal processing for filtering, smoothing, and arrhythmia analysis have been presented in [32, 33]. Open and commercial Android frameworks that provide a BT-to-Ethernet gateway have also been described in [34, 35]. Although our work is more similar in its foundation to [36], which deals with the receipt, conversion, and transmission of data packets between BT and Wireless. Our focus is on efficient real-time processing, rather than interoperability.

2.1 Driver Architecture

We used a single-lead, wearable patch (ST Micro BodyGateway, or BGW), supporting Bluetooth connectivity, very similar to Boston Scientific's BodyGuardian mini/mini plus. Bodyguardian mini has been used to record data for almost 1 million patients per year in the USA.

An Odroid XU4 board acts as a gateway to manage a complex BT-to-Ethernet (IEEE 802.11) protocol stack using four software components [37,38,39], as Figure 2.1 shows.

- The *BT Connect Function* uses *bluez* tools to pair with the BGW device. Similarly, a *TCP Connect Function* connects to the server.
- Then, the *BT Writer Thread* transmits command packets to configure and request periodic ECG data from the BGW device, i.e., up to 256 samples of ECG data per sec (with 12-bit precision); notice that the BGW device can also support the transmission of bioimpedance and accelerometer data.
- The *BT Reader Thread* operates in a loop. It first uses the `rfcomm` protocol to connect and periodically receive raw Bluetooth data at the specified rate. Then, it extracts the ECG signal from the Bluetooth packet via bit operations, considering ARM's big-endian architecture. Finally, it inserts signal data either to a traditional producer-consumer lock-based "circular FIFO" (or a concurrent queue), as shown in Figure 2.1.
- The *BT-to-Ethernet Sharing Thread* periodically dequeues all available biometric data in the concurrent queue (or resp. the "circular FIFO" of the list) and transmits them in real-time to a file server (an Odroid XU3 board) when their number exceeds the current rate;

this optimization helps improve the waiting time. An isolated 2.1 Gbit/s TP-Link Archer C5400 router is used for the Ethernet connection between the gateway and the server.

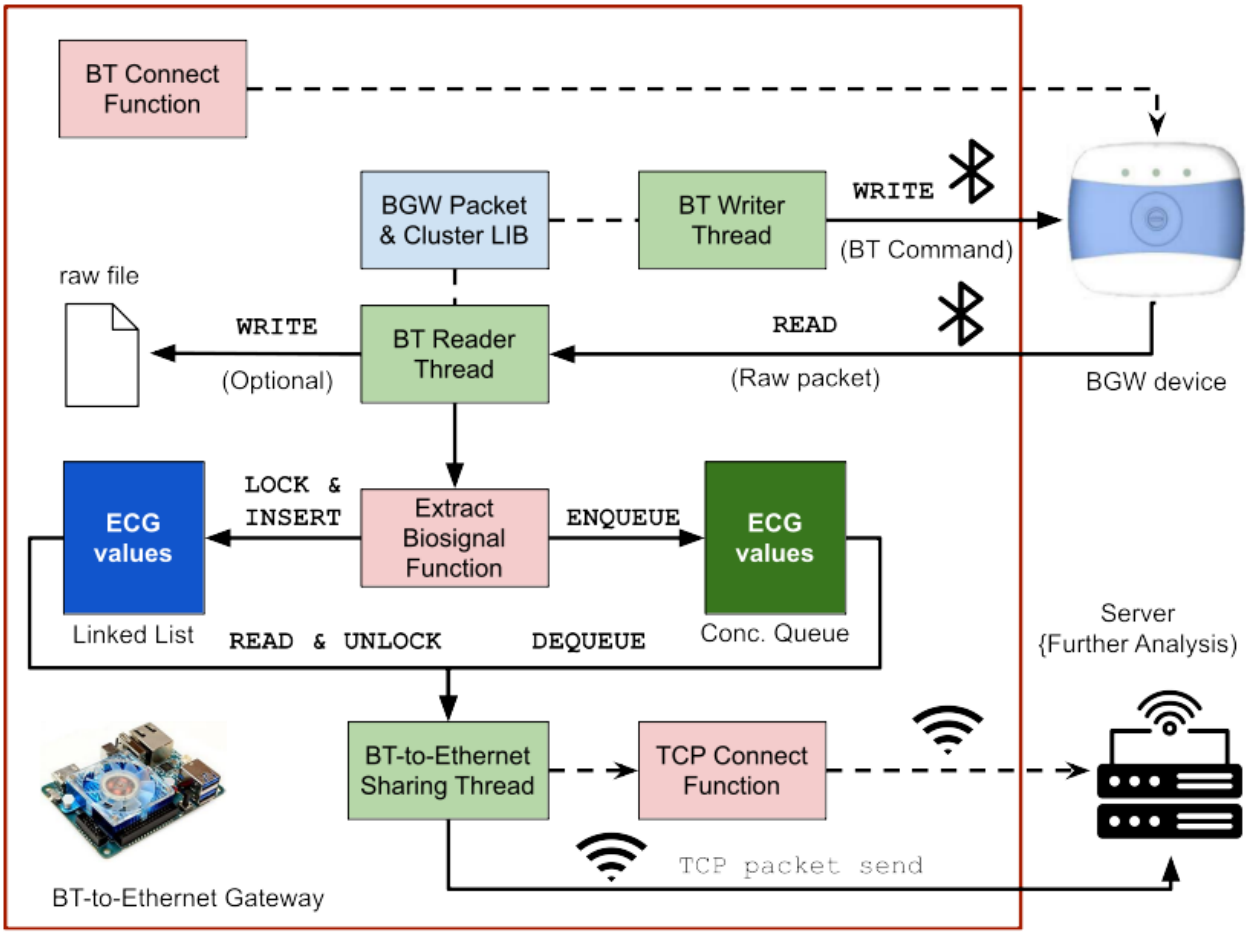


Figure 2.1. The gateway architecture with a complex BT-to-Wireless protocol stack.

2.2 Methodology and Optimizations

The first idea was to replace the circular FIFO with a single-producer single-consumer concurrent queue. Apart from that, we decided to locate time-consuming modules which could be more optimized and could improve the overall driver performance. Hence, to achieve that goal, we

1) split the existing C/C++ code into single modules according to their functionality,
2) monitor each of them separately, and
3) provide alternative optimized implementations for selected modules.

2.2.1 Optimizations

Several minor optimizations were made; however, an important module is recognized as time-consuming: the extraction and transmission of values to the server over a TCP socket. In this module, *BT-to-Ethernet Sharing Thread* repeatedly removes values from the FIFO and concatenates them to a TCP packet that is being constructed at the same time. Once no more values are found on the FIFO, the thread writes the TCP packet to the socket and starts constructing a new one.

We discovered that when the production rate is higher than the consumption, the TCP packet becomes very large and delays the sending process. Moreover, the TCP overheads gradually increase the remaining time on the data structure, as a result, an overflow is caused after exceeding the queue size limit. Additionally, constant small overheads are added in the case of a higher consumption rate, a fact that significantly delays the thread's execution.

Addressing the issue above, we limit the minimum and the maximum TCP packet size depending on the ECG rate of the sensor (e.g., for a rate of 128 pulses/sec the number of the values in each packet is 128). For higher rates, beyond 512 pulses/sec, the packet contains 512 values in order not to exceed the memory bounds.

2.2.2 Digital Twin Concept

A Digital Twin [40, 41] is a virtual replica of a real object or system, which can digitally represent a physical object with all its properties. Digital twins are important methods because they enable studying the behavior of a physical object in conditions that cannot be supported in real life.

Since BGW does not support ECG rates higher than 256 pulses/sec, we developed a self-similar, digital twin of BGW (DTBGW) for ECG rates between 512 and 4096 pulses/sec to further investigate the performance of our implementation for higher ECG rates.

DTBGW implementation was done in three steps: (1) periodical feeding of the *BGW Reader Thread* with packets identical to the real BGW packets, (2) developing a delaying

mechanism to ensure the requested ECG rates inside *BGW Read Thread* and *BT-to-Ethernet Sharing Thread* functions, and (3) the retainment of all the remaining functionality (run-time initialization, configurations, etc.) as they were at the original system.

We bypassed the BT socket buffer using a mechanism to randomly feed *BGW Reader Thread* with real ECG packets from a dataset taken from previous runs. The packets are streamed every N ns, where N results from a function that measures the current time and calculates the remaining delay that should be applied (`usleep`) to retain the requested ECG rate in *BGW Reader Thread*.

2.3 Experimental Results

The optimized implementation, which was based on the concurrent queue, was evaluated for its performance on an ARM7 Odroid XU4. For each implementation (list and CQ), we run numerous tests considering different sensor transmission rates. The only supported rates for the BGW sensor are 128 pulses/sec and 256 pulses/sec. For higher ECG rates (above 256 pulses/s), we used the self-similar, digital twin sensor process. Another parameter we tested, apart from the rates and the data structures, was the locality of the server and the client.

2.3.1 Gateway Metrics: Average Waiting Time vs BGW ECG Rate

Focusing on the first evaluation scenario, we assume that both the client (driver) and the server, which receives and presents the data, run simultaneously on the same device (Odroid XU4). In Figure 2.2, we present the performance results of our implementations on six different ECG rates.

