Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Middleware Support for Communication in Connected Vehicles

Prabhjot Singh

August 9, 2021

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Abstract

V2X technologies have seen great progress in vehicular applications in the last decade. this evolution can be credited to availability of sensor data made available by evolution in IOT. this project is motivated to design a communication system that fulfils advanced V2X applications, requiring high throughput, low latency and high reliability. the background study conducted as part of the project concludes key technology choices made in the project, which are, i) to base the design on IEEE standards based physical layer and data link layer, ii) base the core distributed messaging system on Kafka, iii)develop a collection/delivery system for CVs that takes close considerations to challenges involved in bridge them to core messaging system. the proposed design is optimized for exchange of data streams between CVs and other vehicular infrastructure. for that purpose, design purposes reliability and buffering mechanisms that can provide the continuous flow of data. Close attention has been paid to challenges that are faced with communication between CVs and RSUs, for instance, intermittent connectivity and high reliability requirement which it poses. Kafka forms the backbone of the proposed design as the messaging system that various services can produce and consume data from. while Kafka is a system capable of fulfilling latency and throughput requirements set forth by advanced V2X, it is a rigid system that supports static topics for different data streams. the collection system proposed is based on RabbitMQ which provides dynamic queues to support CVs and bridge them to messaging backbone. RabbitMQ uses AMQP protocol designed to fulfil low latency requirements. The design is implemented in a simulated V2X environment that takes into consideration the handover of CV between RSUs and simulates message drops to test the capabilities of design. the implementation draws results and conclusions on the latency and throughput performance of the design.

Acknowledgements

I would like to thank Professor Vinny Cahill for introducing the field to me and extending his continued support throughout the course of the project. I offer my sincere appreciation for the feedback and support provided by him.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Project Overview . . . 3

1.2.1 Distributed Messaging System . . . 3

1.2.2 Collection/delivery system for CVs . . . 4

1.2.3 Research Aims . . . 4

1.2.4 Road Map . . . 5

2 Background **6**

2.1 V2X Standards . . . 6

2.1.1 IEEE standards . . . 6

2.1.2 C-V2X standards . . . 7

2.1.3 Performance comparison . . . 8

2.2 Existing Middlewares . . . 10

2.2.1 An overview of publish subscribe . . . 10

2.2.2 Apache Kafka . . . 11

2.2.3 RabbitMQ . . . 12

2.2.4 Others . . . 13

2.2.5 Comparative evaluation . . . 14

2.3 IOT Protocols . . . 15

2.3.1 An overview of prevalent IOT protocols . . . 16

2.3.2 Comparison . . . 17

2.4 Similar Works . . . 19

3 Design **23**

3.1 Design considerations . . . 23

3.1.1 Basing on IEEE standards . . . 23

3.1.2 Kafka as core of delivery system . . . 24

3.1.3 RabbitMQ as bridge between Kafka and CVs . . . 24

3.1.4 Programming language . . . 24

3.1.5 Other considerations . . . 25
3.2 Initial attempts . . . 25
3.3 Overall architecture . . . 27
3.4 Lifecycle phases . . . 28
3.4.1 Overall lifecycle . . . 29
3.4.2 CV-RSU Handshake . . . 29
3.4.3 Producing data . . . 31
3.4.4 Consuming data . . . 31
3.4.5 Termination of connection and connecting to next RSU . . . 32
3.4.6 Buffer/Caching mechanisms . . . 33
3.4.7 Format of the messages . . . 33
3.4.8 Clean up mechanisms . . . 33
3.5 Other measures to enhance performance . . . 34
3.6 Extensibility . . . 34

4 Implementation 36
4.1 Overall implementation . . . 36
4.2 Simulating a V2X environment . . . 36
4.3 Simulations and results . . . 38
4.3.1 Average CV to RSU latency as CV moves between RSUs . . . 38
4.3.2 Average CV to RSU latency vs time . . . 40
4.3.3 Throughput Analysis . . . 40

5 Summary and conclusions 42
5.1 Future Works . . . 42

List of Figures

1.1 Middleware support provided by proposed design 2

2.1 Kafka Architecture (1) 11
2.2 RabbitMQ(AMQP) Architecture (1) 13

3.1 Overall architecture of initial designs 26
3.2 Overall Architecture 27
3.3 Overall lifecycle 30

4.1 Average re-transmission attempts vs position 38
4.2 average latency vs position 39
4.3 average latency vs position 39
4.4 average latency vs time 40

