Received 18 April 2024, accepted 2 May 2024, date of publication 6 May 2024, date of current version 13 May 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3397467*

RESEARCH ARTICLE

# A C-ITS Architecture for MEC and Cloud Native Back-End Services

**JAVIER ARIN**[1,2], **GORKA VELEZ**[1], **AND PAUL BUSTAMANTE**[2,3]
[1]Fundación Vicomtech, Basque Research and Technology Alliance (BRTA), 20009 Donostia-San Sebastian, Spain
[2]Tecnun, Universidad de Navarra, 20018 Donostia-San Sebastian, Spain
[3]CEIT-Basque Research and Technology Alliance (BRTA), 20018 Donostia-San Sebastian, Spain

Corresponding author: Gorka Velez (gvelez@vicomtech.org)

This work was supported in part by the European Union's Horizon 2020 Research and Innovation Program through the Project Monetizing car & mobility data for new Entrants, Technologies and Actors (5GMETA) under Grant 957360.

**ABSTRACT** Advances in connectivity and computing infrastructure facilitate the introduction of innovative Cooperative Intelligent Transport Systems (C-ITS) services. However, meeting the requirements of these highly demanding services calls for novel computing architectures that handle extensive device connections, minimize latency, and support multiple resource-intensive services concurrently. To overcome these challenges, this work presents an architecture that comprises three layers: 1) on-board unit (OBU) mainly as a data producer; 2) intermediate edge layer where low-latency backend services can be deployed; and 3) cloud layer for non-real-time backend services. The OBU software stack implements the ETSI C-ITS standard and supports multicast over the cellular network. The edge layer includes an in-memory database, and the cloud layer a persistent database. Each layer has its own Application Programming Interface (API) for data consumption. We conducted several experiments to demonstrate the feasibility of our proposed system that ensures scalability and interconnection between vehicles, edge and cloud servers. We also assess the delay caused by each of the elements of the architecture, and we discuss the potential solutions for the identified issues.

**INDEX TERMS** V2X, C-V2X, 5G, MEC, cloud, C-ITS.

## I. INTRODUCTION

In the rapidly evolving transportation landscape, the advent of Cooperative Intelligent Transport Systems (C-ITS) has led to a paradigm shift that holds unprecedented promise for enhancing road network safety, efficiency, and sustainability. By seamlessly integrating advanced communication and information technologies into vehicles and infrastructure, C-ITS facilitates real-time data exchange, cooperative decision-making, and dynamic coordination among all elements of the transportation ecosystem. The data exchange between ITS stations such as cars, roadside units (RSU) or even smart devices is handled by mature ad-hoc short-range communication technologies like ETSI ITS G5 or IEEE WAVE, both based on 802.11p and complemented by wide area communication systems like 4G and 5G.

With advancements in connectivity and data handling technology, along with the development of robust infrastructures, a wave of intriguing and innovative Intelligent Transport Systems (ITS) services and applications has emerged. Nevertheless, these novel services bring forth new challenges, particularly in the realms of low latency and high throughput. Addressing these demands necessitates the creation of novel computing architectures capable of accommodating extensive device connectivity, ensuring minimal latency, and supporting multiple resource-intensive services concurrently.

This paper aims to demonstrate that it is possible to build a three-layer architecture composed of On-Board Units (OBUs), Multi-Access Edge Computing (MEC) and cloud that is capable of gathering and providing timely data to MEC and cloud-native back-end services.

The existing solutions are heterogeneous. Some are targeting only ad-hoc short-range communication technologies [1]

The associate editor coordinating the review of this manuscript and approving it for publication was Shaohua Wan.

and therefore, do not leverage MEC or Cloud technologies that allow offloading computational tasks and deploying advanced ITS services by third parties. Others actually implement a MEC-Cloud based architecture. However, they lack either storage capability [2] or explicit security measures [3], [4], [5], [6]. Moreover, to the best of the authors' knowledge, no work in the literature proposes an implementation of the C-ITS stack with multicast over the cellular network.

The architecture proposed in the present work has several key features that collectively enhance the Cooperative Intelligent Transport Systems (C-ITS) framework, enabling seamless communication and data exchange among OBUs, MEC, and cloud. The key features can be summarised as follows:

- *Enhanced OBU Communication Stack:* The proposed OBU communication stack implements the ETSI C-ITS standard and supports versatile ETSI Cooperative Awareness Messages (CAMs) and Cooperative Position Messages (CPMs) exchange options, including communication between OBUs through Multicast over Ethernet or 5G. The stack makes handling CAM and CPM messages easier by converting binary messages to and from the JSON text format and using MQTT as an abstraction layer. Additionally, the stack incorporates support for Single Hop Broadcast and implements security mechanisms.
- *MEC Access Network Application:* This software module for the MEC that acts as an access network efficiently receives CAMs and CPMs in *PER* format from sources such as Multicast over 5G and seamlessly relays them to the ITS back-end service via MQTT. By bridging communication between vehicular components and the MEC, this application enhances data availability and accessibility within the ecosystem.
- *Dual ITS Back-End Service:* A novel ITS back-end service is proposed, offering extensive storage and data delivery capabilities for CAMs and CPMs. The data storage system is designed with a dual structure, comprising cache databases situated within individual MEC servers and a cloud-based database optimised for handling substantial data volumes. The cache databases ensure rapid data insertion and low-latency retrieval, catering to real-time applications. On the other hand, while managing significant data loads, the cloud database prioritises efficiency over constant insertion times and low latency, making it suitable for non-real-time applications.

To the best of the authors' knowledge, an architecture including the mentioned features is not available in the literature. Collectively, these features empower a C-ITS architecture where OBUs, MEC, and cloud infrastructures synergize to optimise data communication, storage, and retrieval. The proposed architecture makes vehicle and roadside infrastructure data accessible to MEC and cloud-based back-end services, using standard vehicular messaging formats. The study conducted in the present paper clarifies how such an architecture can be implemented using open-source tools and identifies the impact of each architecture element on the total pipeline latency.

The rest of the paper is organised as follows: Section II reviews the related work, Section III describes the architecture and implementation of the proposed system, Section IV describes how the evaluation was conducted, Section V shows the results obtained in the experimentation, and Section VI concludes the paper.

## II. RELATED WORK

This section presents an overview of the state of the art about vehicular communications, C-ITS messaging, MEC and cloud technologies. Then, it reviews the ITS messaging architectures found in the literature, highlighting the differences with the present work.

### A. VEHICULAR COMMUNICATION STANDARDS

The landscape of vehicular communication standards is evolving, with a shift from Dedicated Short Range Communication (DSRC) to Cellular Vehicle-to-Everything (C-V2X) technology. While DSRC initially gained traction, C-V2X is gaining momentum through hardware like Qualcomm's 9150 chipset, exemplified by Commsignia's ITS-RS4D and Cohda Wireless' MK6, which supports both DSRC, New Radio (NR), and Long Term Evolution (LTE) C-V2X [7]. ITS-G5, the European standard for DSRC in ITS, is optimized for cost and simplicity and inherently supports distributed operation. This is what is commonly referred to as V2X.

The performance of DSRC or ITS-G5 and LTE C-V2X is compared in several studies. Maglogiannis et al. [8] conducted an evaluation revealing that, for short-range technologies, C-V2X PC5 generally exhibits a greater range than ITS-G5. However ITS-G5 offers lower latency than C-V2X PC5 in low-density scenarios. Zhao et al. [9] performed simulations showing that the difference in communication reliability between DSRC and LTE is small at short communication distances. LTE ensures more reliable communication when vehicle density is low, whereas DSRC outperforms LTE in high-density scenarios. Additionally, Petrov et al. [10] noted that LTE C-V2X is suitable for most low-frequency Vehicle-to-Infrastructure (V2I) services within a limited communication range (<600 m) and for lower traffic intensities.

Using NR V2X instead of LTE-based C-V2X offers advantages like higher throughput, higher reliability, and lower latency V2X services for new sidelink functions [11]. The results in [12] show that denser infrastructure deployments are very beneficial for both technologies and their modes. In this study, the upgrade from 2 base stations to 4 base stations offered significant improvements for all the considered V2I services as well as C-V2X technologies and their modes, particularly in the case of 5G-based C-V2X.

### B. C-ITS MESSAGING STANDARDS

In the realm of ad-hoc networks, conventional IP routing faces limitations due to the dynamic nature of these networks, necessitating alternative routing protocols. While an array of routing protocols exists for IoT ad-hoc networks, the European Telecommunications Standards Institute (ETSI) undertook the development of a custom routing protocol alongside a comprehensive standard. The standard encompasses multiple layers, mirroring the structure of the OSI model, with each layer serving a distinct purpose: "Access" representing OSI layers 1 and 2, "Networking & Transport" representing OSI layers 3 and 4, and "Facilities" representing OSI layers 5, 6 and 7 [13].

The first release of the standard addresses the technology-agnostic aspects of the access layer and outlines specifications for the ETSI ITS G5 interface [14]. A significant highlight of the C-ITS standard is its emphasis on compatibility. Existing ad-hoc routing protocols often require convoluted adaptations to function with IP. In contrast, the C-ITS standard natively supports IPv6 and TCP/UDP. Furthermore, the standard introduces the GeoNetworking protocol, which addresses routing challenges inherent to ad-hoc networks while accommodating IPv6 compatibility.

GeoNetworking introduces innovative forwarding schemes such as Single Hop Broadcast, Topologically Scoped Broadcast, GeoUnicast, GeoBroadcast, and GeoAnycast. These schemes leverage geographical addressing and forwarding to enable efficient packet delivery without the need for elaborate routing tables.

To ensure low-latency, reliable, and efficient communications, the C-ITS standard employs mechanisms such as Status Information Signaling, Priority and Buffering, and Decentralized Congestion Control. Moreover, public key infrastructure is harnessed to enhance security, with cryptographic signatures serving as a safeguard against potential security breaches.

Some works of the literature have addressed the limitations of the infrastructure-based trust model defined in the C-ITS standard and have proposed other approaches. Liu et al. [15] proposed a trust cascading-based emergency message dissemination model in VANETs, incorporating entity-oriented trust values into data-oriented trust evaluation. Entity-oriented trust models typically establish trust relationships between vehicles through factors such as similarity, experience, and role. Conversely, data-oriented trust models prioritize assessing the trustworthiness of received messages rather than the vehicle that broadcasts the message. A Privacy-Preserving Reputation Updating (PPRU) scheme for cloud-assisted vehicular networks is proposed in [16]. A more complex network architecture is introduced in [17], integrating space segment (including satellites, the corresponding ground stations, etc.), air segment (including Unmanned Aerial Vehicles (UAVs), airships, etc.), and ground segment (including RSUs, BSs, vehicles, etc.). The same work proposes a privacy-preserving trust management scheme for emergency message dissemination in space-air-ground integrated vehicular networks.

The network architecture prescribed by the C-ITS standard is described in *ETSI EN 302 636-3* [18]. A fundamental distinction is drawn between internal and external networks within the standard. The internal network of an ITS station interconnects its constituent components, while external networks facilitate interconnection between ITS stations and other network entities.

Three primary external networks are identified within the C-ITS standard:

- *ITS ad-hoc network:* Serving as the primary framework, this network encompasses vehicle, roadside, and personal ITS stations. It leverages short-range communication technologies such as ITS-G5 (802.11p) or cellular (LTE, NR) to enable ad-hoc communications.
- *Access network:* A dedicated network facilitating access to specific ITS services and applications. It serves as a bridge between the ad-hoc network and the core network, enabling communication with road traffic management centres and back-end services. Access layers depend heavily on the technology used for communications. Release 1 of the C-ITS standard gives a technology-agnostic take on the access layer as well as a specific implementation for the ITS-G5 communication technology.
- *Core network:* Offering legacy services akin to regular internet, this network provides connectivity for services such as www and email. At the core of the network architecture resides the ITS station, functioning as both a communication source and sink. The ITS station plays a pivotal role in consuming and forwarding data, while the internal network hosts the ITS stack.

### C. MULTI-ACCESS EDGE COMPUTING (MEC)

The advent of 5G, Internet of Things (IoT), and Vehicle Ad-hoc Networks (VANETs) is driving significant changes in internet and service distribution. Emerging applications demand enhanced Quality of Service (QoS), reduced energy consumption, lower latency, and increased capacity [19]. This impetus has led to the emergence of novel computing paradigms like fog computing and multi-access edge computing (MEC) [20], [21].

MEC stands out in vehicular applications for its support of high dynamicity and mobile clients [19], enabling diverse applications within the context of C-V2X such as cooperative autonomous driving [22], collision avoidance [23], platooning [24], [25], back situation awareness [3], video streaming between vehicles [4], [5] or Edge Dynamic Maps [6].