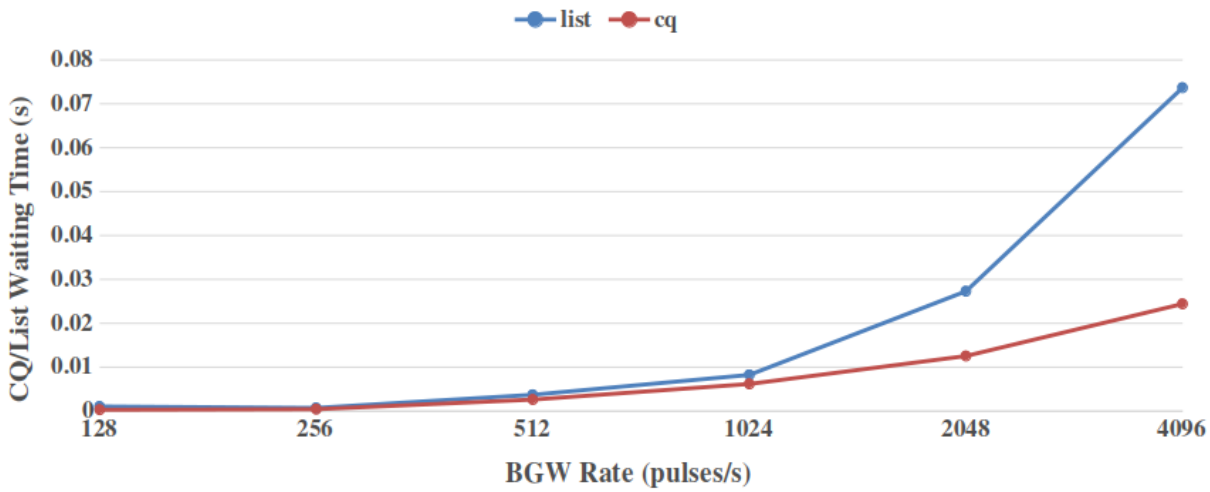


Figure 2.2. Waiting time in the CQ or the list (circular FIFO) vs the BGW rate, both running simultaneously on the same device.

Herein, we measured the average time that a packet remains on each data structure until the consumer removes it. For small BGW rates (up to 256 pulses/s), CQ and List have a similar waiting time. In this case, both implementations manage to extract packets immediately, without accumulating them in the queue or circular FIFO. However, CQ outperforms the list for high BGW rates (above 512 pulses/s); these rates are obtained using our digital twin model.

Next, we used an Odroid XU3 as a server and measured the performance of the BGW driver. In the new setup, the driver transmits ECG values over Ethernet to the server. Figure 2.3 shows the new metrics.

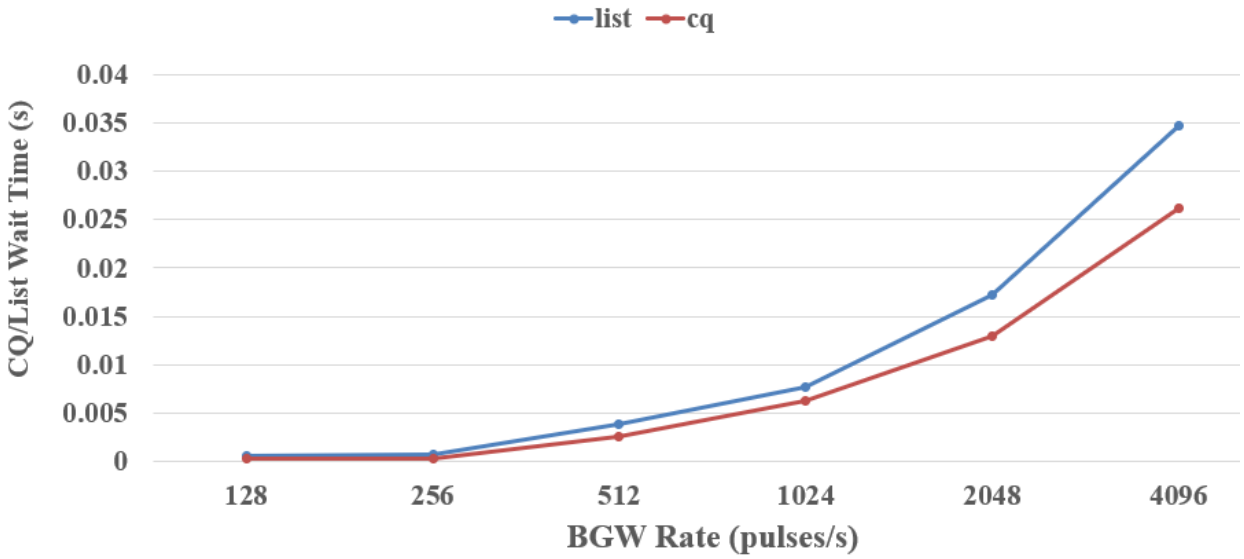


Figure 2.3. Waiting time in the CQ or the list (circular FIFO) vs the BGW rate, both running on different devices.

The performance, in this case, follows a similar manner. The remarkable point in this figure is the decreased average waiting time per ECG rate in comparison with Figure 2.2. Running the server application on a different device boosts the driver's efficiency.

2.3.2 File Server Application Metrics: Instant Pulse Rate vs Time

This experiment examines the server application performance when the BGW heart rate sensor transmits over 128, 256 pulses/sec (via the digital twin model). Figures 2.4.a and 2.4.b examine the instant sample rate of 128 pulses/sec (or 256 pulses/sec) versus time. At lower rates, large instant rate variations are observed because the BGW transmits pulses in bursts (up to 128 samples in a single burst).

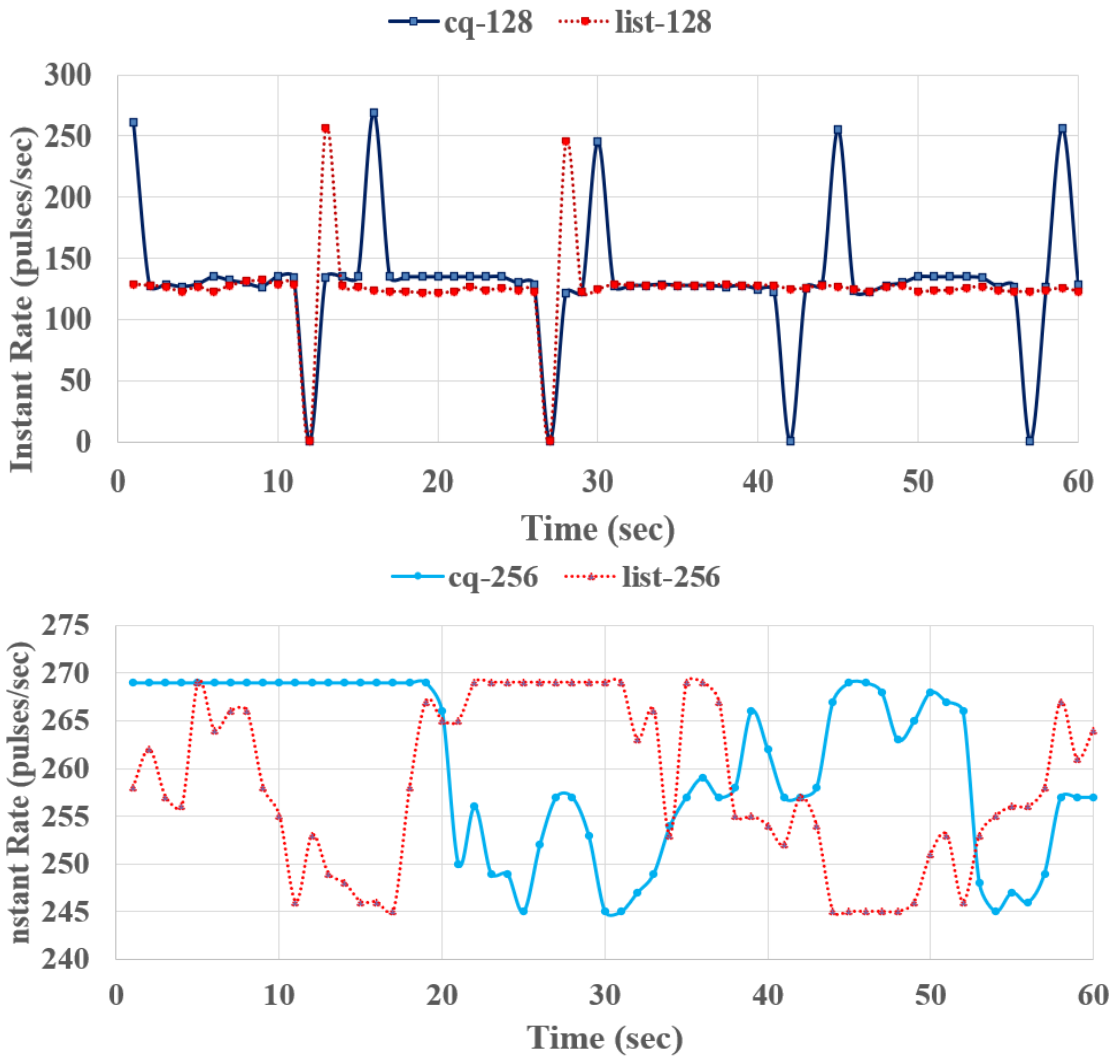


Figure 2.4. The instant arrival rate of ECG samples at the server for CQ and list, when (a) BGW is set at 128 pulses/s (up), or (b) 256 pulses/s (down).