Nomenclature

AV	Automated Vehicle
CV	Connected Vehicle
RSU	Roadside Unit
V2X	Vehicle to everything
LDM	Local Dynamic Map
IEEE	Institute of Electrical and Electronics Engineers
IOT	Internet of Things
CAM	Cooperative awareness messages
MAC	Media Access Control
CSMA/CA	Carrier-sense Multiple Access
WLAN	Wireless local area network
DSRC	Dedicated short range communications
LTE	Long Term Evolution
5G NR V2X	5G based New Radio V2X
HARQ	Hybrid Automatic repeat request
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
ITS-G5	Intelligent Transport systems
QoS	Quality of service
AMQP	Advanced Message Queuing Protocol
MQTT	Messsage Queuing Telemetry Transport
XMPP	Extensible Messaging and Presence Protocol
HTTP	Hypertext Transfer Protocol
CoAP	Constrained Application Protocol
DDS	Data Distribution Service
HDFS	Hadoop Distributed File System
ROS	Robotic Operating System
CVRIA	Connected Vehicles Reference Implementation Architecture
USDOT	US Department of Transportation
RPM	Revolutions per minute
API	Application Programming interface
ETSI	The European Telecommunication Standards Institute
C-ITS	Cooperative Intelligent Transport System

1 Introduction

This dissertation presents design for middleware system that caters to communication requirements of advanced V2X applications such as traffic management and road safety applications that demand high awareness of vehicular environment. data sensors in CVs and RSUs, and information available from public services can provide awareness of road environment that the vehicles on the road require to realize these advanced V2X applications. the challenge lies in reliable delivery and processing of this available data while fulfilling low latency and high throughput. the design is optimized for communicating different types of data streams available from sensors in CVs and from external services to CVs. Figure 1.1 gives an overview of middleware support that the proposed design provides. Apache Kafka is proposed as the basis for core delivery system to communicate data streams between on-road V2X infrastructure with management and services. data collection and delivery system is designed for CVs on road, to bridge Kafka based distributed messaging system and CVs. the data collection/delivery system uses RabbitMQ with AMQP for communication with CVs and is designed to provide seamless flow of data by employing in memory buffer systems to maximize throughput and latency performances. reliable re transmission mechanisms are designed, taking throughput and latency performances in consideration to provide reliability needed in the collection/delivery system. the design gives well thought considerations to reserving scope for pre-processing at the edge of infrastructure.

The background study conducted as part of the dissertation justifies the choices of technologies made throughout the design. The design bases physical and data link layer on IEEE based V2X standards which provide better performance than other available alternatives. IEEE based standards rely on RSUs to connect with CVs, which introduces challenge of intermittent connectivity that design takes into consideration. Kafka at the core of proposed infrastructure provides high throughput and low latency delivery to entities involved in infrastructure. RabbitMQ, employed at edge, is a lightweight message queuing system capable of scaling up to the expected connections between a RSU and CVs.