In the context of MEC, resource allocation for client services is crucial, given real-time application requirements. Innovative offloading algorithms [26], [27] and mobility-aware strategies [28], [29], [30] are proposed to optimise resource distribution and accommodate mobility challenges.

### D. CLOUD INFRASTRUCTURE

Cloud infrastructure has become increasingly important in recent years as organizations have adopted cloud computing to meet their computing needs. Cloud infrastructure offers several benefits, including on-demand access to resources, scalability, and cost savings compared to maintaining in-house infrastructure [31].

#### 1) ACCESSING CLOUD SERVICES

Many different paradigms have been developed to access cloud services. The most used ones are Representational State Transfer API (RESTful or REST API), Remote Procedure Call (RPC) and publish-subscribe (Pub/Sub) message brokers [32].

REST APIs are based on the HTTP protocol and use standard methods to retrieve or modify data, such as GET, POST, PUT, and DELETE. They are typically request-response based, with the client sending a request and the server sending a response, they were first introduced in the year 2000 [33] and have shaped the internet since [34].

RPC is a technology that enables a computer program to initiate the execution of functions or procedures located in a separate program or device as if they were accessible and executable within the local program. It is often used in distributed systems to enable communication between components located on different machines or networks [35].

Pub/Sub message brokers are used for building real-time data pipelines and streaming applications, as well as enabling communication between microservices and other distributed systems [36].

#### 2) CLOUD DATA STORAGE

Data storage is one of the key components of cloud computing [37]. The demand for high computing capabilities often comes with the need for high data storage volumes. When classifying data storage by data availability, we can differentiate between

- *In-Memory databases:* in-memory or cache databases are designed to be fast, with low latencies and high read and write speeds. They use simple data models like key-value but sacrifice data consistency for speed. Examples of in-memory databases include Memcached and Redis [38].
- *Persistent databases:* These databases support very complex data models and queries and preserve data consistency at the expense of speed. PostgresDb or MongoDb are examples of persistent or non-cache databases.

Another key classification between databases is if they are relational or non-relational [39].

- *Relational or SQL:* They are based on the relational model, which organises data into tables of rows and columns. They excel at storing structured data.
- *Non-relational or No-SQL:* They are not based on the relational model and can handle unstructured data. They are more horizontally scalable than SQL databases but do not offer transactions.

### E. ITS MESSAGING ARCHITECTURES

Vehicular communication standards, C-ITS messaging standards, Edge, Cloud or in general, IoT technologies are considered when building a messaging or data-sharing architecture for ITS. In the literature, several approaches can be found that combine these elements in different ways.

An approach to bridge C-ITS messages over MQTT to Apache Kafka is presented in [40]. This facilitates a topic mapping between MQTT and Kafka, from vehicles and roadside units to a central application, and in the opposite direction. This study validates the feasibility of Kafka and MQTT as possible building blocks in a C-ITS architecture. Several messaging technologies (Apache Kafka, Eclipse Zenoh, Apache ActiveMQ, Apache Pulsar and KubeMQ) applicable in vehicular data platforms are compared in [41]. However, the study does not consider MQTT, which is the messaging technology used in the present paper.

An intra- and inter-vehicle sensory data collection system is presented in [42]. The ETSI C-ITS stack is only implemented for ITS-G5. The data is also shared with a smartphone using WiFi and MQTT. The collected data by the ITS-G5 station or the smartphone are sent to a cloud platform using MQTT. ITS-G5 is also used in [43], in this case for the communication between vehicles and RSUs. The RSUs are deployed with a cellular link or fibre optics connection, enabling connection to a cloud-based MQTT broker. This work is continued in [44], proposing a secure communication design based on a multi-layered blockchain architecture.

A 5G and MEC-enabled Server Local Dynamic Map (S-LDM), aimed at collecting information about vehicles, based on standard-compliant messages, is presented in [45]. In the proposed architecture, AMQP is used for collecting data, and REST API for making data accessible to other MEC services. In the implemented infrastructure, a single instance of the S-LDM is able to manage up to at least 550 vehicles. 5G and MEC technologies are also leveraged in [2], where a novel Network Application concept applied to the automotive domain is introduced. In the context of this work, a Network Application is essentially a virtual application that can be deployed in a 5G infrastructure and can use 5G services. A Network Application platform is proposed that includes the integration of mobile far-edge resources. However, the work is purely theoretical and no experimentation results are presented for comparison. Other works [3], [4], [5], [6] also use 5G, MEC and cloud technologies, and include storage capability, but they all lack explicit security measurements and the implementation of the C-ITS stack with multicast, as it is done in the present paper.

To sum up, the present work differentiates from the scientific literature (see Table 1) by 1) proposing an ETSI

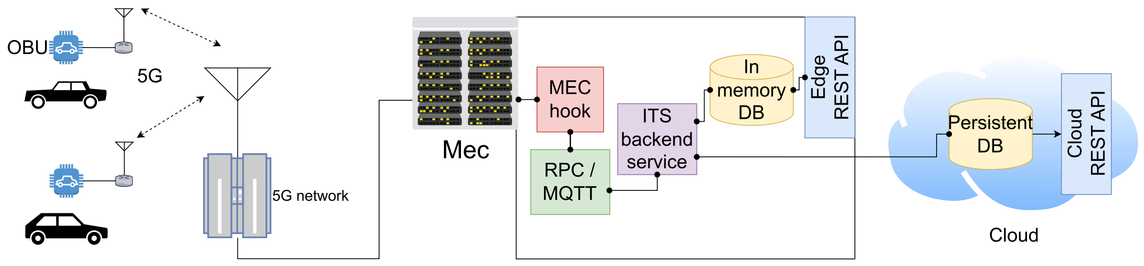


FIGURE 1. High-level system architecture.

C-ITS communication stack through Multicast over 5G, 2) proposing a 3-layer architecture based on OBU, MEC and cloud that implements the security measures defined in the ETSI C-ITS standard, 3) proposing a data storage system with a dual structure, formed by a cache database in the MEC and persistent database in the cloud, and 4) studying in detail the impact of each architecture element on the total pipeline latency, paving the way for future enhanced architectures.

TABLE 1. Architecture elements and features implemented on each work.

| Work | C-ITS stack with multicast over 5G | MEC | Cloud | Storage | Security |
|---|---|---|---|---|---|
| [3]–[6] | | ✓ | ✓ | ✓ | |
| [40] | | | ✓ | ✓ | |
| [41], [42] | | | ✓ | | |
| [43] | | | ✓ | | ✓ |
| [44] | | | | ✓ | ✓ |
| [45] | | ✓ | | ✓ | |
| [2] | | ✓ | ✓ | | ✓ |
| Present work | ✓ | ✓ | ✓ | ✓ | ✓ |

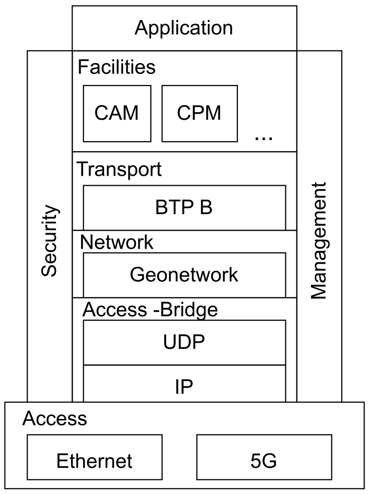
## III. ARCHITECTURE IMPLEMENTATION

Figure 1 shows the proposed system architecture. Vehicles send standard ETSI C-ITS messages that are captured by the MEC. The MEC inserts the received data into two different databases: an in-memory database, and a persistent database. The low-latency applications running in the MEC access the in-memory database, while the cloud applications access the persistent database. The proposed architecture is therefore divided into different elements: On-board Unit (OBU) stack, MEC hook, ITS back-end service, and Consumer APIs. In the following subsections, each element is described in detail.

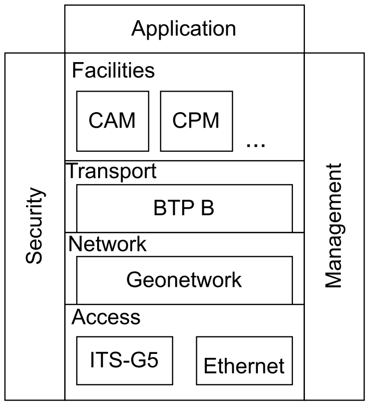
### A. OBU STACK

The OBU software contains the implementation of the whole C-ITS communications stack, from the access layer to the facilities layer, offering a usable API for applications. Due to the restricted resources found commonly in OBU hardware, the programming language chosen is C++11.

The software listens to multicast, raw ethernet or Cohda Wireless MK5 OBU's packets from the selected interface.



(a) Multicast access layer implementation



(b) Raw Ethernet access layer implementation

FIGURE 2. Complete stacks with different access layer implementations.

The original ETSI C-ITS standard does not specify a multicast access layer. Due to the lack of support of the standard by conventional routers, a multicast bridge is needed for multi-network simulations, whereas the raw ethernet access layer works in single-network simulations.

The proposed software architecture resembles the standard. It is divided into different sequential layers that act as a pipeline. Each step of the pipeline has a set of specific tasks. An MQTT message broker is used as an API between the facilities and the application layers. The facilities layer is extendable to suit different use cases such as CAM message generation and codification, CPM message reception or Local Dynamic Map generation.

The OBU software architecture is greatly shaped by Vanetza [46]. Vanetza implements some parts of the standard as well as provides a framework to extend the stack capabilities. It currently implements a GeoNetworking router with SHB and GeoBroadcasting forwarding support. It can also work with Basic Transport Protocol (BTP) which is the connectionless transport protocol defined by the standard as well as implementing a Public Key Infrastructure (PKI) for the secure signing of the packets. Vanetza acts as the base-building block for the entire OBU software.

Vanetza is designed to work as a single-thread event loop. The proposed system is asynchronously listening for incoming packets as well as listening for software timers. These timers are used to start all the services that need to be called periodically, such as the CAM message generation and the CPM message generation.

### 1) THE EVENT LOOP

The event loop works by asynchronously polling different event sources. Once one of these event sources triggers an event, the event is posted in the loop and the loop takes care of it. The loop forwards the event to the event dispatcher which decides where which system should handle the event. In the case of receiving a packet, the packet reception pipeline takes care of processing the packet and forwarding it to the proper facility. Figure 3 shows the flow architecture of the event loop implemented.

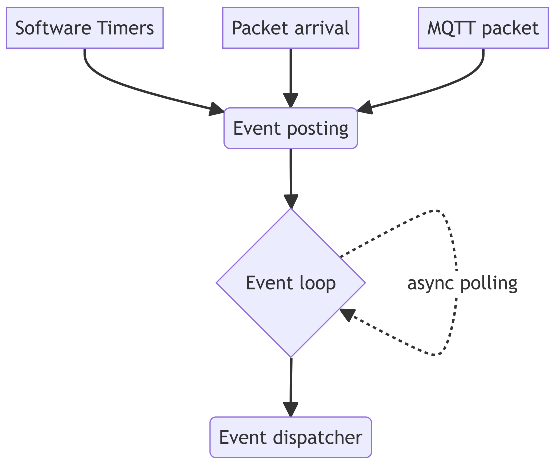


**FIGURE 3.** **Event loop architecture.**

Depending on the timer source, the event dispatcher runs the CAM generation and transmission pipeline or the CPM transmission pipeline. Finally, if an MQTT event is issued, the event dispatcher runs the pipeline dedicated to receiving the CPM and storing the CPM until the next CPM transmission is needed.

A scheduler was built to share resources for the periodical tasks, such as updating the MQTT status or sending CAMs/CPMs. While periodic tasks work in conjunction with the scheduler, the packet reception is out of the scheduler and always starts immediately.

### 2) PACKET RECEPTION PIPELINE

As the communications stack is bidirectional, we can separate it into incoming packets and outgoing packets. A sequence diagram of the reception pipeline is shown in Figure 4. When a packet is received in the access layer, it is processed considering the type of access layer used.

- For the *Multicast* access layers, the packet is received and the IP/UDP Multicast headers are stripped by the OS. The remaining packet is stored in a buffer and passed to the network layer.
- For the *raw Ethernet* access layer, it depends on whether it receives ITS-G5 packets using the Cohda Wireless SDK or via Ethernet. When receiving ITS-G5 packets, the ITS-G5 specific headers are removed by the Cohda wireless SDK, and the packet is passed to the network layer. We check and remove the Ethernet header and pass the packet up to the network layer when using Ethernet.

Once on the network layer, GeoNetworking takes care of the packet. GeoNetworking uses its own headers to know whether the packet is for the current ITS station or for another one. GeoNetworking packet header fields and structure can be found in [47].