Figure 2.5 shows that for higher rates (1024 pulses/sec and above), implemented using the digital twin concept, the instant rate has a relatively small statistical variation from the mean value. This is due to the increased number of samples per unit of time.

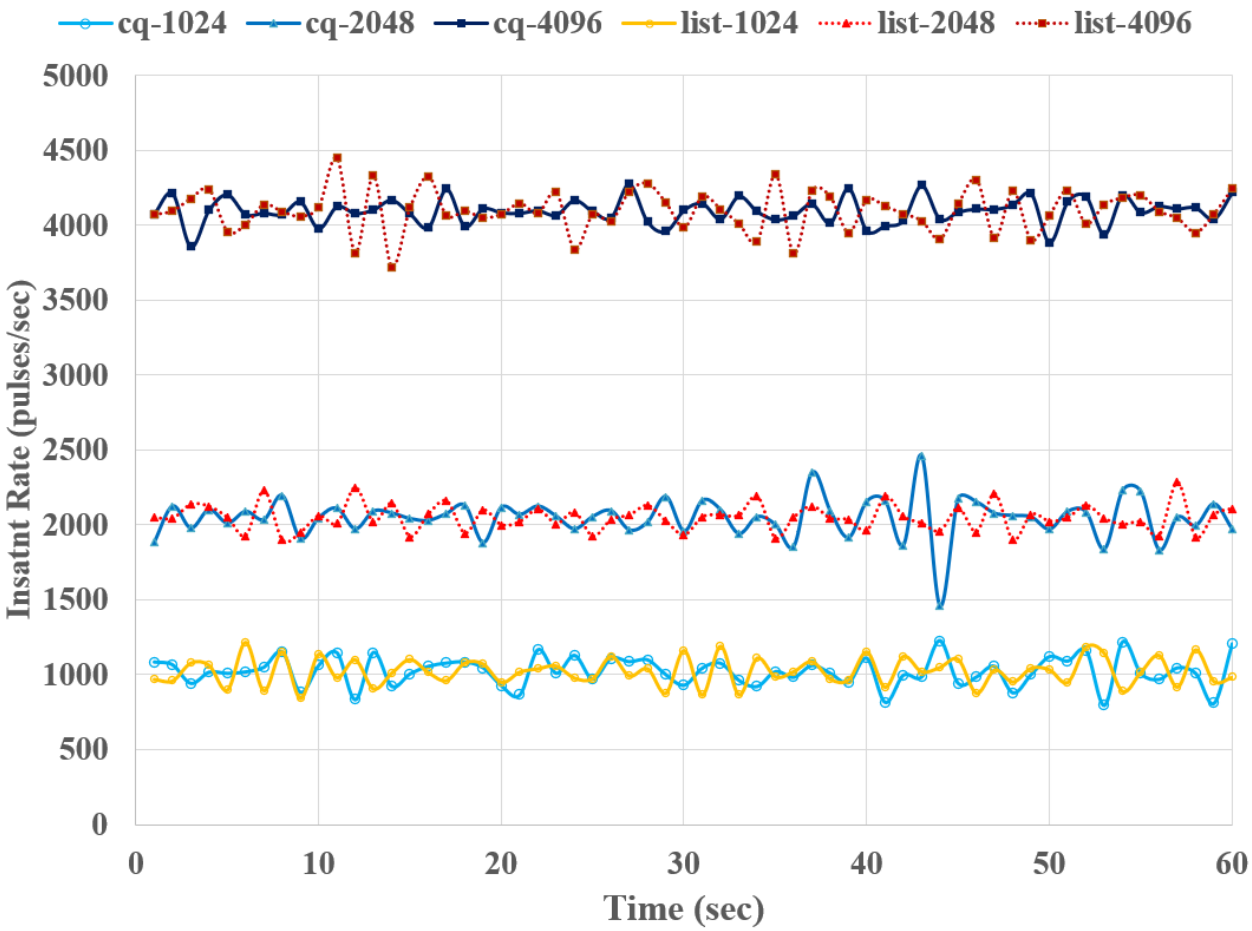


Figure 2.5. The instant arrival rate of ECG samples at the server, for CQ and list, is shown; higher BGW rates (1024, 2048, or 4096 pulses/s) are based on a normalized digital twin model.

2.3.3 File Server Application Metrics: Cumulative Average Pulse Rate vs Time

In Figures 2.6 and 2.7, we examine the file server application and compute the cumulative (running) average rate at the server, using the number of ECG samples processed and the inter-arrival latency. As shown, during the 60-second period of measurements, both implementations (CQ and list) sustain the BGW rate in real-time: usually in 1-2 seconds, or at most ~10 sec for CQ when the BGW sensor is operated at 128 pulses/s.

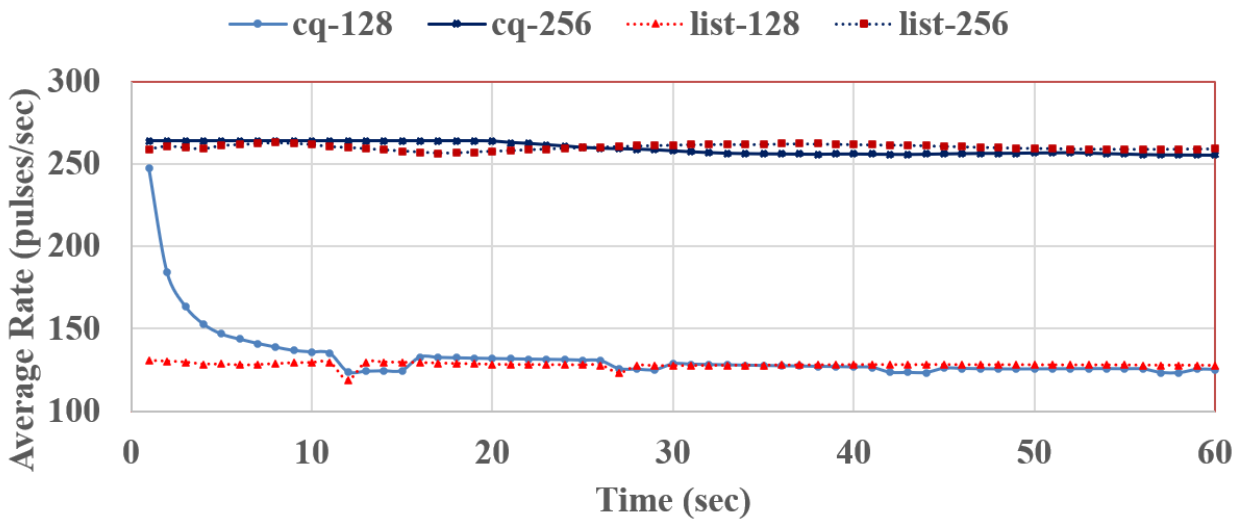


Figure 2.6. The cumulative average arrival rate of ECG samples at the server for CQ and list, when BGW is set at 128 pulses/s, or 256 pulses/s.