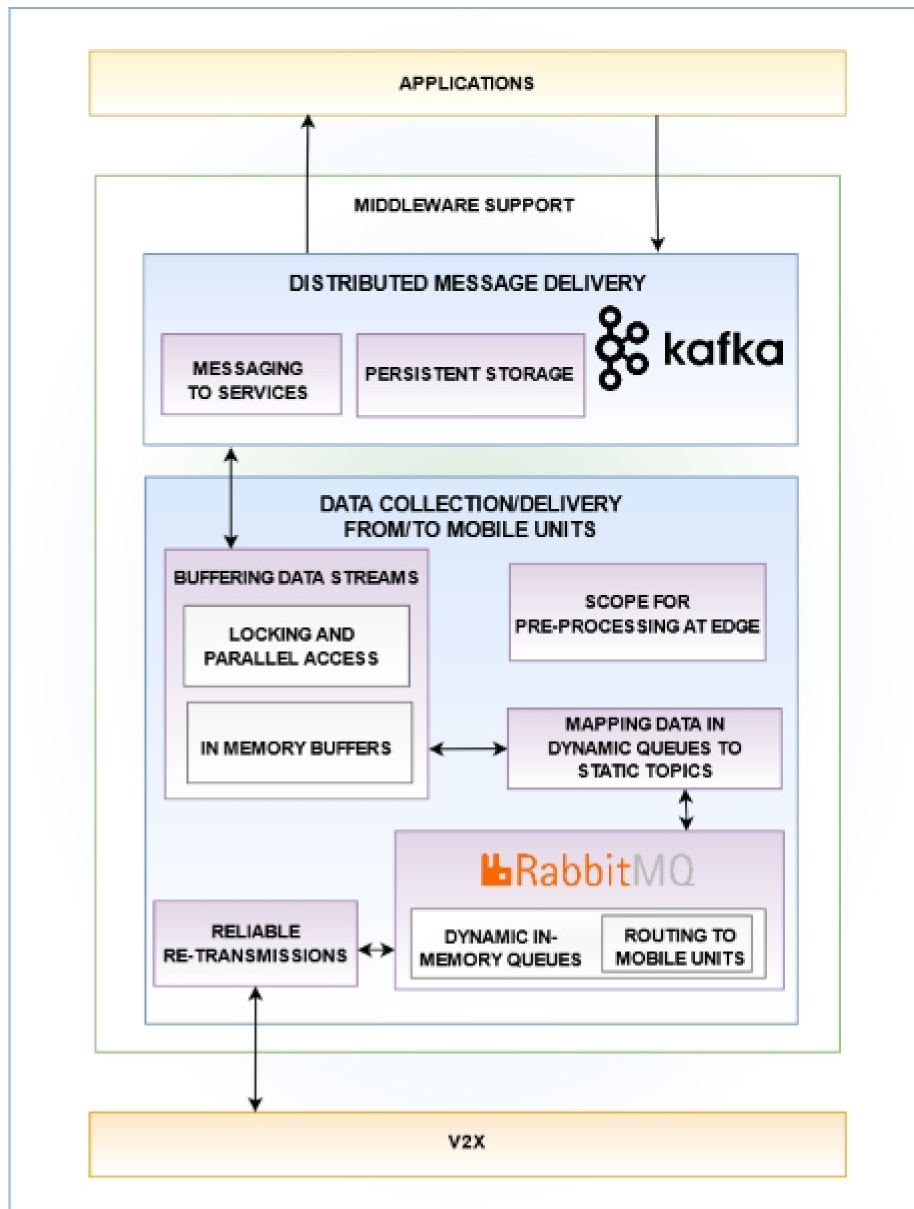

Figure 1.1: Middleware support provided by proposed design

1.1 Motivation

Recent years have seen much progress in vehicular safety, traffic management and evolution of autonomous vehicles. this evolution in vehicles have been long awaited. a decade ago in 2010, vehicular communication standards and systems all around the world were still in very early phases. this recent progress can largely be credited to evolution of IOT concepts and communications, or in simpler terms, the availability of information and capability of networks in today's connected world. V2X or vehicle to everything is the class of communication concepts and systems dealing with evolution in the field of connectivity in vehicles, roadside infrastructures and exploring possible vehicular applications. V2X has largely been evolved by standards regulated by government bodies and focuses on safety

applications and traffic management. basic V2X applications include sending warnings about weather and road conditions, notifications about speed limits and safety rules. this was made possible by CAM messages, which are lightweight, sent aperiodically with low frequency with requirements on a moderate level of latency. the advanced applications like autonomous driving and precautionary safety more effective than ever, however, require co-operative awareness of surroundings that current V2X infrastructures are not fully equipped to deliver. studies like (2) have spectated that V2X communication systems of future, are likely to be distributed delivery systems making use of different data sources. support to transport data from distributed sources to distributed targets is needed and it has to be done with high reliability while maintaining low latency and high throughput.
So, this study aims at proposing a design that is capable of collection and delivery of data at high frequencies to the CVs on road.

1.2 Project Overview

This section presents an overview of the project starting with research aims, and then giving a high level description of the two main components in the design, namely core distributed messaging system, and collection/delivery system, while covering some key considerations behind their design. last subsection presents the roadmap of the presented thesis outlining it's structure.