Vanetza currently only supports GeoBroadcast packets and single hop broadcast (SHB). The forwarding of the packets is handled by Vanetza in the following way:

1) First, the network layer receives an encoded packet with the packet structure shown in Figure 5. If the packet does not follow this structure in any of the following steps, it is discarded.
2) The basic header is processed. It first checks the version and then the NH parameter to see if the next header is a secured header or not.
3) If the packet is not secured, it jumps to step 4. Otherwise, it is passed to the security entity, where its signature is validated. If the signature is invalid, the packet is dropped, whereas if the signature is valid, the common header is passed to the next step.
4) The common header is processed. First, the maximum hop limit of the header is checked. Then all packets that are pending forwarding are forwarded.
5) The Header Type (HT) of the packet is checked.

Different forwarding schemas are activated depending on the packet's HT field.

- If the HT parameter is 5, a single hop broadcast packet has been received, and the SHB extended header is read and directly sent to the transport layer.
- If the HT parameter is 4 the packet is processed by the GeoBroadcasting mechanism

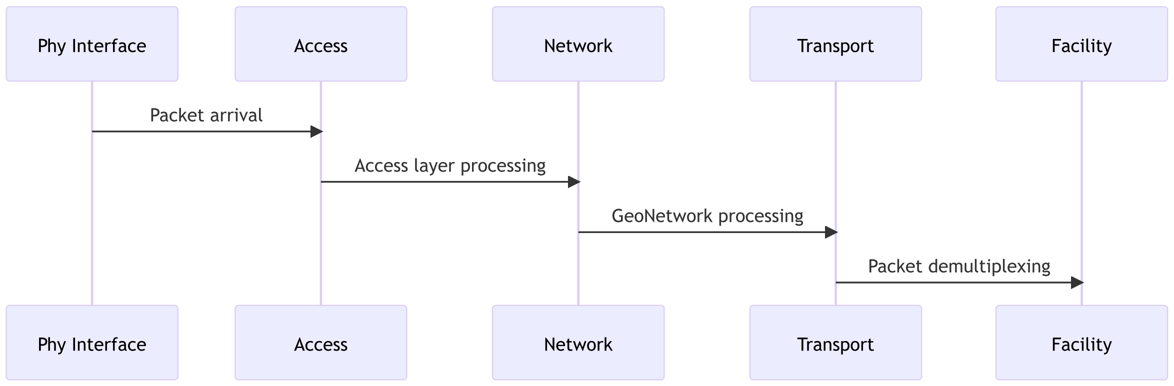


**FIGURE 4.** Packet reception pipeline.

| | | | | |
|---|---|---|---|---|
| Access layer header | GN Basic header | GN Common header | GN Extended header (opt) | Payload (opt) |

(a) Packet with no security

| | | | | | |
|---|---|---|---|---|---|
| Access layer header | GN Basic header | GN secured header | GN Common header | GN Extended header (opt) | Payload (opt) |

(b) Packet with security

**FIGURE 5.** GeoNetworking packet structure.

1) The ITS station determines if it is inside the target geographical area by consulting the GeoBroadcast extended header.
2) If the ITS station is inside the area, the packet payload is passed to the transport layer. If the station is not inside the area, the packet is forwarded to the next station that is the closest to the target area.

Once on the transport layer, the protocol ruling the next steps is the Basic Transport Protocol (BTP). BTP is quite simple in that its only job is to provide for facility demultiplexing. It is a 4-byte header protocol that requires minimal processing and just stores the target and source ports. It does not provide any reliability mechanism and assumes other layers to provide reliability if needed.

The packets that are moved to this layer are demultiplexed to the facility listening for the port that matches the BTP port. So, packets belonging to CPM are delivered to the CPM facility application, whilst CAM packets are handed to the CAM facility application.

In packet reception, both facility applications work very similarly. They decode the payload of the packet without taking into account any header and store it in an intermediate C++ representation object. then, RapidJSON is used to parse the C++ object to a valid JSON which is dispatched to the MQTT broker, and applications subscribed to it receive the JSON.

### 3) PACKET GENERATION PIPELINE

The packet generation pipeline starts on the facility layer that requests to send a message. Figure 6 shows the sequence diagram of this pipeline. If the requested message is a CAM, the facility layer generates a new CAM message using the available parameters in a C++ object and encodes it using Packed Encoding Rules (PER) encoding. If the requested message is a CPM, the facility checks whether a CPM has already been received or not. If a CPM has already been received via MQTT, it is encoded. A default CPM packet is generated and encoded if no CPM has been received.

Once the packet is encoded, it is passed down to the transport layer with an indication object containing the desired destination. In the case of CPM and CAM messages, the destination is always a single-hop broadcast.

The transport layer takes the encoded packet, adds its source/destination port header to the head of the packet and passes it down to the network layer with the indication.

In the network layer, the following steps are followed:

1) A SHB extended header is added and filled.
2) A Common header is added and filled.
3) A Basic Header is added and filled.
4) The packet is signed, and a security header is added if security is activated.
5) If there is any neighbour, it sends the packet to the access layer. If there is no neighbour, it stores the packet in the forwarding packet buffer instead.

Finally, in the access layer, the specific operations of the selected layer are executed, and the packet is sent.

### 4) MQTT PIPELINE

The MQTT can be subdivided into the diffusion and reception pipes. These pipelines are shown in Figure 7. The diffusion one is embedded in the packet reception pipeline and emits packets received by the facility layers to the applications that are subscribed to it.

The MQTT CPM reception pipeline starts in user applications. User applications that want to send a CPM generate a JSON version of the CPM and send it through the MQTT API to the CPM facility. The CPM facility stores the latest received CPM. The stored CPM is not sent until the CPM

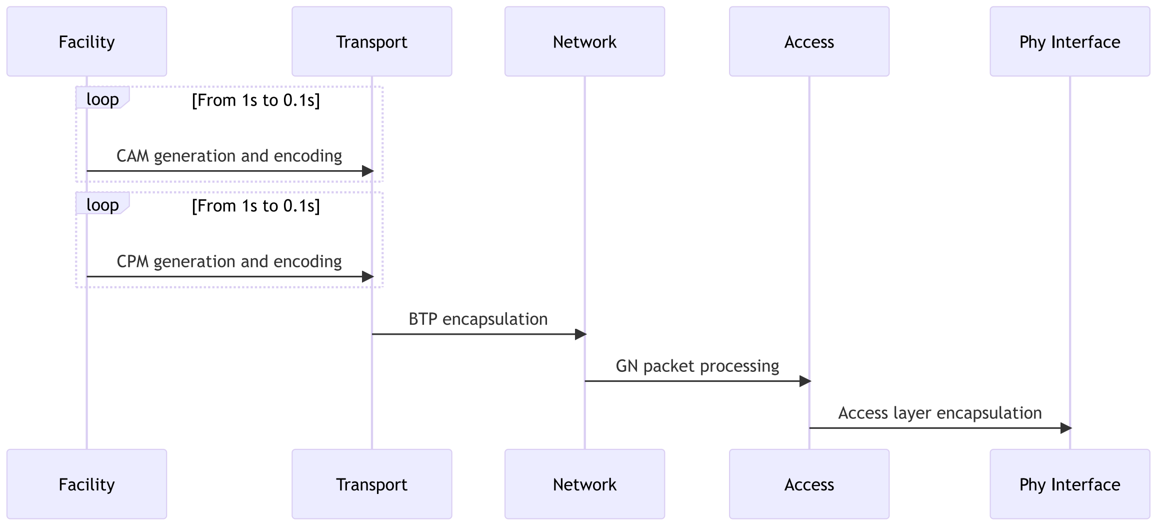


**FIGURE 6.** **Packet diffusion pipeline.**

generation timer triggers by using the packet generation pipeline.

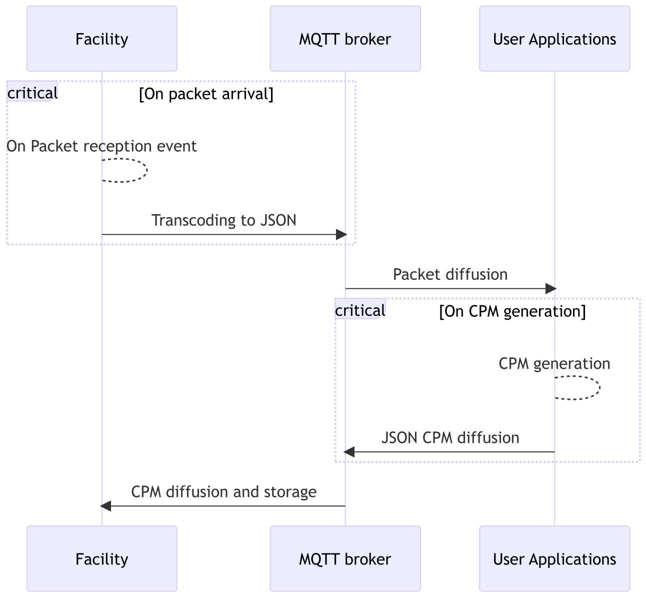


**FIGURE 7.** **MQTT pipeline.**

## *B. MEC HOOK*

The MEC hook is the software responsible for bridging the VANET and the core network. It corresponds to the access network and passes all the received data to the traffic back-end services. It must handle high I/O loads and decode many packets, but it would run on much more powerful hardware than the OBUs.

The MEC hook uses the same architecture as the OBU software except for packet decoding and MQTT transmission. The OBU software is completely single-threaded, whereas the MEC hook uses a hybrid approach. Due to the internal structure of Vanetza, everything must be single threaded up to the facility layer, so packets are captured and forwarded to the facility layer sequentially in the main thread. In the facilities layer, we use worker threads.

Many different approaches could be taken, but thread pools give the best ratio between efficiency and scaling [48]. A thread pool would not let threads starve for resources and would be able to process in parallel. Figure 8 shows the functional structure of the thread pool.

We take advantage of the fact that each packet in the MEC hook is independent of the others, so they can be processed in parallel without having to worry about synchronisation. The only synchronisation point is the message queue for the MQTT. This queue stores the messages until the MQTT client dispatches them.

## *C. ITS BACK-END SERVICE*

The ITS back-end service stores all the CAM and CPM data received by different MEC hooks. A central cloud computing system stores the data for non-real-time applications, and a distributed system temporarily caches the data for real-time applications and then sends it to the central cloud.

The real-time option is only available to services running inside the MEC. Eventually, all of the data will end up in the cloud storage, so applications that need to process big data should just use the cloud side or would need to build a distributed system that would also work in a MEC native architecture.

This back-end service manages the database instances and moves the data around. Services are able to access the data via HTTP APIs. In the proposed implementation, the service is written in TypeScript and uses the NodeJS runtime. NodeJS is based on an event loop and can handle heavy amounts of I/O.

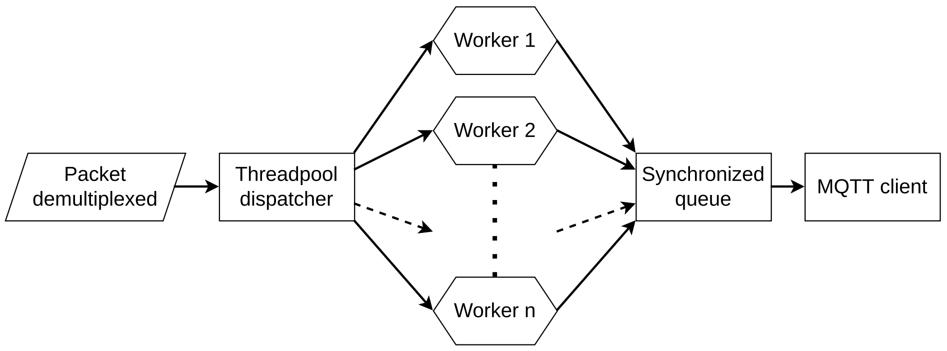


**FIGURE 8.** MEC hook facility application architecture using the worker-pool executor.

### 1) IN-MEMORY DATABASE

The selected in-memory database needs to have strict timing requirements for data insertion and retrieval, including low latency and predictable access time.

For a hard disk drive access on a 1 GHz processor and a million cycles, the time it takes to load a memory block from a hard disk is around 1 ms [49], [50], and this access is not guaranteed to contain the desired data. This makes databases that store data outside of the RAM too slow for our requirements, so we must select a cache database. Hard disks also tend to be very inconsistent with access times as they depend on the current position of the disk and the cursor, as well as the fragmentation of the data.