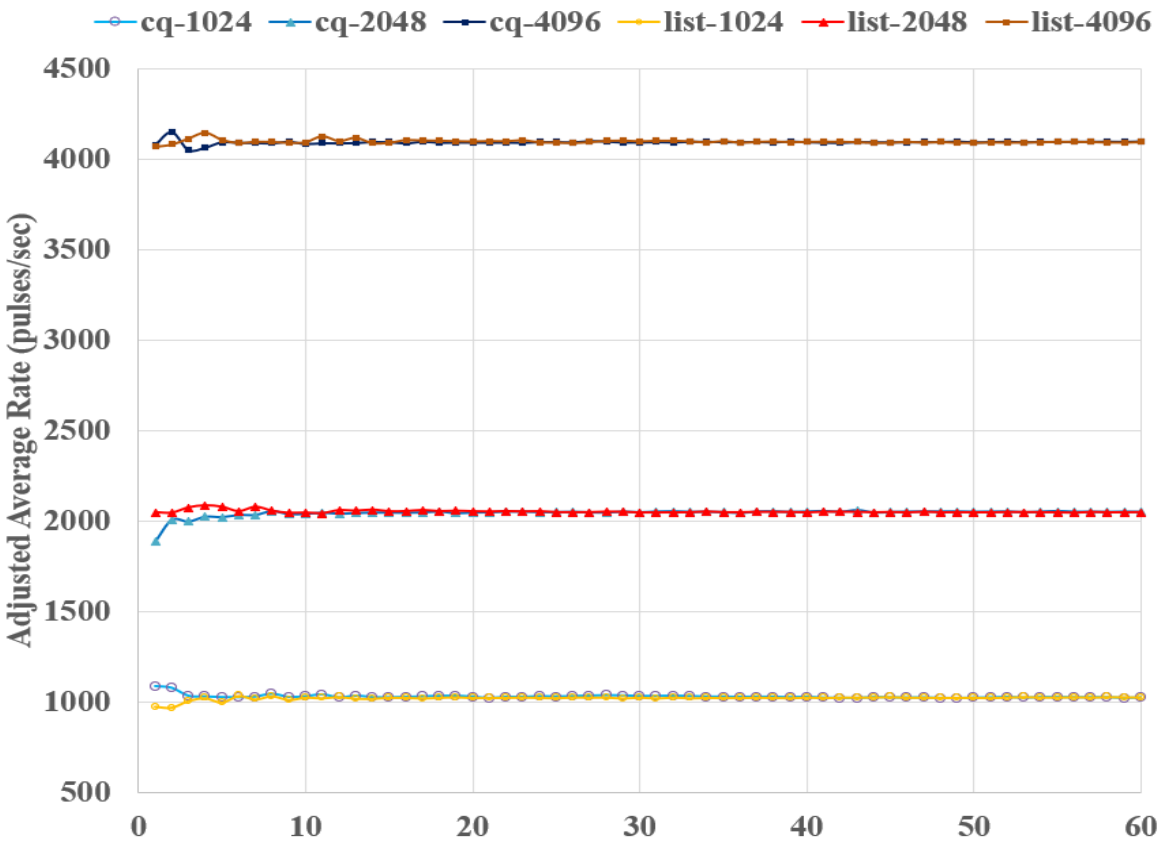


Figure 2.7. The cumulative arrival rate of ECG samples on the server for CQ and list. When using the normalized digital twin concept, the BGW rate is set to (1024, 2048, or 4096) pulses/sec.

2.3.4 Energy Consumption

We used Odroid XU3's INA231 sensors to profile energy consumption in the gateway and the server. Both queues have very low energy costs (around 0.20 Watts on average, with a rare peak of 0.6 Watts). Figure 2.8 shows the measurements of the live energy monitoring at two different rates.

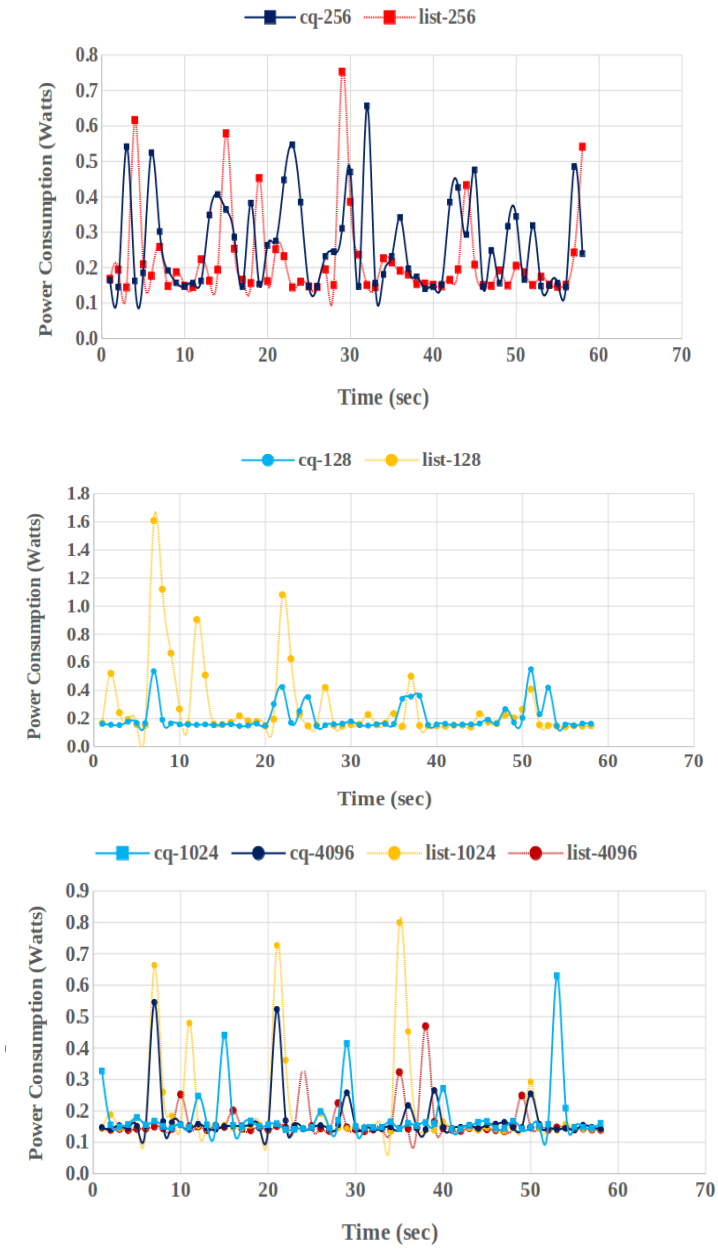


Figure 2.8. Energy consumption (in Watts) for CQ and list at selected rates.

Chapter 3 - Efficient In-Vehicle Gateway

Modern vehicles are equipped with numerous devices, sensors, and actuators to improve occupants' safety and experience. Some devices are connected to the internet, while all end devices and the Engine Control Unit (ECU) are interconnected through different wired or wireless networks (such as CAN, BT, etc.). The devices receive messages from sensors or other devices, analyze them, and send messages back to actuators. During a one-hour car ride, millions or even billions of in-vehicle messages could be exchanged with different priority levels, depending on the technologies used on that vehicle.

Due to that increasing demand for in-vehicle traffic data for supporting new functionalities, automakers connect multiple CAN buses via a central gateway. Automotive gateways aim to transfer messages from one communication bus to another, possibly filtering (intercepting or altering messages).

While in the old days, CAN-based communication was through an independent ECU (often called a BCM integrated gateway), currently, in most vehicles, a Domain Centralized Electrical/Electronic Architecture (central gateway) is used. This gateway further connects to a few domain controllers, such as powertrain, transmission, chassis, interior, infotainment, telematics, and diagnostics.

Recently proposed high-end automotive solutions require a centralized E/E architecture [42, 43]. A central gateway (many-core CPU) is connected via OBD with many zone controllers via 1000BASE-T1 Ethernet, enabling extensive telematics services and smart Cloud-based solutions. Zone controllers act as smaller in-vehicle gateways to distribute data & power based on the physical location of ECUs that control vehicle sensors and actuators. This new approach, although not fully implemented, reduces wiring costs, and enables safer direct communication from high-bandwidth sensors and actuators to central components.

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. Controller Area Network (CAN) is the most popular in-vehicle network [44, 45]. 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.

One or more gateway devices are often placed as intermediate agents to collect, analyze and distribute data from/to those end devices interconnected to the CAN bus. GW devices play an important role in the whole process, as they can filter messages in terms of security, safety, and error detection.

In the current scenario, we investigate the efficiency of a concurrent queue as an in-gateway buffer between the producer and consumer threads, and how it could impact the total performance. We examine concurrent packet transport mechanisms using traditional producer-consumer solutions (bounded buffer problem) and single-producer, single-consumer concurrent lock-free queues. producer-consumer philosophy across multiple CAN networks

3.1 Software Stack

We utilized an experimental embedded system platform that went through several phases of upgrades. Figure 3.1 shows 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 only two CAN buses and three CAN nodes, it can be extended to include 4 CAN networks and tens of CAN nodes. Below we describe the architecture of these nodes:

- An **Odroid XU3/XU4** acts as an intermediate node, a gateway. It runs Ubuntu 18.04 LTS (Linux kernel 4.14 with `preempt_rt` patch is used to target embedded systems with latency requirements in the `us` to `ms` time range). Odroid XU3/XU4 has a 2GB DDR3 and uses the Samsung Exynos 5422 chipset with Big-Little CPU architecture (Cortex A15 quadcore and second smaller Cortex A7 quadcore).
- Two **Raspberry Pi 3** version B nodes running 2019-04-08-raspbian (Linux kernel 4.9 with `preempt_rt` patch) are used as representative nodes for the non-critical CAN (dashboard sender) and the critical CAN network (engine receiver). We used IndustrialBerry's Canberry Dual v2.1 shield to configure a Raspberry with 4 independent CAN interfaces. However, only two of them are used in this use case.