1.2.1 Distributed Messaging System

data streams collected from vehicles have usefulness in variety of areas. traffic services can process the data collected from vehicles to obtain meaningful analysis of the road environment. public services like health and safety can make use of the data collected to track accidents on the road and provide emergency services. In addition, enforcement authorities can make use of the collected data for violations and tracking. vice-versa, CVs on the road can make use of data from a variety of sources. primary application would be traffic services providing analysis of the environment that CVs are in. In addition, external services like weather can make use of the delivery system to deliver weather related notifications and alerts. public services can make use of the system to deliver public alerts. this calls for a highly scalable distributed messaging system for CVs.

Apache Kafka was developed by LinkedIn to address High throughput requirements for moving their log files around. Since, it has developed into a highly scalable system with many versatile applications. Its usefulness in Connected vehicular technology is now one of the topics of interest. Apache Kafka provides high scalability at each of producing, brokering and consuming stages. Kafka is a well tested and rapidly evolving system with active

community. studies like (2) have shown the effectiveness of Kafka in delivering high throughput and low latency needed for V2X applications. hence, the proposed distributed messaging system is based on Apache Kafka.

1.2.2 Collection/delivery system for CVs

After background study for current progress of V2X systems on physical and data link layer, choice was made to design an infrastructure that uses IEEE based standards at physical and data link layer. IEEE based standards are well tested and mature class of technology with evolving V2X standards. the newly evolved standards are backward compatible with already existing infrastructure based on previous IEEE standards. IEEE standards provide superior overall performance than the other available alternatives in terms of latency and throughput. but this choice introduces inherent challenges like intermittent connectivity between RSUs and CVs.

Kafka based delivery system alone can not fulfil these requirements. so, a collection/delivery system based on RabbitMQ is designed to form a bridge between the Kafka based message delivery system and CVs. RabbitMQ is a widely used message broker for many IOT applications. RabbitMQ originally started by implementing Advanced Message Queuing Protocol (AMQP) and has since extended to support MQTT and several other IOT protocols. For design in this dissertation, AMQP was chose to be the suitable choice after background study on prevalant IOT standards. RabbitMQ handles messages in DRAM and is optimized for empty or nearly empty queues. It is able to provide single digit millisecond latency in favourable conditions. RabbitMQ also provides flexibility in terms of dynamic queue declarations and deletions with cheap overheads.

1.2.3 Research Aims

This dissertation aimed to develop a V2X communication system with main goal to design a system that could provide seamless flow of data streams from CVs and various services that could make use of data generated in CV and vice-versa. In order to realize this goal, two main challenges that this dissertation aimed to address are given below:

- Designing an infrastructure that could provide high throughput, low latency and high reliability throughout the communication.
- addressing challenges introduced by IEEE based standards between CV and RSU, including intermittent connectivity and requirement for high reliability.

1.2.4 Road Map

The remaining part of the dissertation is organized as follows:

- Chapter 2 will provide background on V2X standards and available technologies relevant to the study, justifying the choices of technologies adopted for the the final design.
- Chapter 3 will present design of the system and technical workings of the proposed design.
- Chapter 4 presents the implementation of the design and feasibilty tests done and results produced as part of this dissertation.
- Chapter 5 will conclude the thesis with a summary and scope of the project for future.

2 Background

This chapter presents background study relevant to this dissertation. section 2.1 covers current V2X standards at physical and data link layer. A comparison of current middlewares with usefulness in V2X is done in section 2.2, then comparison of prevalent IOT protocols in context of V2X is done in section 2.3. section 2.4 reviews the similar works that have been conducted in recent years.

2.1 V2X Standards

This section presents currently available V2X standards at physical and data link layer. physical layer is responsible for transporting, receiving and sending data bits across the communication system. characteristics of physical layer directly affect the physical data rates (3). mechanisms like error checking, frame synchronization and MAC are part of data link layer. a look at physical and data link layer of these already present standards and their comparison reveals best possible choice to form foundation for a V2X infrastructure capable of fulfilling requirements for V2X applications.