As we need to be able to insert in constant predictable times, insertion should be $\mathcal{O}(1)$ which means we need a very big hashmap-like implementation. Redis and Memcached both fulfil those requirements, and while Memcached is faster [51], it lacks the ability to query by indexes, so it is not suitable for our case. Redis has support for sharding and replication, which are other requisites as these increase the system availability and resilience to failure. Considering all these points, Redis is chosen as the in-memory database.

### 2) PERSISTENT DATABASE

Storing large amounts of data in Redis would result in a big RAM consumption. For example, a CAM message formatted in JSON occupies around 4 KB. In a scenario of 10, 000 cars each sending a CAM every second for 30 days, that would produce 103, 68 Terabytes of uncompressed data. Data availability is also very important. Cache databases lose all their information if the system goes down, while non-cache databases do not.

Due to the flexibility of the data we are dealing with, SQL databases are not a good choice. SQL databases depend on fixed data structures to be efficient. However, C-ITS messages contain optional fields only sent under certain circumstances. The format of the messages will also probably change with time as new versions of the standards are released, so the rigidity of SQL databases will not help.

The two most common No-SQL databases are MongoDB and Cassandra. Both are scalable and replicable using different methods. Cassandra is multi-main, while MongoDB is main-secondary based. Cassandra is column-based, while MongoDB is document-based and supports JSON natively. Both have their own query language and support transactions. Performing a Yahoo Cloud Serving Benchmark (YCSB) test on both returns mixed results [52], so finally MongoDB was chosen because of its native support for JSON format.

### 3) SOFTWARE ARCHITECTURE

The software architecture is designed to be horizontally scalable and flexible. Figure 9 shows the overall structure of the system with the Consumer APIs attached to the sides and Figure 10 shows the flow of data inside the software.

The system is divided into different interfaces. These interfaces are agnostic to the technologies used and the communication methods. They are designed so changes in the message system or the databases do not change the rest of the service while also allowing parts of the service to sit in different servers in a micro-service-like architecture. The interfaces are:

- *Message manager*: Implementations of this interface are responsible for tapping into the message broker and receiving the data. It passes this data into the DB manager for further storage.
- *In-Memory controller*: The In-Memory controller implementation manages the creation of the indexes in the in-memory database as well as inserting the data in the proper internal database representation of it. It also retrieves data in a streaming or non-streaming manner to rebase it to the persistent database. The implementation in this work is adapted to Redis.
- *Persistent controller*: This interface's implementation creates database collections and indexes for the CAM and CPM. They must accept in-memory data streams to fill the database with the rebase data. The implementation in this work is adapted to MongoDB.
- *DB manager*: The DB manager is responsible for creating the controller instances and creating the rebase pipeline and scheduling the rebase interval. There should be 1 DB Manager per In-Memory database instance.

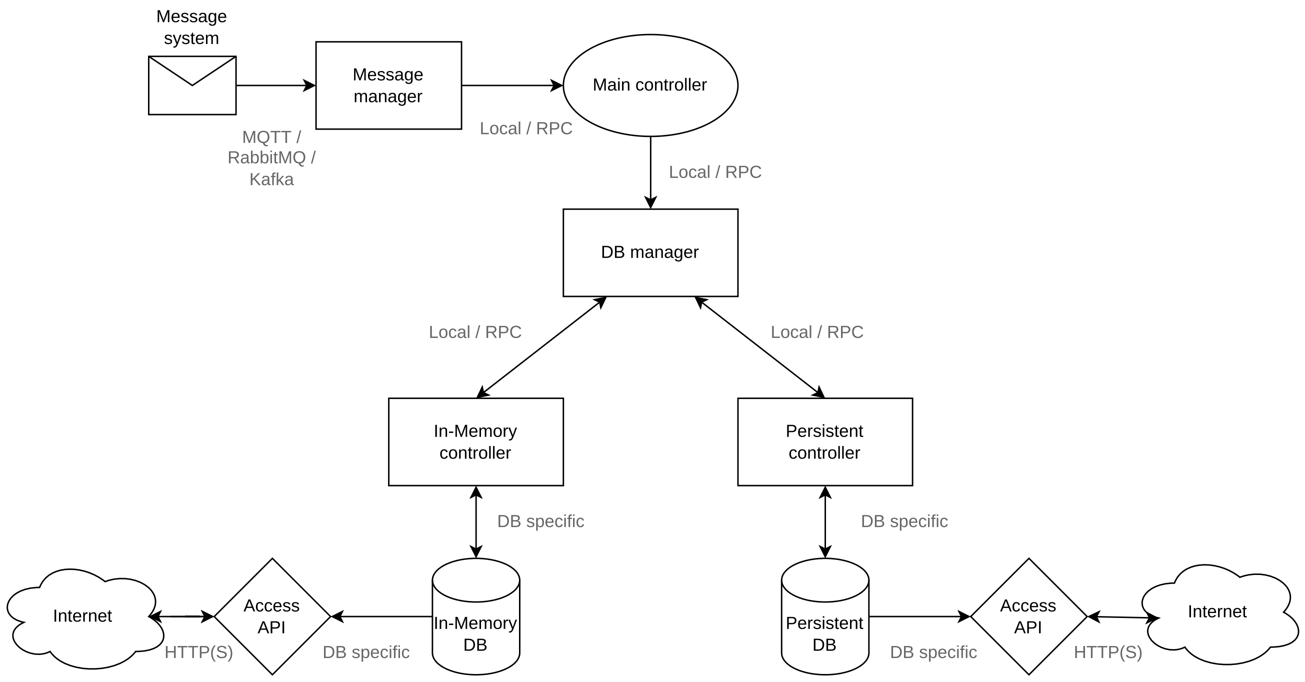


**FIGURE 9.** **Back-end service software architecture with data access APIs.**

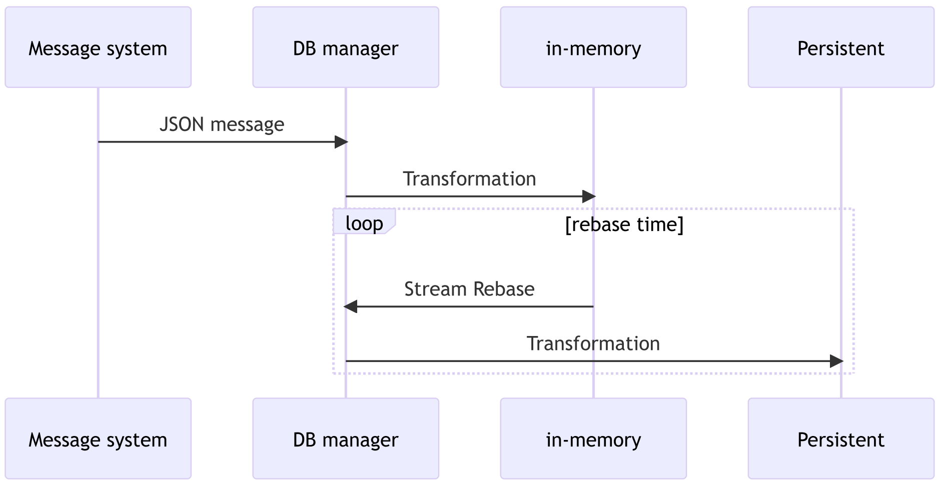


**FIGURE 10.** **Back-end service data flow diagram.**

- *Main Controller*: The main controller creates the DB manager and message manager instances and connects them.

Besides moving data around, the in-memory and persistent controllers must make some transformations to the data to place it in the database. For example, we use the RedisSearch module to create indexes and query the data. RedisSearch manages the indexes internally by dividing inputs into different hash tables for the different indexes achieving high speeds, but this requires us to extract some data from the message previously. So, we finally store the whole message string with a duplicate of the timestamp, the position, the MAC and the type. While this redundancy can seem odd at first, this would also let us compress the message string in the future if it is needed.

Just as with Redis, MongoDB also has its internal representation of the messages, so when rebasing the data, it has to be transformed again. The difference between the Redis and MongoDB representations is the position coordinates. MongoDB supports GeoJSON, so the position must be transformed into a GeoJSON position.

It may be the case that when the rebasing is triggered, there are thousands or hundreds of thousand messages to migrate from the in-memory to the persistent. Given a

JSON representation of 4KB, 100, 000 documents would weigh 400 MB. NodeJS has a heap limit of 700 MB, effectively consuming almost all the available memory, even in the case of leaking memory. We also would like to limit the amount of RAM the service uses as it is the critical resource for the in-memory databases.

To tackle the memory issues, NodeJS and TypeScript add the notion of streams. Streams work in a producer-consumer architecture with an intermediate buffer with limited memory. The producer pushes information to the buffer until it is filled up and then waits until there is more room to push data. The consumer consumes the information from the buffer as fast as possible in an asynchronous manner. This way, memory usage is limited, and during the idle times of either the producer or the consumer, other tasks are scheduled and executed. Streams also have the notion of transformers that act as middleware between the producer and the consumer and transform the data

This stream implementation has been added to the controllers in order to save resources and increase the performance of the back-end service. This was necessary as rebases larger than 10, 000 resulted in crashes from the NodeJS memory allocator due to internal memory requirements.

#### 4) SCALING OUT

One of the requisites of the service is the ability to be horizontally scalable. A microservice architecture aims to provide scalability using the Single Responsibility Principle (SRP), loose coupling and decentralisation.

The SRP states that entities or services should only do one function. Here, we divided the interface implementation by different objectives; for example, database controllers only store or load data.

Loose coupling means that each service should not know much about the others yet be able to work with them. In our architecture, we expose some general interfaces that do not require the knowledge of the actual implementation to work. The persistent database should not care about the internal representation of the Redis data. It expects to receive a generic in-memory item type and work with it.

Finally, every component is decentralised in the sense that it can be instantiated in different machines and work as a single entity by using RPC.

This microservices approach allows us to use load-balancing techniques to generate a seamlessly connected system that is scalable and resistant to failure.

### D. CONSUMER APIS

For other ITS services to consume the data we gather and store, we need to provide them with an API. While there are many ways of making APIs, nowadays, REST APIs dominate the internet. REST APIs use HTTP 1.1 to communicate with the server, notify the intent, and append data.

There are two consumer APIs, one dedicated to the in-memory database for all the services running in the same MEC as the in-memory database, and another one for the persistent one in the cloud. These APIs answer user queries with the proper JSON objects.

Both web APIs are created in TypeScript using NodeJS as runtime and ExpressJS as the REST API framework. ExpressJS is a minimal and flexible NodeJS web server framework that provides a robust set of features for web and mobile applications. It is middleware based, which means that it acts as a configurable pipeline where you can create new steps to transform and process the data of the pipeline. Its native support for the TypeScript Async API makes it ideal for handling multiple clients, as web services are usually heavy on I/O which yields long waits.

The consumer API also provides a cluster manager. NodeJS implements a load-balancing library that lets the programmer define a cluster that executes as many service instances as required and balances the requests between them, scaling even further the capabilities of the APIs.

Both APIs parse the query and use a reduced version of the in-memory/persistent controller implementation to retrieve the data and send it back to the user as a JSON HTTP response.

## IV. EVALUATION

This section describes how the evaluation tests were performed, while the results are in Section V.

### A. OBU STACK

Several tests were set up to measure the performance of different scenarios. The OBU used for the tests was the Cohda Wireless' MK5 OBU. The MK5 OBU is shipped with an NXP i.MX6 DL @ 600 MHz processor and 1 GB of SDRAM. The tests measured the Round Trip Time (RTT) following the flow depicted in Figure 11.

#### 1) OBU - OBU COMMUNICATION OVER ETHERNET

In order to measure the impact of the access layer specified by the standard, a Round Trip Time (RTT) test was performed. In this RTT test, the processing time of each layer and the MQTT time were measured. To do so, we used the stack shown in Figure 2b in the setup depicted in Figure 12.

As ETSI CAM and CPM messages are sent at a maximum of 10 Hz, we have a 100 ms margin for sending and receiving the information. While OBU-OBU trips under 100ms would be acceptable, the lower we can get that number, the better.

#### 2) OBU-OBU UDP COMMUNICATION OVER ETHERNET

This test was conducted under the same scenario as *OBU - OBU communication over Ethernet* but with the multicast UDP access layer. The stack from Figure 2a was used in the setup depicted in Figure 13.