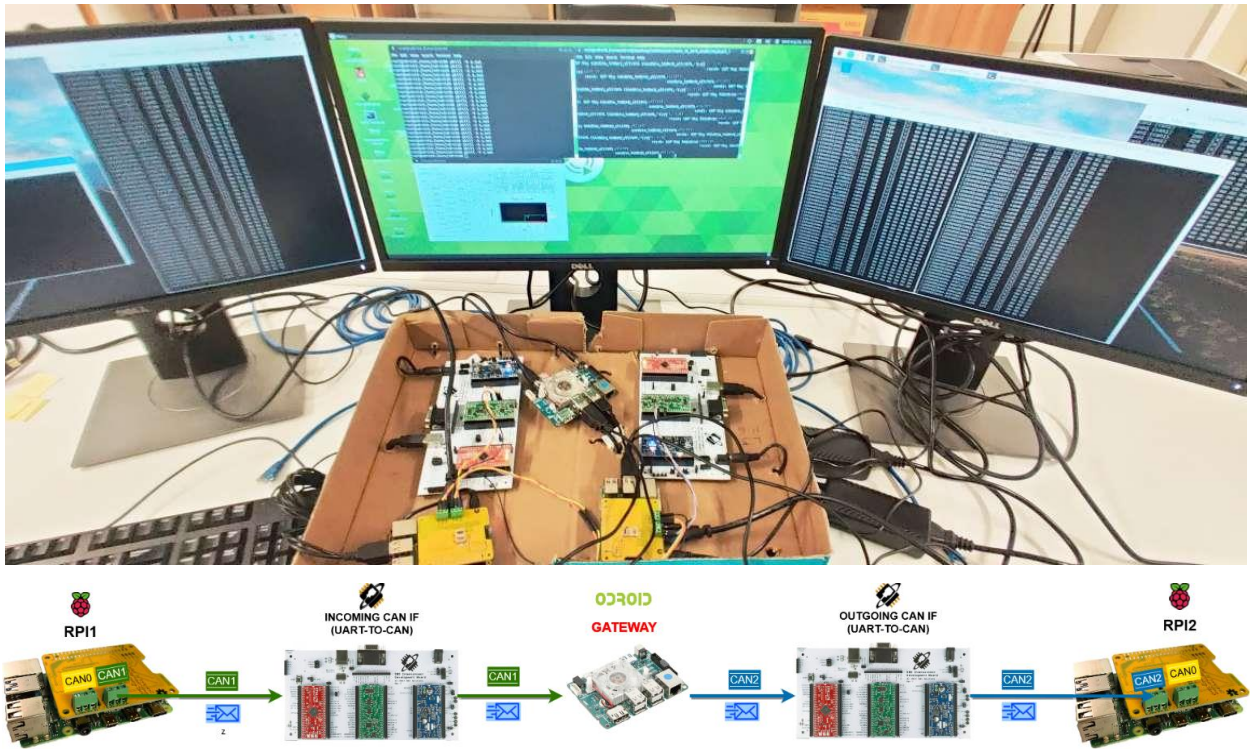


Figure 3.1. A prototype of our open distributed embedded platform. ECU components are based on Raspberry Pi with Canberry SPI-to-CAN interfaces, while the gateway is an Odroid XU3 with UART-to-CAN interfaces (via OBD development kit) [46]

Odroid XU3/XU4 is connected to CAN via *two Scantool OBD dev kits* [47]. Each kit supports a one-way (incoming or outgoing) USB-to-CAN interface using ELM327 and specialized STN2120 software capabilities (i.e., serial AT and ST commands). Commands related to interface configuration, for example, a) "`STP31`" (or "`STP33`") for RAW CAN (or ISO 15765 OBD-based protocol), b) "`STSBR2M`" to set the baud rate (2Mbps baud rate is required for a CAN rate of 500Kbps), or c) "`STWBR`" to write permanently the serial baud rate. Other commands relate to sending a CAN message, for example, "`ATSHA`" to change CAN ID (header of the message) or receiving a CAN message, e.g., "`ATMA`" to set flow control, pass, and block filters. Access to various CAN interface on the OBD SDK, e.g., DB15, two RX/TX and two CAN High/Low, is useful for prototyping on our platform. Another key component of the *OBD development kit* is its power module, which provides an extremely fast sleep/wake mechanism (up to 20 ns), crucial for suspending interface actions.

Our embedded application software toolchain runs on the above 3 nodes as follows.

- For the Raspberry PI 3 nodes (sender and receiver), built-in Linux kernel modules support Linux CAN utilities (`can-utils`) to send, receive or control messages on the Canberry shield via `cansend`, `candump`, `canplay`, `cangw`, `cansniff`, etc (see [48]). Note that the time between arrivals can be measured on the sender using the Linux `candump -t a` command.
- Our open-source embedded software code on the Odroid XU3/XU4 gateway extends serial terminal (TTY) code, like minicom in [49]. The code (~25,000 lines of C code) uses two POSIX threads in a single-producer single-consumer (bounded buffer) synchronization pattern to control incoming and outgoing Rx/Tx traffic from each UART-to-CAN interface. These two threads can receive CAN messages from `CAN1`, process them, and respectively send them to the receiver device. Thread access to the shared circular queue (bounded buffer or CQ) that stores pending (unprocessed) messages are synchronized either with POSIX mutexes or lock-free/wait-free methods. The receiver fills and the sender concurrently empties this queue. The queue size is chosen during configuration to store many CAN frames (header and data). As previously mentioned, the two USB-to-CAN gateway interfaces are configured and controlled using STN2120's ELM327 AT, and ST commands. Figure 3.2 shows the process described below.

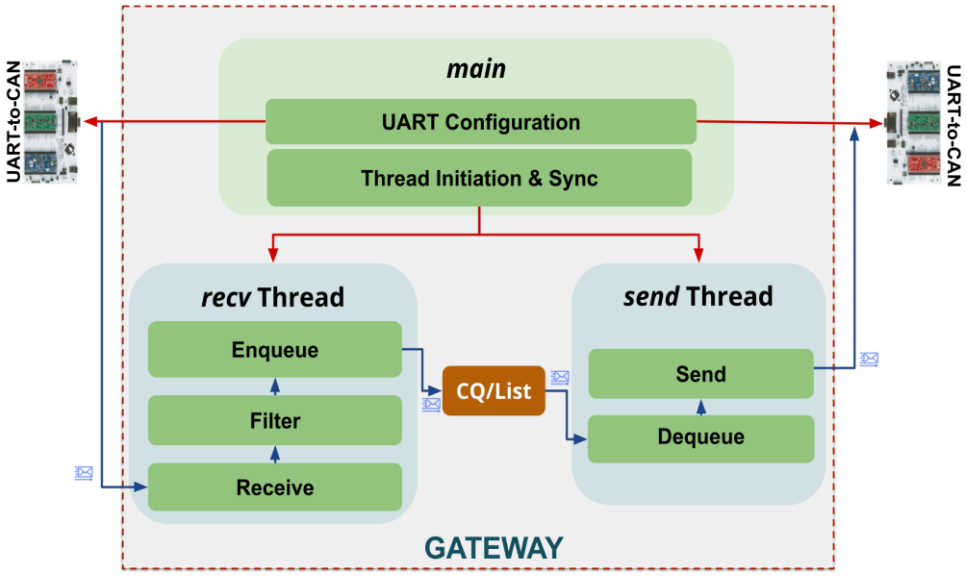


Figure 3.2. The embedded software stack at the gateway. Threads compete for the shared buffer; the architecture is extendible to many (incoming/outgoing) interfaces.

3.2 Methodology & Optimizations

We identify three criteria that can optimize the total performance of the above system:

1. the replacement of the data structure by a single-producer single-consumer (SPSC) concurrent queue, which might eliminate the delays between the threads in high injection/output rates,
2. the detection of the optimal delay time to write on the CAN bus (via UART) during transmission from the gateway to the receiver, to decrease lost and corrupted packets,
3. code optimizations in thread functions that could speed up execution.

For achieving the goals above, we have also used the SPSC concurrent queue implementation, described in Chapter 2. By such a change, we hoped that the lack of locks and barriers would improve the real-time functionality of the whole system and would increase the total complete packets sent to the receiver. For point 2, we measured the performance of the system in different UART inter-character delay values to determine which is the optimal one (see §3.3.). For our last goal, we optimized loops and memory accesses/bindings without changing the logical flow.

3.3 Performance Evaluation

Using the above experimental framework, we examine the efficiency of frame exchanges at the gateway by considering input parameters, such as injection rate (ingress frame rate), and performance metrics, such as packet loss, output rate, and queue waiting time. For a fair comparison, we leave all other CAN parameters identical, such as cable length, bus load, frame density, and CAN bit rate. In addition, we consider power consumption metrics related to Cortex-A15 tiles.

We evaluate platform performance using different experiments, considering frame loss, egress frame rate, waiting time, and energy consumption. Next, we describe each experiment separately.

3.3.1 Detecting the Optimal 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. Our experiments indicate that frame loss depends on the inter-character timeout of the UART-to-CAN outgoing (sender) interface leading to the RPI2 (end-node). Notice that frame loss corresponds 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 (8-byte data) diagnostic frames that the end node knows how to handle according to specifications.

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

Since for a speed of 500k frames/sec, the serial port is set to 2 Mbps (with 8N1: 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) = 5 us. Therefore, inter-character timeout must be larger than 5 us.