Two classes of standards have emerged in that context, first class is formed by IEEE standards developed for vehicular applications which resemble physical and data link characteristics of WLAN technology standards, this class of standards is often commonly referred to as DSRC. the second class, commonly referred to as C-V2X is formed by standards resembling physical and data link characteristics of cellular communication technologies. In this section, first, a brief overview of each of these classes is covered, followed by a comparison concluding the choice of standard to base an infrastructure upon for this dissertation.

2.1.1 IEEE standards

IEEE standards have been around since 2010 making it the more traditional and well tested alternative. The first IEEE standard for V2X was IEEE 802.11p developed by making enhancements to WLAN standard IEEE 802.11a to accommodate challenges like high

mobility in connected vehicles. IEEE 802.11p can provide transmission rates 3 to 27 Mbps and maximum outdoor range of 1000m in some geographical environments for some applications (3). (4) shows that IEEE 802.11p delivers a performance good enough to support applications requiring end to end latency around 100msec as long as vehicle density does not exceed a certain limit. the MAC layer employs CSMA/CA without exponential back off as it was in case of 802.11a. IEEE 802.11bd is the latest standard for V2X in this class and was developed as enhanced version of 802.11p to fulfill low latency, high reliability, high throughput and high range requirements by borrowing characteristics and enhancements from successors of 802.11a i.e., 802.11 n/ac/ax (5). other requirements in development of 802.11bd were supporting high relative speeds and provide interoperability, backward compatibility and coexistence with 802.11p as it is the standard that is already in use by existing connected vehicles on the road. to fulfil these requirements 802.11bd standard makes use of 60GHz radio band in addition to 5.9GHz used in 802.11p and LDPC channel coding instead of BCC in 802.11p. 802.11bd also implements congestion dependent re-transmissions, countermeasures against doppler effect, varied sub carrier spacing and multiple spatial streams among other enhancements. supported relative speeds for 802.11bd are increased to 500kmph from 252kmph for 802.11p (5)

2.1.2 C-V2X standards

first prominent standard in this class was LTE based C-V2X which was part of 3GPP release 14 in 2017 which makes these standards fairly recent as compared to the IEEE standards. most of the results to characterize and evaluate this class of standards are simulation based. C-V2X was developed by 3GPP making use of already existing cellular infrastructure. since the cellular coverage is not reliable and available at all times and all geographical areas, sidelink communication, which was first introduced in release 12 3GPP, is used to enable direct communication between connected devices(cars and infrastructure). LTE-Uu interface is used for traditional cellular based communication between a transmitter node and the user equipment(CVs in our case). PC5 interface provides sidelink i.e., direct communication between multiple user equipment. PC5 interface for sidelink can be used in and out of coverage, for this purpose sidelink modes 3 and 4 are developed for each of the scenarios. although, it is noticeable that C-V2X shows a deterioration in performance too as the traffic density increases, more significantly for sidelink mode 4, which is the only alternative in absence of cellular coverage (6). the latest evolution in C-V2X is NR V2X which was introduced in 3GPP release 16. NR V2X is aimed to coexist with C-V2X rather than replacing it(5) which removes the requirement for backward compatibility. the enhancements are focused on providing service with varying latency, reliability and throughput for advanced applications. sidelink mode 1 and sidelink mode 2 in NR V2X are developed to provide communication in cellular coverage areas and out of coverage areas respectively, much like

mode 3 and 4 in V2X. NR V2X provides groupcast and unicast communication modes in addition to broadcast in LTE based C-V2X, HARQ based retransmission mechanisms rather than blind retransmissions in LTE based C-V2X and varied sub-carrier spacing among other enhancements to fulfill its target requirements (5).