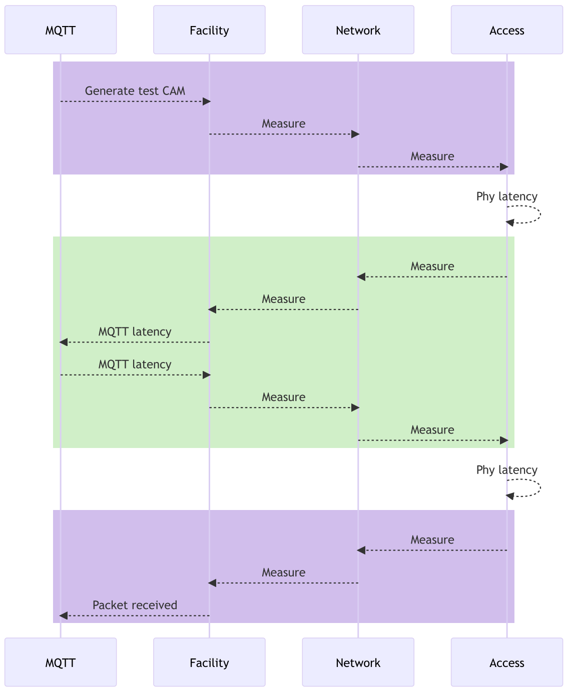


**FIGURE 11.** **Sequence diagram to measure the OBU-OBU delay.**

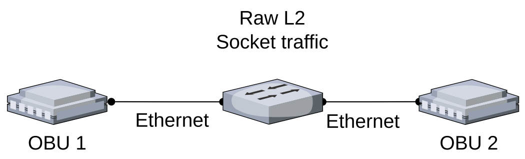


**FIGURE 12.** **OBU-OBU ethernet test setup.**

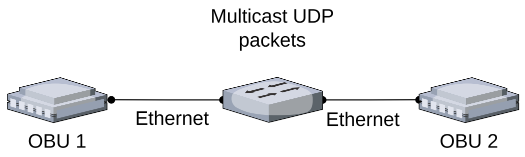


**FIGURE 13.** **OBU-OBU UDP test setup.**

### 3) OBU-OBU UDP COMMUNICATION OVER 5G

This test was conducted under the same scenario as *OBU - OBU UDP communication over Ethernet*, but using a 5G network. A 5G modem was connected to each of the OBUs and a 5G network was generated using Amarisoft's AMARI Callbox Mini as the base station. Figure 14 shows the setup.

### 4) SECURITY POLICIES

Using the test setup described in IV-A2, we tested the network layer performance using three different security policies: *none*, *dummy* and *certs*. We generated the necessary keys and certificates using Vanetza's *Certify* tool.

The tested security policies work in the following way:

- *None*: does not check if the packets are secured and just decodes them, ignoring signatures.

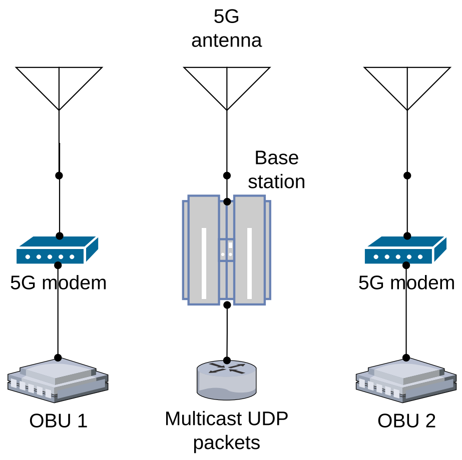


**FIGURE 14.** **OBU-OBU 5G test setup.**

- *Dummy*: checks whether packets are secured but does not validate the packet signature.
- *Certs*: checks if packets are secured and drops them if their signatures are not valid according to the supplied root certificates.

### 5) MAXIMUM NUMBER OF SIMULTANEOUS VEHICLES

Finally, we conducted a load test on one of the OBUs by connecting it via Ethernet to a synthetic CAM generator that sent an increasing number of messages per second with random intervals between them to test the maximum messages per second the application is capable of processing.

## *B. MEC HOOK*

As the only difference between the OBU software and the MEC hook is the support for multithreaded packet processing in the MEC hook, the only parameter that can vary between them is the maximum simultaneous vehicles. To test it, we performed the same test as Section IV-A5. The MEC hook is tested in a Quad Core I5-7500 @ 3 GHz processor and 8 GB of DRAM.

## *C. BACK-END SERVICE*

The most relevant tests on the back-end service are the database tests. We performed stress tests on the databases to assess their limits and behaviour under different circumstances.

Due to the memory limits on the node runtime, we first checked the difference in performance between the stream-based implementation and the direct one.

Then, we tested both databases. In order to test them, a synthetic data generator was created that, using the Fisher-Yates shuffling algorithm could create statistically uniform independent data with well-known proportions. All back-end and consumer API tests were performed on a 14 core Intel Core i9-7940X @ 3.10 GHz processor and 64 GB of DRAM.

In order to test the databases, the following tests were conducted:

- Test the timestamp index by randomly querying 100 elements in a growing database of 1, 000, 10, 000, 100, 000 and 1, 000, 000 messages.
- Test the timestamp index by randomly querying 100, 1, 000, 10, 000 and 100, 000 elements in a fixed-size database of 1, 000, 000 messages.
- Test the Mac ID index by randomly querying 100 elements in a growing database of 1, 000, 10, 000, 100, 000 and 1, 000, 000 messages.
- Test the Mac ID index by randomly querying 100, 1, 000, 10, 000 and 100, 000 elements in a fixed-size database of 1, 000, 000 messages.
- Test the Position index by randomly querying 100 elements in a growing database of 1, 000, 10, 000, 100, 000 and 1, 000, 000 messages.
- Test the Position index by randomly querying 100, 1, 000, 10, 000 and 100, 000 elements in a fixed-size database of 1, 000, 000 messages.
- Test the insertion of one element in the case of Redis and 10 elements in the case of MongoDB in a growing database of 1, 000, 10, 000, 100, 000 and 1, 000, 000 messages.

### D. CONSUMER APIS

We performed two stress tests on the consumer APIs, one for the in-memory database API and one for the persistent database API. To execute these tests, we used Artillery.io. Artillery.io is a JavaScript framework for testing REST APIs capable of generating different workload scenarios.

We set up the same three scenarios for both APIs, but using GET requests in the in-memory database API and POST requests in the persistent database API:

- *Timestamp:* query 100 elements per request with different timestamps
- *Mac ID:* query 100 elements per request with different Mac IDs.
- *Position:* query 100 elements per request with different positions

and the flow of the test was

1) *Warm up phase:* start with 100 requests per second for 15 seconds to warm up the system.
2) *Ramp up phase:* linearly increase the requests per second to 300 along 60 seconds to measure the behaviour of the system under an increasing workload.
3) *Hold 300 phase:* hold the workload at 300 requests per second for 60 seconds to measure the behaviour of the system under a static workload.
4) *Hold 150 phase:* switch the rate to 150 requests per second to test the system for abrupt workload changes for 60 seconds.
5) *Full stress:* linearly increase the request per second rate from 150 requests to 1000 requests over 60 seconds.

## V. RESULTS

In this section, the results of the evaluation tests defined in Section IV are presented and discussed.

### A. OBU STACK

#### 1) OBU-OBU COMMUNICATION OVER ETHERNET

This test measures the time each layer spends processing the packets. Three Domains are separated for the results: the communication stack, the physical network delay and the MQTT communication delay. A total of 1000 packets are sent and measured.

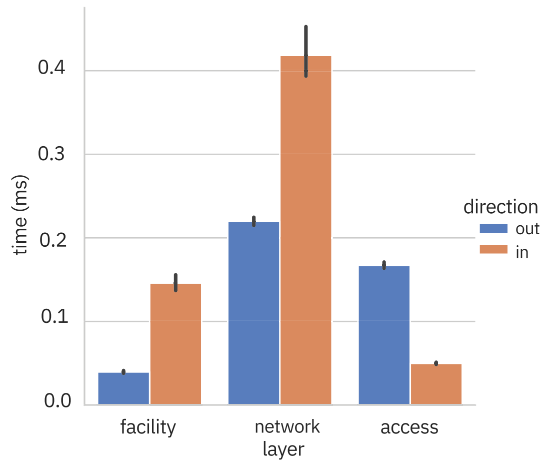


**FIGURE 15.** OBU-OBU communication over Ethernet: mean C-ITS stack delays for incoming and outgoing packets. Lines represent the 95% confidence interval.

Figure 15 shows the mean value of the stack layers under outgoing and incoming packet directions.

As expected, the processing times for the different layers are very low. The facilities layer is the most efficient one, even if it needs the convert the packets from PER-encoded bytes to JSON. RapidJSON does a good job at managing memory for repeated packet types using arena allocators and move semantics. The memory access is also cache-friendly as PER-encoded packets have a size of 256 bytes and are arranged contiguously.

Raw Ethernet packets over a wired Ethernet network were sent to study the physical network delay. These packets are the most lightweight of all the OBU-OBU performed tests due to not bridging to conventional IP. These packets have negligible physical delays, as can be seen in Figure 16.

The delay introduced by the MQTT is studied in Figure 17. The results are surprisingly slow compared to the rest of the processing times. It is two orders of magnitude bigger than the rest. The MQTT broker is the Mosquitto MQTT broker 2.0.15. This high latency seems to be produced by Mosquitto's Nagle algorithm which tries to gather multiple packets to reduce the overall number of packets sent by the interface, which yields higher throughput at the expense of latency. This is a common problem in MQTT where throughput is prioritised against latency.

**TABLE 2.** OBU-OBU communication over Ethernet: statistics for delays.

| | Facility out (ms) | Network out (ms) | Access out (ms) | Access in (ms) | Network in (ms) | Facility in (ms) | Physical network (ms) | MQTT (ms) | Total (ms) |
|---|---|---|---|---|---|---|---|---|---|
| **Mean** | 0.0399 | 0.2209 | 0.1681 | 0.0497 | 0.4152 | 0.1444 | 0.0396 | 28.6794 | 29.7572 |
| **SD** | 0.0116 | 0.0380 | 0.0289 | 0.0085 | 0.1911 | 0.0576 | 0.0758 | 3.8268 | 4.2383 |
| **95% CI** | 0.0612 | 0.2735 | 0.2245 | 0.0630 | 0.4881 | 0.2173 | 0.1306 | 32.4348 | 33.893 |
| **99% CI** | 0.0683 | 0.3649 | 0.2743 | 0.0652 | 1.1118 | 0.2539 | 0.4102 | 33.0198 | 35.5684 |
| **Max** | 0.0875 | 0.4750 | 0.3488 | 0.0695 | 2.2463 | 0.5284 | 0.6981 | 33.0784 | 37.532 |

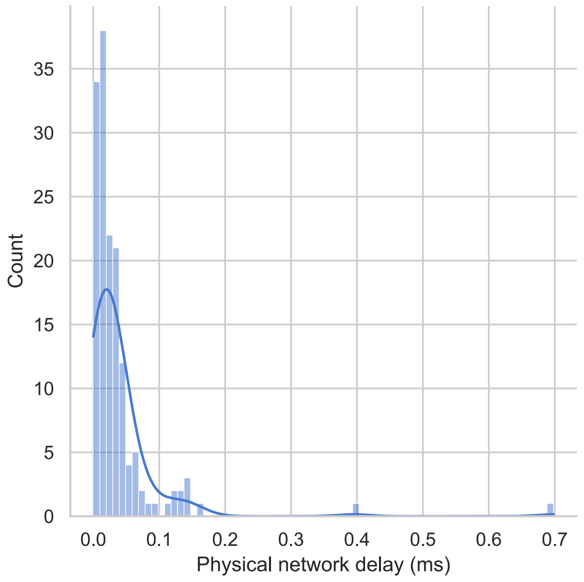


**FIGURE 16.** OBU-OBU communication over Ethernet: physical network delay distribution.

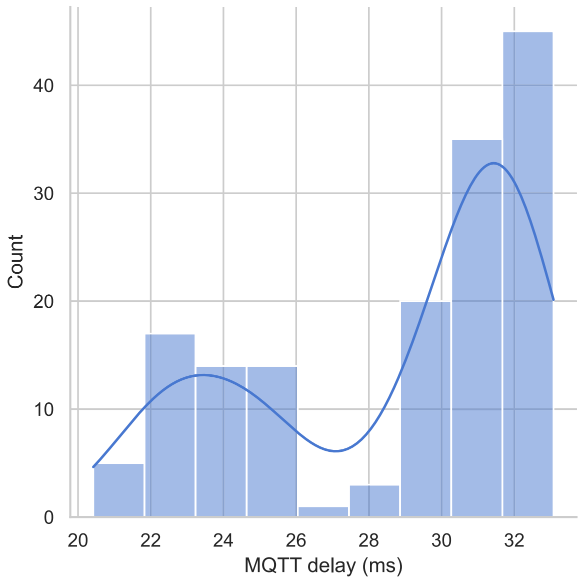


**FIGURE 17.** OBU-OBU communication over Ethernet: MQTT delay distribution.

A table summarising the results can be found in Table 2, including the following metrics: mean, standard deviation (SD), 95% confidence interval (CI), 99% CI and maximum value. The total delay of the complete chain is well below the target limit of 100 ms, even in the worst case.