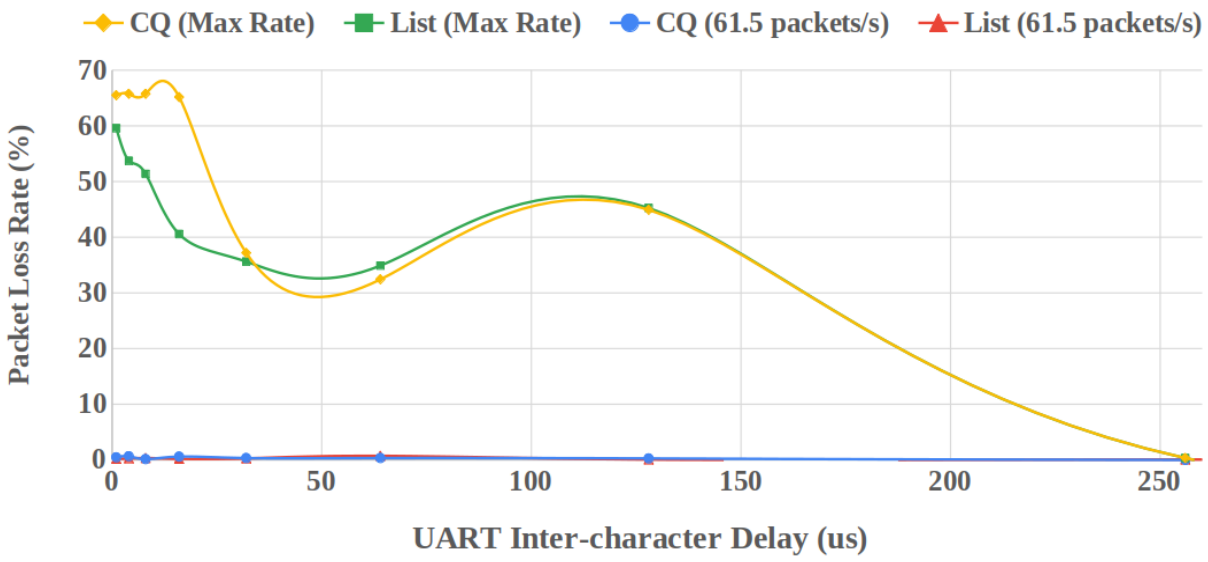


Figure 3.3. Packet losses on different UART inter-character delay values for two different rates (192 and 61.5 frames/sec).

In Figure 3.3, we examine frame loss at the outgoing link of the Odroid XU3 gateway for a wide range of UART inter-character delays. We consider CQ and List 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 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 on the RPI2 for small inter-character delays, especially below 5 us; however, for large UART inter-character delays above 256 ns, 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 5 us, as predicted. However, as the UART inter-character delay increases further, we may violate the real-time functionality in the CAN network, since this 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 [50], 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 messages in automotive communications.

In conclusion, the optimum delay value between characters seems to be at 256μs, as, on the one hand, both data structures show packet losses near 0% (0.325%) at the maximum rate, and on the other hand, we had no packet losses at 61.5Hz. However, that value can accumulatively violate the real-time functionality of the system after receiving thousands or millions of successive packets.

3.3.2 Queue Performance on the Odroid XU3 Gateway

In this experiment, we aim to characterize different queue performance metrics by examining the average CAN frame waiting time in the queue. More specifically, we measure the elapsed time interval from enqueuing to dequeuing when CAN-to-CAN communication is in place at the gateway. We consider different injection rates, from 30 to 973 frames/s). In all experiments, UART inter-character delay is set to 8us.

As shown in Figure 3.4, the average waiting time for both CQ and List have similar behavior in low and medium frame injection rates. However, the CQ implementation performs slightly better (~2.2%) for rates higher than 730 frames/second. This extra slack time can help perform slightly more sophisticated frame filtering.

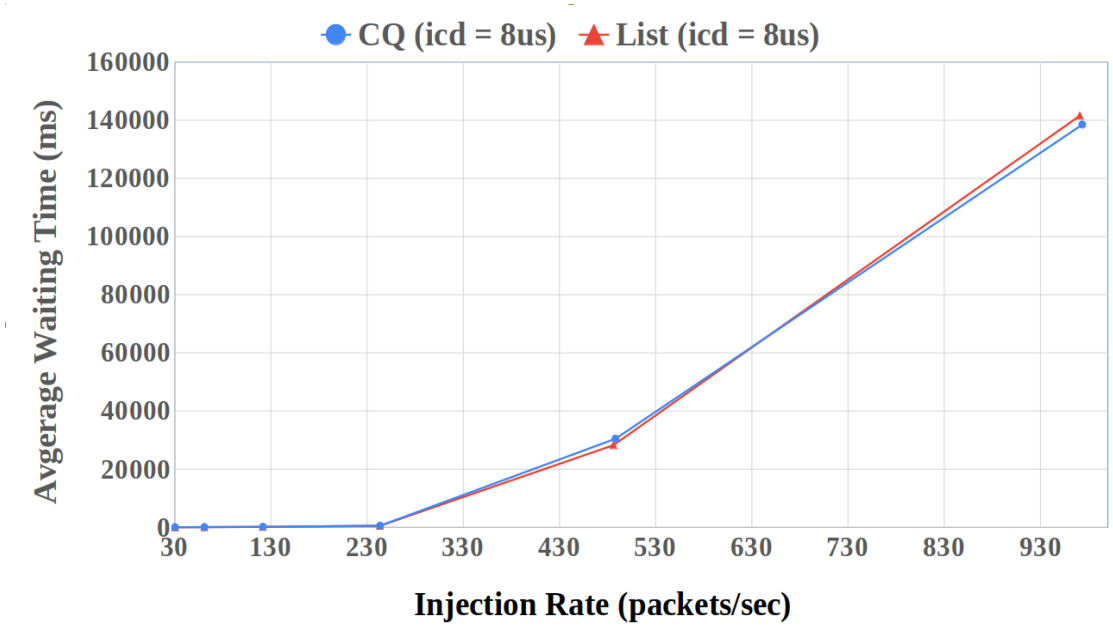


Figure 3.4. Average packet waiting time at various injection rates for CQ and List. The inter-character delay is set to 8us

In our next experiment, we focus on examining the gap between injection and output rates for large injection rates. Figure 3.5 shows the output rate vs. the frame injection rate, considering two different UART inter-character delay values: 4 us and 8 us. Notice that before saturation, List and CQ perform well, reaching a rate of almost 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 seems to slightly outperform CQ.

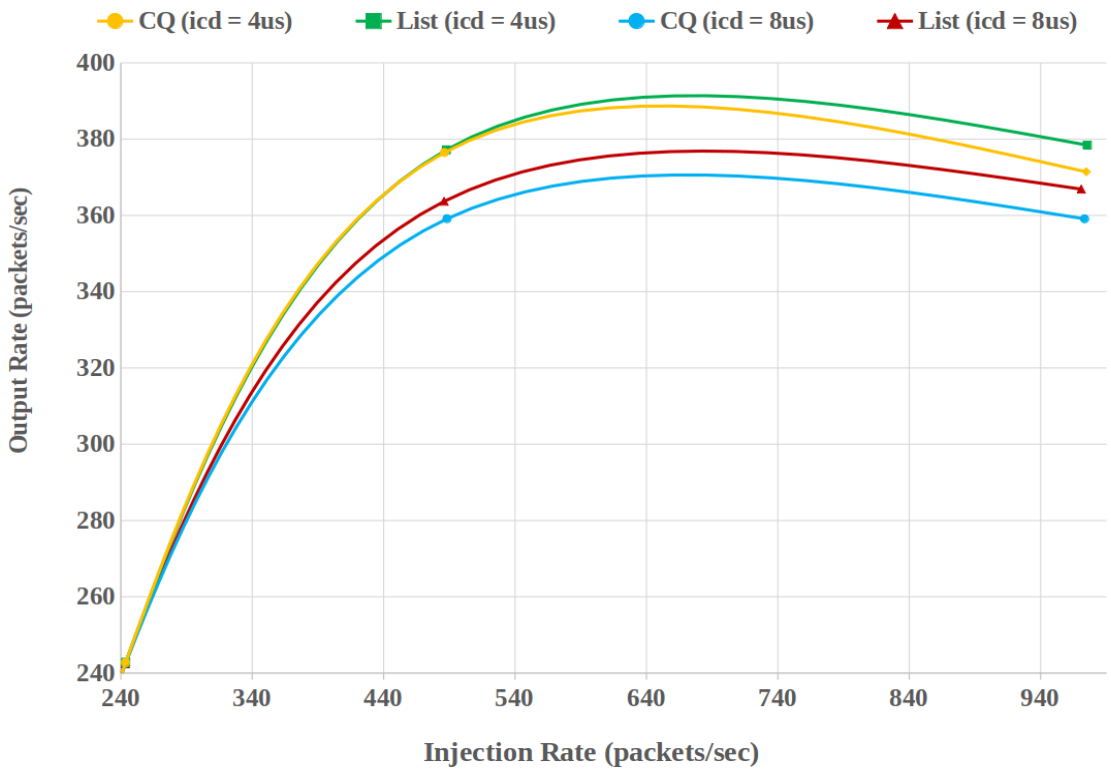


Figure 3.5. Correspondence between injection and output rate for each implementation

Finally, Figure 3.6 examines frame loss vs injection rate, considering two different UART inter-character delay values: 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 high especially for CQ (between 2.0% and 26.5%) when the UART delay is set to 4 us.