### 2) OBU - OBU UDP COMMUNICATION OVER ETHERNET

This test is identical to the previous one, with a change in the access layer. This change in the access layer affects the access layer timing and the physical network delay. The access layer timing changes due to the need to construct a UDP packet on top of the C-ITS stack, and the network layer changes due to the possible need for routing and the bigger payload size.

The access layer processing times increase a bit, as can be seen in Figure 18, which in worst-case scenarios can add up to a millisecond to the previous worst time.

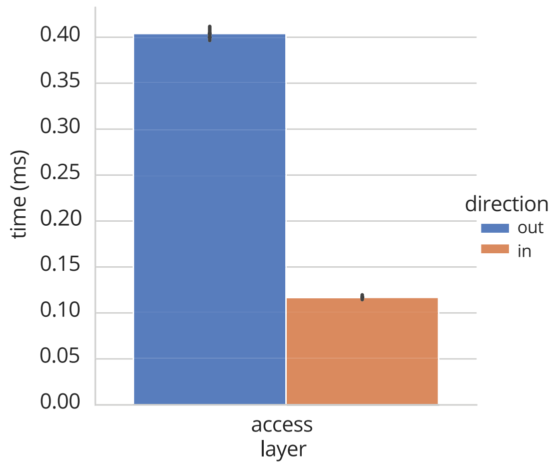


**FIGURE 18.** OBU - OBU UDP communication over Ethernet: mean C-ITS access layer delays for incoming and outgoing packets. Lines represent the 95% confidence interval.

The effect on the network delay is analysed in Figure 19. The whole distribution is shifted 0.1 ms with respect to the physical network delay introduced by raw Ethernet, with a maximum delay of 0.2003 ms. In the worst case, this setup is 1.2003 ms slower than the raw Ethernet. Still, far enough from the 100 ms limit.

### 3) OBU - OBU UDP COMMUNICATION OVER 5G

Finally, the same tests are performed using the 5G infrastructure described in Section IV-A. The only difference between the previous and current setup is the physical

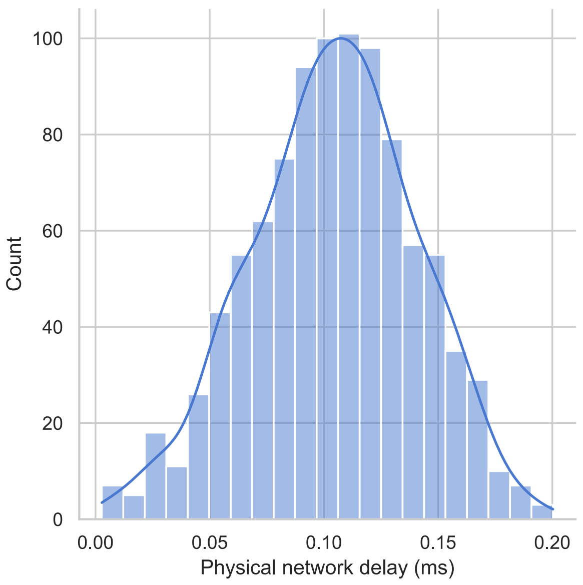


**FIGURE 19.** OBU - OBU UDP communication over Ethernet: physical network delay distribution.

layer used, so only the physical network delay is affected. The latency generated by the physical layer is depicted in Figure 20. Table 3 shows the total delay time of the complete chain.

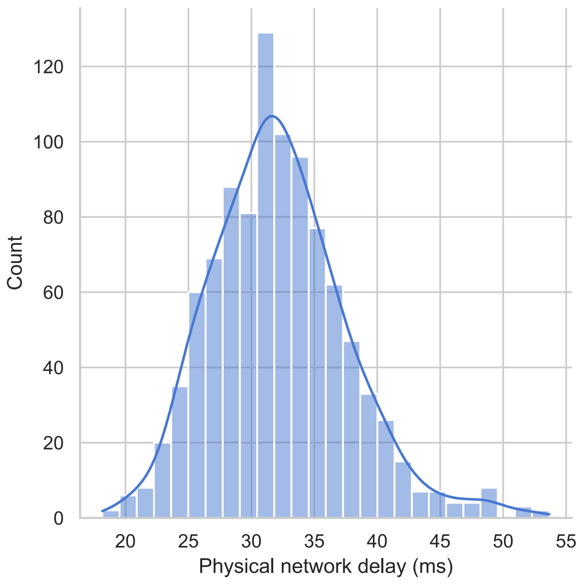
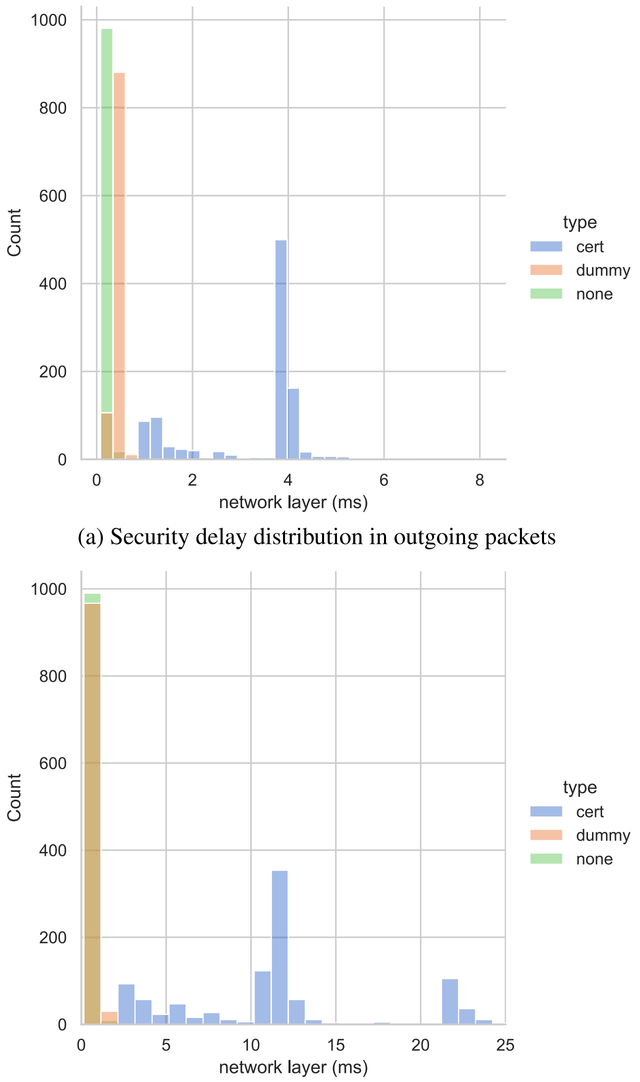


**FIGURE 20.** OBU - OBU UDP communication over 5G: physical network delay distribution.

The obtained physical layer delay can be reduced using an improved 5G network configuration, for instance, using a terahertz frequency range that has been demonstrated to achieve ultra-low latency (less than 1 ms) [53]. In any case, the obtained total delay time is less than the target limit of 100 ms.

**TABLE 3.** OBU - OBU UDP communication over 5G: total delay time.

| | Total delay time (ms) |
|---|---|
| **Mean** | 61.9536 |
| **SD** | 9.5697 |
| **95 % CI** | 75.1426 |
| **99 % CI** | 83.7973 |
| **Max** | 90.495 |



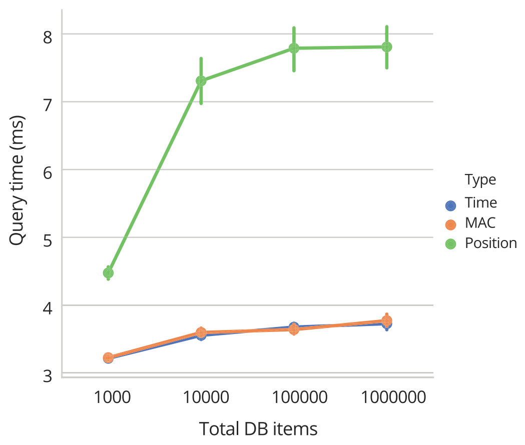
(a) Security delay distribution in outgoing packets

(b) Security delay distribution in incoming packets

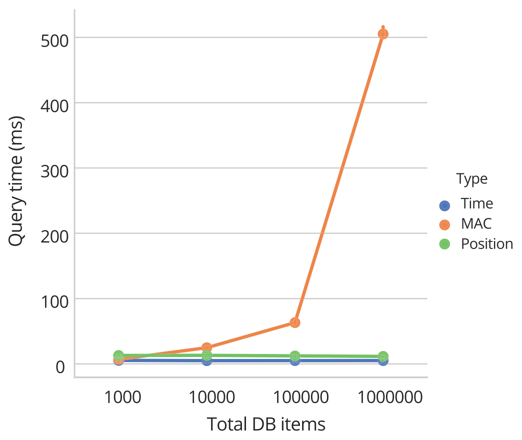
**FIGURE 21.** Delay added in the network layer by the different security policies.

## 4) SECURITY POLICIES

The delay added by the security policies for incoming and outcoming packets is depicted in Figure 21. The security policies affect the operations made in the network layer. The *None* security policy does not perform any check on the packets, the *Dummy* security policy checks if a secured header exists and the *Cert* policy checks the cryptographic signature of the packet.

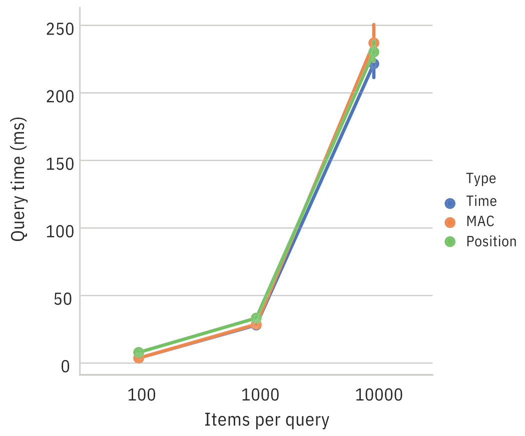


(a) Mean query time for Redis for 100 items per query with 95% confidence interval

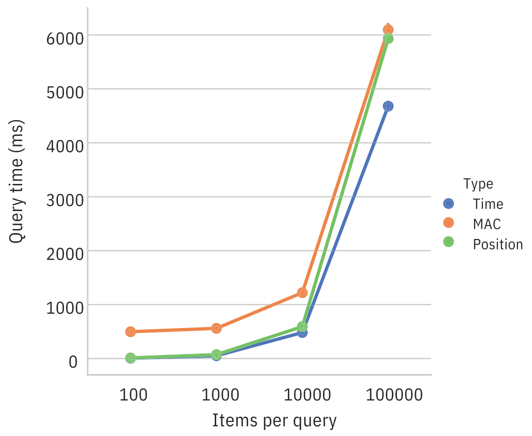


(b) Mean query time for MongoDB for 100 items per query with 95% confidence interval

**FIGURE 22. Comparison between Redis and MongoDB query performance for different database sizes.**



(a) Mean query times for a fixed database size of 1,000,000 elements and z-score filtered with 95% confidence interval



(b) Mean query times for a fixed database size of 1,000,000 elements and z-score filtered with 95% confidence interval

**FIGURE 23. Comparison between Redis and MongoDB query performance in a fixed-size database.**

While the first two policies have nearly no impact on the processing time, signing and checking packet signatures are expensive operations. It is important to note that the tested implementation does not use any hardware acceleration to perform packet signing. While this does not have much impact on outgoing packets, it adds a significant delay in incoming packets.

#### 5) MAXIMUM NUMBER OF SIMULTANEOUS VEHICLES

The maximum number of simultaneous vehicles for an OBU has been tested. The choke point for the OBU software is located at 682 vehicles sending messages at a rate of 10 messages per second. This performance is good enough as we do not expect cars to receive that much information in any case.

### *B. MEC HOOK*

The number of concurrent vehicles supported by the MEC hook is 786 under the same conditions as the OBU. One can think that this result is worse than expected if we consider the facility layer and the JSON transcoding as the bottlenecks. However, we found that the synchronised queue and the MQTT are the actual bottlenecks.

From the point of view of software engineering, this limit could be surpassed using load balancers and multiple instancing. A connection-per-thread architecture could also improve performance. More computationally capable hardware can also improve the obtained result. In any case, the number of supported concurrent vehicles is higher than the 550 vehicles supported by the infrastructure presented in [45].

## C. BACK-END SERVICE

The first step in the back-end service testing was to test whether the non-stream-based implementation could support the load. For queries bigger than 5000 elements, the virtual machine node would overload and crash, while the stream-based implementation was able to perform fine with more than 1,000,000 elements, making this the only viable option.

Then, the performance of both databases, the in-memory database (Redis) and the persistent database (MongoDB), was tested. In the first test, the timestamp, MAC and position queries were tested with a query size of 100 items for different database sizes. A side-by-side comparison between Redis and MongoDB performance can be seen in Figure 22. Note that the figure is at a logarithmic scale.

As expected, with Redis the size of the database does not have much impact on the timestamp and MAC query times, which grow linearly with the number of elements. However, when querying position values, we see a logarithmic-like growth in the query time as the total number of elements in the database grows. This is because position queries have an $\mathcal{O}(n \log(n))$ complexity.

Unlike Redis, MongoDB treats MAC IDs as text, which results in higher query times, as Figure 22 shows. A way to tackle this would be to hash the MAC IDs to optimise comparisons, as text searches usually show linear complexity, which can be seen in the plots.

In MongoDB, the complexity of spatial queries is not $\mathcal{O}n \log(n)$, as dynamic spatial subdivisions are performed, greatly decreasing query times. Figure 22 shows no increase in the position query times by the total number of items while having a bigger initial cost than the Redis counterpart.

Figure 23 shows a linear growth on both database types as the number of query items grows. Note that Redis failed to deliver more than 50,000 elements per query. This is a hard limit we cannot avoid, but we expect services that require that much data do not belong to the real-time domain.

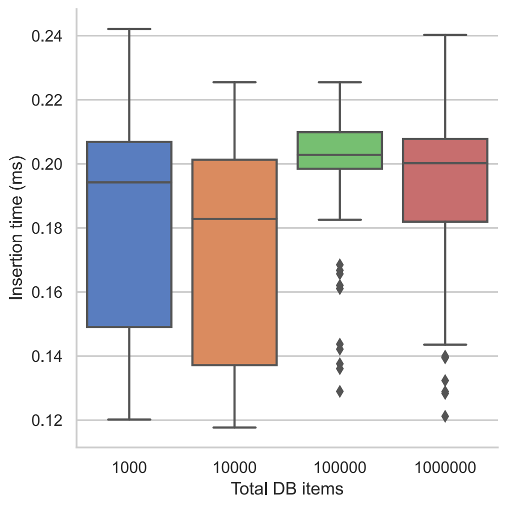
A comparison of insertion times between Redis and MongoDB for different database sizes can be found in Figure 24. A constant insertion time independent of the total database size can be appreciated in Redis, which was one of the goals we were trying to achieve by using a hashmap-like database. These insertions should not be affected by multiple clients writing simultaneously because of Redis' use of distributed lock managers that prioritise being lock-free at the cost of eventual consistency. In MongoDB, the insertion times are also higher than in Redis and slightly increase with the database size.

## D. CONSUMER API

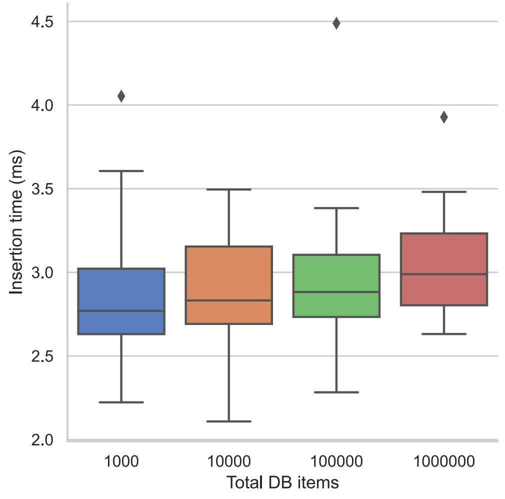
Both consumer APIs were tested using Artillery and were subjected to the same stress test flow. First, the Redis database was tested with no incorrect response.

In the worst-case scenario, the REST consumer API response time for the in-memory database does not exceed the 100 ms threshold for up to 700 requests per second while letting big timing margins 95% of the time. Once the number of concurrent requests is big, the growth of the response times becomes exponential as more time is spent in switching contexts. However, the response times are stable and predictable for a low number of requests per second, which was one of the requirements. Figure 25 shows the results in detail.

While the Redis consumer API is efficient and predictable, MongoDB has more chaotic results. Figure 26 shows the results. This consumer API can handle in predictable ways less load than the Redis one. This was expected as the MongoDB database needs to load data from disk which depends on the current state of the disk as well as the different sectors that are currently loaded in memory. The performance of the system also depends on the evolution of the load. Abrupt changes in the load really hurt the performance. The advantage that this consumer API offers is much richer query capabilities, as well as the increase in the maximum number of items clients are able to request per query.

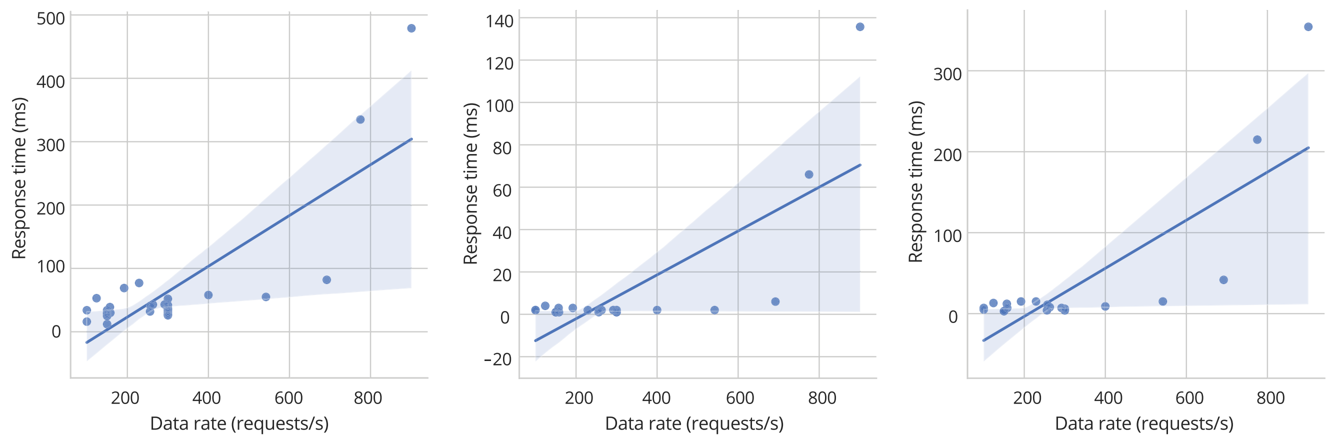


(a) Insertion time for different Redis database sizes



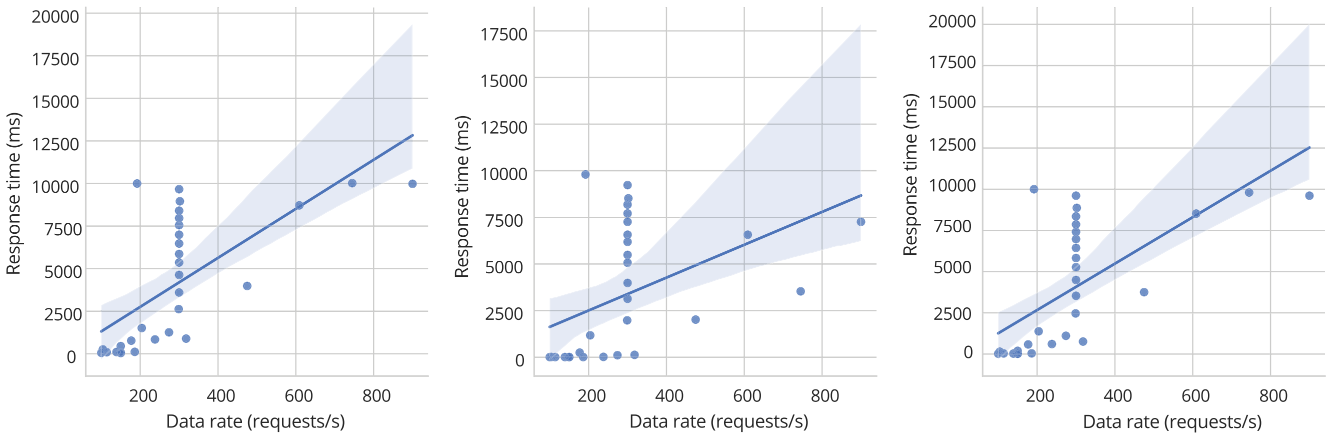
(b) Insertion time for different MongoDB database sizes

**FIGURE 24. Comparison between Redis and MongoDB insertion times.**



(a) Max response times (b) Mean response times (c) 95% CI response times

**FIGURE 25.** Redis consumer API stats, 100 elements returned per request with linear regression and 95% CI.



(a) Max response times (b) Mean response times (c) 95% CI response times

**FIGURE 26.** MongoDB consumer API stats, 100 elements returned per request with linear regression and 95% CI.

## VI. CONCLUSION

This paper proposes a novel C-ITS architecture that makes vehicle data accessible to MEC and cloud-native services. The major contributions of this work can be summarised as follows. Firstly, we have introduced an ETSI C-ITS communication stack using Multicast over 5G. Secondly, we have presented a three-layer architecture comprising OBU, MEC, and cloud components, which effectively implements the security protocols outlined in the ETSI C-ITS standard. Thirdly, we have proposed a dual-structured data storage system, comprising an in-memory database within the MEC and a persistent database in the cloud. This hybrid architecture is designed to maximise the number of supported use cases. Lastly, we have meticulously examined the impact of each architectural element on the overall pipeline latency, thus laying the groundwork for the development of more advanced architectures in the future.

We have shown the different bottlenecks MQTT, 5G and packet signing produced in the communication pipeline. In order to address the MQTT bottleneck, we should switch to another Pub/Sub message broker that can offer better scalability and performance, like Kafka or RabbitMQ. To reduce the effect of packet signing and signature validation, we should change the security back-end to a hardware-accelerated one, as it would speed up the calculations reducing the impact. The lower the time we spend communicating the packets, the more time clients can spend on their own processing.

An efficient MEC native back-end service has been developed and tested with very positive results. It can handle up to 786 concurrent vehicles which is 115,24% better than the OBU implementation. We expected a bigger increase in the capacity due to the parallelisation of the facility layer, but in the end, it turned out to be the fastest layer of all which translated into a lower gain. In order to fix this, a lock-free approach to the synchronised queue should be taken. Additionally, a message broker connection per thread in the thread pool could also be used to make the MQTT transmission, which is the most expensive one, independent from each thread, reducing the lock times and thus greatly increasing the efficiency at the cost of more connections to the broker.

The proposed architecture uses cache databases for real-time data processing and persistent databases for the rest. Redis and MongoDB were chosen for the cache and the persistent database respectively. As expected, Redis had better base performance and predictability than MongoDB in every type of query or insertion, but the growth of the cost by the number of elements was much worse than MongoDB's. Tests also showed poor behaviour from MAC IDs in the MongoDB. Since we were storing the MAC IDs as text, the complexity of the queries grew linearly with the number of elements in the database which is undesirable. A much better approach could be made by hashing the MAC ID, so it can be stored as a number instead of a text string in MongoDB.

Two consumer REST APIs using NodeJS and ExpressJS serve as access APIs for the cache and persistent data. The Redis API offers more scalability and predictability, while the other offers richer query capabilities. As these APIs are based on middleware they are easily extendable to include authentication, routing or other mechanisms.

The present study is limited to handling ETSI CAM and CPM messages. However, the architecture can easily be extended for accommodating other ETSI message formats. As future work, other message types beyond the ones defined in the ETSI C-ITS standard could be studied and implemented. For instance, camera images could be embodied in messages and shared in a similar way.

## ACKNOWLEDGMENT

Content reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

## REFERENCES

[1] M. U. Ghazi, M. A. K. Khattak, B. Shabir, A. W. Malik, and M. S. Ramzan, "Emergency message dissemination in vehicular networks: A review," *IEEE Access*, vol. 8, pp. 38606–38621, 2020.

[2] K. V. Katsaros, E. Liotou, F. Moscatelli, T. Rokkas, G. Drainakis, E. Bonetto, D. Brevi, D. Klonidis, I. Neokosmidis, and A. Amditis, "Enabling far-edge intelligent services with network applications: The automotive case," *IEEE Internet Things Mag.*, vol. 5, no. 4, pp. 122–128, Dec. 2022.

[3] N. Slamnik-Kriještorac, F. Z. Yousaf, G. M. Yilma, R. Halili, M. Liebsch, and J. M. Marquez-Barja, "Edge-aware cloud-native service for enhancing back situation awareness in 5G-based vehicular systems," *IEEE Trans. Veh. Technol.*, vol. 73, no. 1, pp. 660–677, Jan. 2024.