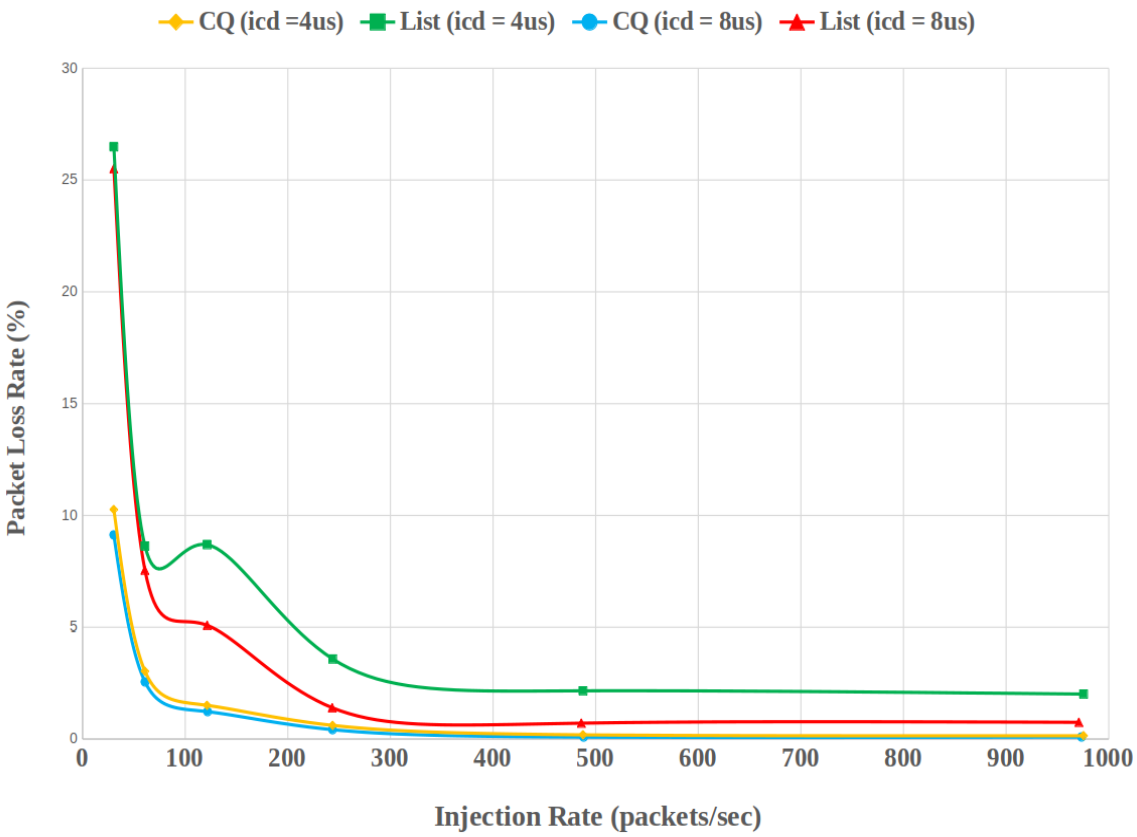


Figure 3.6. Percentage of packet losses vs injection rate.

3.3.3 Energy Consumption

For power estimation, the Odroid XU3 gateway integrates four TI INA231 current-shunt and power sensors equipped with 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 [51]. The driver is configured so that on-chip sensor configuration and data registers can be sampled from within our CAN Gateway application running on the A15 cores via `sysfs` file system entries [52]. This allows us to visualize all on-chip sensors (including thermal) by adapting the open-source Qt-based Energy Monitor [53]; another compatible power tool that enables per-process power profiling is ALEA [54].

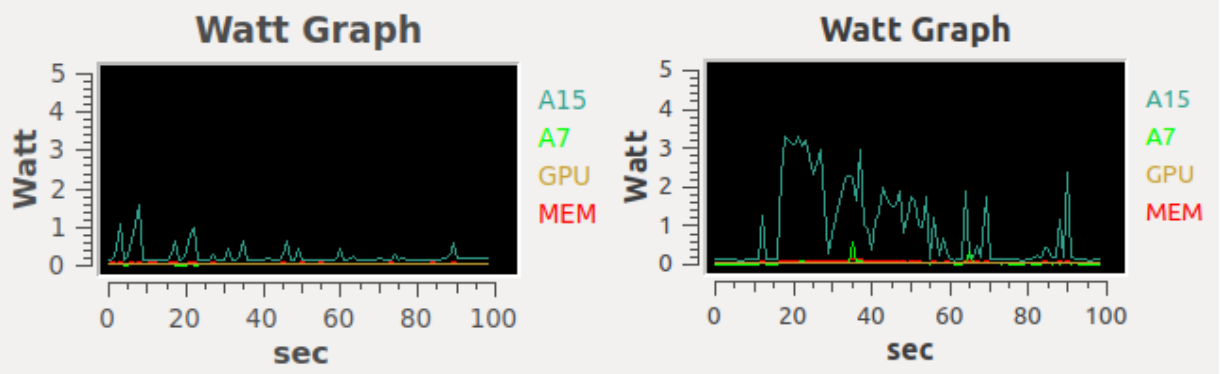


Figure 3.7. Typical power profiles from the Energy Monitor for a) List (left) and b) CQ (right), assuming a UART inter-character delay of 8 us.

Power metrics rise as more packets enter the Gateway. The I2C driver supports an update rate of 200ms (or higher depending on configuration), however, by averaging every instant value, we can obtain average power consumption for each cluster (big or little) once per second. Moreover, it is possible to view power/thermal graphics using the architecture-independent Qt-based Energy Monitor.

Figure 3.7 (a) and (b) show CPU, memory, and GPU power consumption for List and CQ. Notice that only A15 power consumption is significant since threads *recv* and *send* execute on two Cortex-A15 cores and mainly access registers and cache. In addition, observe that the CQ is much more power intensive (~1.6 Watt/s more) than List. CQ is a wait-free non-blocking concurrent data structure with more complex synchronization mechanisms, while 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.

Finally, notice that 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.

Chapter 4 – Conclusions

Internet of Things (IoT), an emerging trend nowadays, consists of numerous sensors and actuators, which intercommunicate over several heterogeneous networks and produce a huge amount of data. Transmission, manipulation, and analysis of the increasing data size are difficult to manage on the server. Gateways are commonly used to lighten server workload as they can synchronize heterogeneous networks, process the data, control the actuators, and feed the server with a subset of the data required.

In this thesis, we develop two different gateway applications, study their architecture, and evaluate performance using embedded system prototypes. We focus on both Bluetooth-Ethernet and CAN-CAN/Ethernet architectures. Parallel execution and concurrent buffers can speed up the performance of gateways. Thus, we focus on analyzing the performance of parallel data structures when integrated with the gateway architecture, focusing on concurrent queues. In our use cases, we compare some of the most popular parallel queue implementations.

More specifically, we examine the design of a gateway architecture that efficiently transports ECG data between the sensor and the server over heterogeneous networks (from Bluetooth to Ethernet - Wi-Fi). As the BGW does not support pulse rates higher than 256 pulses/sec, we develop a digital twin of the BGW to examine the performance at these rates. Our evaluation includes metrics that point out that the concurrent queue has a significantly better overall performance at higher biosensor rates than traditional lock-based producer-consumer synchronization.

In addition, we examine the design of a gateway architecture that efficiently transports a CAN frame across in-vehicle CAN networks using either traditional lock-based producer-consumer or lock-free concurrent queues. In our case study, we consider the implementation of an open, embedded CAN gateway platform and examine real-time performance characteristics when receiving and transmitting CAN frames. Our work examines low-level issues, such as inter-character timeout, and points out that while both queue implementations support high performance with dequeue rates reaching ~380 CAN frames/s, lock-free concurrent queues are faster since they have a smaller waiting time, but they also have higher energy requirements.

Chapter 5 - Future Work

Based on our conclusions and the performance of the concurrent queues, our future work is based on improving further the performance and scalability of gateways. More specifically, we suggest implementing a more complex system, with different networks, more connected devices, heavy data processing, and two-way communication between the gateway and the devices.

Especially for e-Health gateways, we hope to develop efficient implementations for collecting data simultaneously from many heterogeneous sensors, connected over different networks, (Wi-Fi, Bluetooth, Zigbee, etc.), which operate at different rates. Regarding in-vehicle networks, our future work will focus on hybrid gateway architectures that connect to multiple in-vehicle subsystems that operate at different rates, e.g., LIN, CAN, MOST, FlexRay, and CAN FD, and may benefit from multiprotocol OBD II simulators, such as ECUSIM2000.

For both cases, synchronization tasks within our software gateway architecture can rely on multiple-producer multiple-consumer (MPMC) queues or appropriately adapted single-producer single-consumer (SPSC) solutions using buffered multistage networks. In addition, Linux RT policies can be examined to help improve the real-time performance of our gateway architecture.

Apart from these two application domains in e-Health remote sensing and automotive, presented in this study, these gateway prototypes could be used in other domains, such as smart homes and the Industrial Internet of Things (IIoT).

Bibliography

[1] Madakam, S., Ramaswamy, R. & Tripathi, S. (2015). Internet of Things (IoT): A Literature Review. Journal of Computer and Communications, 3, 164-173. doi: 10.4236/jcc.2015.35021.

[2] How gateways can secure IoT architectures. (n.d.). Retrieved November 5, 2022, from https://www.mouser.com/blog/how-gateways-secure-iot-architectures

[3] DiGiuseppe, R. (2020, July 1). Automotive gateway IP enabling scalable automotive platforms. Retrieved November 5, 2022, from Semiconductor Engineering website: https://semiengineering.com/automotive-gateway-ip-enabling-scalable-automotive-platforms/

[4] HPL, S. C. (2002). Introduction to the controller area network (CAN). Application Report SLOA101, 1-17.

[5] Chiuchisan, I., Costin, H., & Geman, O. (10 2014). Adopting the Internet of Things Technologies in Health Care Systems. doi:10.1109/ICEPE.2014.6969965

[6] How an IoT gateway device works: Understanding the architecture. (2017, November 20). Retrieved November 5, 2022, from Embitel website: https://www.embitel.com/blog/embedded-blog/understanding-how-an-iot-gateway-architecture-works,

[7] Du, Y., Sun, G., Zheng, B., & Qu, Y. (2021). Design and Implementation of Intelligent Gateway System for Monitoring Livestock and Poultry Feeding Environment Based on Bluetooth Low Energy. Information, 12(6). doi:10.3390/info12060218

[8] Frigerio, Alessandro & Vermeulen, Bart & Goossens, Kees. (2021). Automotive Architecture Topologies: Analysis for Safety-Critical Autonomous Vehicle Applications. IEEE Access. 9. 62837-62846.

[9] Herlihy, M., Eliot, J., & Moss, E. (06 1993). Transactional Memory: Architectural Support For Lock-free Data Structures. 21, 289–300. doi:10.1109/ISCA.1993.698569

[10] Massalin, H., & Pu, C. (1989b). Threads and Input/Output in the Synthesis Kernal. Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, 191–201. doi:10.1145/74850.74869

[11] Lamport, L. (1977). Concurrent Reading and Writing. Commun. ACM, 20(11), 806–811. doi:10.1145/359863.359878

[12] Dijkstra, E. W. (2002). Cooperating Sequential Processes. The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls (σσ. 65–138). Berlin, Heidelberg: Springer-Verlag.

[13] Valois, J. D. (1995). Lock-Free Linked Lists Using Compare-and-Swap. Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, 214–222. doi:10.1145/224964.224988

[14] Herlihy, M. P., & Wing, J. M. (1987). Axioms for Concurrent Objects. Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 13–26. doi:10.1145/41625.41627

[15] Tsigas, P., & Zhang, Y. (2001). A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures - SPAA '01. doi:10.1145/378580.378611

[16] Michael, M. M., & Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing - PODC '96. doi:10.1145/248052.248106

[17] Lanin, V., & Shasha, D. (1988). Concurrent set manipulation without locking. Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '88. doi:10.1145/308386.308442

[18] Anderson, R. J., & Woll, H. (1991). Wait-free parallel algorithms for the union-find problem. Proceedings of the twenty-third annual ACM symposium on Theory of computing - STOC '91. doi:10.1145/103418.103458

[19] Barnes, G. (1994). Wait free algorithms for heaps.

[20] Fraser, K. A. (2003). Practical lock-freedom. King's College.

[21] Valois, J. (1995). Lock-free data structures. PhD thesis, Rensselaer Polytechnic Institute

[22] Herlihy, M. (1991). Wait-Free Synchronization. ACM Trans. Program. Lang. Syst., 13(1), 124–149. doi:10.1145/114005.102808.

[23] Cameron. (n.d.). readerwriterqueue: A fast single-producer, single-consumer lock-free queue for C++. Retrieved from https://github.com/cameron314/readerwriterqueue

[24] Baker, S. B., Xiang, W., & Atkinson, I. (2017). Internet of things for smart healthcare: Technologies, challenges, and opportunities. IEEE access: practical innovations, open solutions, 5, 26521–26544. doi:10.1109/access.2017.2775180

[25] Bodyguardian Control Unit user manual body gateway. (n.d.). ST Microelectronics, Retrieved November 5, 2022, from https://usermanual.wiki/ST-Microelectronics-S-R-L/MHBGW1.Product-literature

[26] BodyGuardian Mini. - Preventice Solutions. (n.d.). Retrieved November 5, 2022, from https://www.preventicesolutions.com/patients/body-guardian-mini

[27] BodyGuardian MiniPlus. - Preventice Solutions, (n.d.). Retrieved November 5, 2022, from https://www.preventicesolutions.com/patients/body-guardian-mini-plus

[28] CardiBeat. (n.d.). Retrieved November 5, 2022, from https://www.theheartcheck.com/cardibeat/index.html

[29] Bittium Faros 360. (n.d.). Retrieved November 5, 2022, from https://www.bittium.com/medical/bittium-faros

[30] D-Heart. (n.d.). Retrieved November 5, 2022, from https://www.d-heartcare.com

[31] EC-12RM. (n.d.). Retrieved November 5, 2022, from https://www.omnia-health.com/product/ec-12rm-cardiospy-mobile-android-12-channel-resting-ecg-system

[32] Pinto, J. R., Cardoso, J. S., & Lourenço, A. (2018). Evolution, current challenges, and future possibilities in ECG biometrics. IEEE Access, 6, 4746–4776.

[33] Teplitzky, B. A., McRoberts, M., & Ghanbari, H. (2020). Deep learning for comprehensive ECG annotation. Heart Rhythm: The Official Journal of the Heart Rhythm Society, 17(5 Pt B), 881–888. doi:10.1016/j.hrthm.2020.02.015

[34] ElAarag, H., Bauschlicher, D., & Bauschlicher, S. (2013). System Architecture of HatterHealthConnect: An Integration of Body Sensor Networks and Social Networks to Improve Health Awareness. arXiv preprint arXiv:1304.4548.

[35] Personal health gateway: Bluetooth low energy manager. (n.d.). Retrieved November 5, 2022, from http://www.pchalliance.org/personal-health-gateway-bluetooth-low-energy-manager

[36] Shin, S.-H., & Suwon, S. (2004). Apparatus and method for linking Bluetooth to Wireless LAN. US Patent 0071123A1.

[37] Grammatikakis, M. D., Koumarelis, A., & Mouzakitis, A. (2020, November). Software architecture of a user-level GNU/Linux driver for a complex E-Health biosensor. In International Conference on Applications in Electronics Pervading Industry, Environment and Society (pp. 97-103). Springer, Cham.

[38] Grammatikakis, M. D., Koumarelis, A., & Ntallaris, E. (2020, November). Validation of soft real-time in remote ECG analysis. In *International Conference on*

Applications in Electronics Pervading Industry, Environment and Society (pp. 90-96). Springer, Cham.

[39] Grammatikakis, M. D., Petrakis, P., Papagrigoriou, A., Kornaros, G., & Coppola, M. (2015, October). High-level security services based on a hardware NoC Firewall module. In *2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)* (pp. 73-78). IEEE.

[40] Corral-Acero, J., Margara, F., Marciniak, M., Rodero, C., Loncaric, F., Feng, Y., … & Lamata, P. (2020). The 'Digital Twin' to enable the vision of precision cardiology. European Heart Journal, 41(48), 4556–4564. doi:10.1093/eurheartj/ehaa159

[41] Boschert, S., & Rosen, R. (2016). Digital Twin—The Simulation Aspect. In Mechatronic Futures (pp. 59–74). doi:10.1007/978-3-319-32156-1_5

[42] The E/E architecture of the future. (n.d.). Retrieved November 5, 2022, from https://www.bosch-mobility-solutions.com/en/mobility-topics/ee-architecture

[43] Pilato, C., Bohm, S., Brocheton, F., Castrillon, J., Cevasco, R., Cima, V., ... & Hagleitner, C. (2021, February). EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1320-1325). IEEE

[44] Wikipedia contributors. (2022, October 24). CAN bus. Retrieved from Wikipedia, The Free Encyclopedia website: https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=1117982169

[45] CAN_E: Learning module CAN. (n.d.). Retrieved November 5, 2022, from https://elearning.vector.com/mod/page/view.php?id=333

[46] Grammatikakis, M. D., Mouzakitis, N., Kypraios, L., & Papatheodorou, N. (2022). DoS Detection on In-Vehicle Networks: Evaluation on an Experimental Embedded System Platform. In International Conference on Applications in Electronics Pervading Industry, Environment and Society (pp. 262-272). Springer, Cham.

[47] OBD Development Kit. (n.d.). Retrieved November 5, 2022, from Scantool.net website: https://www.scantool.net/obd-development-kit/

[48] Kleine-Budde, M. (2012). SocketCAN: The official CAN API of the Linux kernel. In Proc. CAN Conf (pp. 5–17).

[49] Serial Console. (n.d.). Retrieved November 5, 2022, from https://sourceforge.net/projects/serialconsole/files

[50] Nolte, T., Hansson, H., Norström, C., & Punnekkat, S. (2001). Using bit-stuffing distributions in CAN analysis. In IEEE Real-Time Embedded Systems Workshop at the Real-Time Systems Symposium.

[51] INA231. (n.d.). Retrieved November 5, 2022, from http://www.ti.com/product/ina231

[52] Generic Energy/Thermal Sysfs driver. (n.d.). Retrieved November 5, 2022, from https://www.kernel.org/doc/Documentation/thermal/sysfs-api.txt

[53] Imes, C., Bergstrom, L., & Hoffmann, H. (2016). A portable interface for runtime energy monitoring. In Proc. 24th Int Symp (pp. 968–974).

[54] Mukhanov, L., Nikolopoulos, D. S., & De Supinski, B. R. (2015, October). ALEA: Fine-grain energy profiling with basic block sampling. 2015 International Conference on Parallel Architecture and Compilation (PACT). doi:10.1109/pact.2015.16