[4] G. Velez, J. Perez, and A. Martin, "5G MEC-enabled vehicle discovery service for streaming-based CAM applications," *Multimedia Tools Appl.*, vol. 81, no. 9, pp. 12349–12370, Apr. 2022.

[5] Z. Fernández, A. Martín, J. Pérez, M. García, G. Velez, F. Murciano, and S. Peters, "Challenges and solutions for service continuity in inter-PLMN handover for vehicular applications," *IEEE Access*, vol. 11, pp. 8904–8919, 2023.

[6] M. García, G. Velez, J. Pérez, Á. Martín, Z. Fernández, and N. Aginako, "Edge dynamic map architecture for C-ITS applications," 2023, *arXiv:2308.02197*.

[7] K. Kiela, V. Barzdenas, M. Jurgo, V. Macaitis, J. Rafanavicius, A. Vasjanov, L. Kladovscikov, and R. Navickas, "Review of V2X-IoT standards and frameworks for ITS applications," *Appl. Sci.*, vol. 10, no. 12, p. 4314, Jun. 2020.

[8] V. Maglogiannis, D. Naudts, S. Hadiwardoyo, D. van den Akker, J. Marquez-Barja, and I. Moerman, "Experimental V2X evaluation for C-V2X and ITS-G5 technologies in a real-life highway environment," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 2, pp. 1521–1538, Jun. 2022.

[9] J. Zhao, X. Gai, and X. Luo, "Performance comparison of vehicle networking based on DSRC and LTE technology," in *Proc. 6th Int. Conf. Intell. Transp. Eng. (ICITE)*, Z. Zhang, Ed. Singapore: Springer, 2022, pp. 730–746.

[10] T. Petrov, L. Sevcik, P. Pocta, and M. Dado, "A performance benchmark for dedicated short-range communications and LTE-based cellular-V2X in the context of vehicle-to-infrastructure communication and urban scenarios," *Sensors*, vol. 21, no. 15, p. 5095, Jul. 2021.

[11] M. Tabassum, F. H. Bastos, A. Oliveira, and A. Klautau, "NR sidelink performance evaluation for enhanced 5G-V2X services," *Vehicles*, vol. 5, no. 4, pp. 1692–1706, Nov. 2023.

[12] T. Petrov, P. Pocta, and T. Kovacikova, "Benchmarking 4G and 5G-based cellular-V2X for vehicle-to-infrastructure communication and urban scenarios in cooperative intelligent transportation systems," *Appl. Sci.*, vol. 12, no. 19, p. 9677, Sep. 2022.

[13] *Intelligent Transport Systems (ITS); Communications Architecture*, document EN 302 665 Version 1.1.1, ETSI, European Norm, 2010.

[14] *Intelligent Transport Systems (ITS); Cooperative ITS (C-ITS); Release 1*, document TR 101 607, Version 1.2.1, ETSI, 2020.

[15] Z. Liu, J. Weng, J. Ma, J. Guo, B. Feng, Z. Jiang, and K. Wei, "TCEMD: A trust cascading-based emergency message dissemination model in VANETs," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4028–4048, May 2020.

[16] Z. Liu, L. Wan, J. Guo, F. Huang, X. Feng, L. Wang, and J. Ma, "PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks," *IEEE Trans. Veh. Technol.*, early access, 2023, doi: 10.1109/TVT.2023.3340723.

[17] Z. Liu, J. Weng, J. Guo, J. Ma, F. Huang, H. Sun, and Y. Cheng, "PPTM: A privacy-preserving trust management scheme for emergency message dissemination in space–air–ground-integrated vehicular networks," *IEEE Internet Things J.*, vol. 9, no. 8, pp. 5943–5956, Apr. 2022.

[18] *Intelligent Transport Systems (ITS); GeoNetworking; Part 3: Network Architecture*, ETSI Standard EN 302 636-3, Version 1.1.2, ETSI, Sophia Antipolis, France, 2014.

[19] L. Hou, M. A. Gregory, and S. Li, "A survey of multi-access edge computing and vehicular networking," *IEEE Access*, vol. 10, pp. 123436–123451, 2022.

[20] S. Ketu and P. K. Mishra, "Cloud, fog and mist computing in IoT: An indication of emerging opportunities," *IETE Tech. Rev.*, vol. 39, no. 3, pp. 713–724, May 2022.

[21] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for Internet of Things realization," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2961–2991, 4th Quart., 2018.

[22] H. Ma, S. Li, E. Zhang, Z. Lv, J. Hu, and X. Wei, "Cooperative autonomous driving oriented MEC-aided 5G-V2X: Prototype system design, field tests and AI-based optimization tools," *IEEE Access*, vol. 8, pp. 54288–54302, 2020.

[23] W. Hammedi, B. Brik, and S. M. Senouci, "Toward optimal MEC-based collision avoidance system for cooperative inland vessels: A federated deep learning approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 24, no. 2, pp. 2525–2537, Feb. 2023.

[24] C. Quadri, V. Mancuso, M. A. Marsan, and G. P. Rossi, "Edge-based platoon control," *Comput. Commun.*, vol. 181, pp. 17–31, Jan. 2022.

[25] P. Sroka and A. Kliks, "Towards edge intelligence in the automotive scenario: A discourse on architecture for database-supported autonomous platooning," *J. Commun. Netw.*, vol. 24, no. 2, pp. 192–208, Apr. 2022.

[26] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 668–682, Mar. 2019.

[27] Z. Wang, G. Sun, H. Su, H. Yu, B. Lei, and M. Guizani, "Low-latency scheduling approach for dependent tasks in MEC-enabled 5G vehicular networks," *IEEE Internet Things J.*, vol. 11, no. 4, pp. 6278–6289, 2023.

[28] B. Li, F. Chen, Z. Peng, P. Hou, and H. Ding, "Mobility-aware dynamic offloading strategy for C-V2X under multi-access edge computing," *Phys. Commun.*, vol. 49, Dec. 2021, Art. no. 101446.

[29] S. D. A. Shah, M. A. Gregory, S. Li, R. d. R. Fontes, and L. Hou, "SDN-based service mobility management in MEC-enabled 5G and beyond vehicular networks," *IEEE Internet Things J.*, vol. 9, no. 15, pp. 13425–13442, Aug. 2022.

[30] B. Cao, Z. Li, X. Liu, Z. Lv, and H. He, "Mobility-aware multiobjective task offloading for vehicular edge computing in digital twin environment," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 10, pp. 3046–3055, Oct. 2023.

[31] R. J. Kauffman, D. Ma, and M. Yu, "A metrics suite of cloud computing adoption readiness," *Electron. Markets*, vol. 28, no. 1, pp. 11–37, Feb. 2018.

[32] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu, "Lessons learned from migrating complex stateful applications onto serverless platforms," in *Proc. 12th ACM SIGOPS Asia–Pacific Workshop Syst.*, Aug. 2021, pp. 89–96.

[33] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. thesis, Dept. Inf. Comput. Sci., Univ. California Irvine, Irvine, CA, USA, 2000.

[34] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service APIs," *IEEE Trans. Services Comput.*, vol. 14, no. 4, pp. 957–970, Jul. 2021.

[35] L. D. S. B. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture," in *Proc. 7th Int. Conf. Inf. Technol. Res. (ICITR)*, Dec. 2022, pp. 1–6.

[36] Z. Kegenbekov and A. Saparova, "Using the MQTT protocol to transmit vehicle telemetry data," *Transp. Res. Proc.*, vol. 61, pp. 410–417, Jan. 2022.

[37] N. Jain, "The future of database services: Cloud database," in *Proc. Int. Conf. Algorithms, Methodol., Models Appl. Emerg. Technol. (ICAMMAET)*, Feb. 2017, pp. 1–5.

[38] J. Yang, L. Chen, and J. Bai, "Redis automatic performance tuning based on eBPF," in *Proc. 14th Int. Conf. Measuring Technol. Mechatronics Autom. (ICMTMA)*, Jan. 2022, pp. 671–676.

[39] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, and F. Ismaili, "Comparison between relational and NOSQL databases," in *Proc. 41st Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2018, pp. 216–221.

[40] Å. Hugo, B. Morin, and K. Svantorp, "Bridging MQTT and Kafka to support C-ITS: A feasibility study," in *Proc. 21st IEEE Int. Conf. Mobile Data Manage. (MDM)*, Jun. 2020, pp. 371–376.

[41] B. Ayaz, N. Slamnik-Kriještorac, and J. Marquez-Barja, "Data management platform for smart orchestration of decentralized and heterogeneous vehicular edge networks," in *Proc. ACM Conf. Inf. Technol. Social Good*. New Yorkth, NY, USA: Association for Computing Machinery, Sep. 2022, pp. 118–124.

[42] D. Rocha, G. Teixeira, E. Vieira, J. Almeida, and J. Ferreira, "A modular in-vehicle C-ITS architecture for sensor data collection, vehicular communications and cloud connectivity," *Sensors*, vol. 23, no. 3, p. 1724, Feb. 2023.

[43] E. Vieira, J. Almeida, J. Ferreira, T. Dias, A. V. Silva, and L. Moura, "A roadside and cloud-based vehicular communications framework for the provision of C-ITS services," *Information*, vol. 14, no. 3, p. 153, Mar. 2023.

[44] E. Vieira, J. Almeida, J. Ferreira, and P. C. Bartolomeu, "VERCO: A privacy and data-centric architecture for verifiable cooperative maneuvers," *Veh. Commun.*, vol. 42, Aug. 2023, Art. no. 100614.

[45] F. Raviglione, C. M. R. Carletti, C. Casetti, F. Stoffella, G. M. Yilma, and F. Visintainer, "S-LDM: Server local dynamic map for vehicular enhanced collective perception," in *Proc. IEEE 95th Veh. Technol. Conf. (VTC-Spring)*, Jun. 2022, pp. 1–5.

[46] R. Riebl. *Vanetza*. Accessed: Mar. 27, 2024. [Online]. Available: https://web.archive.org/web/20221020125303/

[47] *Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 4: Geographical Addressing and Forwarding for Point-to-Point and Point-to-Multipoint Communications*, ETSI Standard EN 302 636-4-1, Version 1.2.0, ETSI, Sophia Antipolis, France, 2014.

[48] A. M. A. Sai, D. Reddy, P. Raghavendra, G. Y. Kiran, and R. V. Rejeenth, "Producer-consumer problem using thread pool," in *Proc. 3rd Int. Conf. Emerg. Technol. (INCET)*, Dinesh Reddy, Peru, May 2022, pp. 1–5.

[49] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2012.

[50] *Intel$^{®}$ 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Santa Clara, CA, USA, 2023.

[51] A. T. Kabakus and R. Kara, "A performance evaluation of in-memory databases," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 29, no. 4, pp. 520–525, Oct. 2017.

[52] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs Cassandra," in *Proc. Int. Conf. Comput. Sci. Softw. Eng.*, Jul. 2013, pp. 14–22.

[53] R. Ford, M. Zhang, M. Mezzavilla, S. Dutta, S. Rangan, and M. Zorzi, "Achieving ultra-low latency in 5G millimeter wave cellular networks," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 196–203, Mar. 2017.

**JAVIER ARIN** received the bachelor's degree in communications electronic engineering and the master's degree in telecommunications engineering from the University of Navarra, Spain, in 2021 and 2022, respectively. During the bachelor's degree, he was an Intern Student with the Electronics and Communications Department. During his internship at Vicomtech, he worked on the master's thesis about C-ITS communications and data sharing. His research interests include embedded computing, software engineering, and low-level firmware.

**GORKA VELEZ** received the M.Sc. degree in electronic engineering from the University of Mondragon, Spain, in 2007, and the Ph.D. degree from the University of Navarra, Spain, in 2012. He is currently leading the connected and cooperative situation awareness research line at the Intelligent Transportation Systems (ITS) and Engineering Department, Vicomtech. He is a Technical Coordinator of the H2020 project 5GMETA funded by the European Commission. He is also involved in several other CCAM research projects, including 5G-IANA and PoDIUM.

**PAUL BUSTAMANTE** is currently a Researcher with the CEIT-Basque Research and Technology Alliance (BRTA). He is also an Associate Professor in informatics and telecommunications systems with the School of Engineering at San Sebastián (Tecnun), University of Navarra. He had been working in some industrial companies. He is involved in the intelligent systems for Industry 4.0 Group, CEIT, where his experience is mainly in the design of wireless sensor networks (WSNs), developing both hardware and software using standard protocols, such as ZigBee or Bluetooth, or developing ad-hoc communications protocols. Moreover, he has been involved in national and international projects of RF design, communications systems with GSM/GPRS, and internet protocols. He is authoring some articles in journals and conferences.