2.1.3 Performance comparison

(7) provides comparison by modelling physical layer of 802.11p, 802.11bd, LTE-V2X and NR V2X in a MATLAB based simulation framework. packet error rate (PER) (which is defined as ratio of erroneous packets received to all packets received), packet reception ratio(PRR)(which is defined as ratio of packets received to total packets transmitted), net data rates(which is defined as data bits received measured in bits per second (bps)) and packet inter-arrival time(which is defined as time between two successful packet arrival) are compared for these technologies for packet sizes of 100 and 1500 bytes .
PER vs SNR plots in (7) provide reliability based performance comparison, PRR vs distance plots provide range comparison, net data rates provide throughput based performance comparison and packet inter-arrival time is a characteristic valuable for communications with regular updates. for packet size of 100 bytes, for 1/2 QPSK channel modulation, 11bd^{DC}, (which is 11bd with dual carrier modulation (DCM) and range extension mode enabled) outperforms other standards, while, 11p has the worst PER followed by LTE-V2X, 11bd, NR-V2X. for 2/3 64 QAM channel modulation, 11bd provides best PER performance followed by 11p, NR-V2X and then LTE-V2X. similarly, for 1500 bytes packet size, 11bd^{DC} outperforms other standards for 1/2 QPSK and 11bd provides even better performance for 2/3 64QAM, outperforming other standards. (7) PRR comparisons are done with MCS0 and 2/3 64 QAM for both standards. for MCS0 and packet size of 100 bytes, it is concluded that LTE-V2X and NR-V2X provide range twice that of 11p and 11bd respectively. while, for 2/3 64QAM 11bd outperforms every other standard with a marginal difference. for 1500 bytes packet sizes range decreases for every candidate. net data rate comparisons in the same study show NR-V2X outperforms other standards followed by LTE-V2X in terms of throughput for 100 bytes packets while for 1500 bytes packets, 11bd outperforms other standards at shorter distances with deteriorating performance on larger distances. 11bd^{DC} performs slightly better than 11bd in each of the scenarios. packet inter-arrival time for 11bd and 11p is marginally different and shorter than LTE-V2X and NR-V2X for 100 bytes packets for distances < 350m. for 1500 bytes packets, 11bd outperforms others till distance of about 125m as shown in plots in (7).

(8) provides PER, throughput and latency based performance evaluations for 802.11p based ITS-G5 and LTE based C-V2X. the study shows that 802.11p provides similar PER performance for different packet sizes from 200 bytes to 800 bytes while C-V2X performance

for smaller packet sizes is significantly better than larger packet sizes. this observation suggests that performance of C-V2X shows degradation when throughput increases while 802.11p is not affected much. for the biggest packet size considered which is 800 bytes, 802.11p outperforms LTE based C-V2X in terms of PER. Range vs throughput plots in (8) reveal that 802.11p based ITS-G5 provides considerably better throughput than C-V2X for range of up to 300m. the study also presents an evaluation of congestion control by plotting range vs number of connected users /km^2 where range is taken to be the distance with PER of at most 10^{-2}. this evaluation shows that ITS-G5 is able to maintain a stable range as the user density increases, while C-V2X sees a significant degradation in performance. the study also shows that resource access duration time increases for ITS-G5 as the user density increases while C-V2X provides constant access time for variable user density, although, access time for ITS-G5 is considerably lower than C-V2X for up to 3000 users per km^2. overall latency is evaluated as average time to receive a packet correctly in the study and latency comparisons show that latency provided by ITS-G5 is also considerably lower than C-V2X for ranges upto 350m. it is noticeable that ITS-G5 provides significantly better performance in terms of resource access time for low user densities and in terms of overall latency for range of up to 300m.

Summarizing the findings above, IEEE based standards provide better PER, latency and congestion control than C-V2X. inter-arrival time comparison from (7) shows that IEEE based standards are better suited for applications with constant updates. although, same study shows that 5G based NR V2X can provide better throughput performance, but 802.11bd can outperform NR V2X under favourable configurations. this makes IEEE based standards a clear choice for applications requiring low latency, high reliability and high throughput. additionally, IEEE based standards for basic traffic applications are already well established across US and Europe. as 11bd is backward compatible, the established infrastructure which is based on earlier IEEE based standards, can be used for 11bd.

So, this section's conclusion is to adopt physical and data link layer from IEEE based 802.11bd for this dissertation. although, it should be noted these standards are still evolving and one class of standards cannot be concluded to be better than other objectively for future applications. their comparisons are often presented but it is far more likely that both classes of standards will eventually co-exist providing performances depending on the applications.

2.2 Existing Middlewares

traditional database, key value pairs or cloud based deployments are not competent to support real time performance required for V2X applications requiring communication of data streams across various components (9). several modern message queuing systems have capabilities of permanent/temporary data storage while providing real time transfer of data streams. for this dissertation, publish subscribe data systems are of particular interest as they can make the received data available to various entities in parallel while delivering a high performance in terms of latency and throughput. In this section, first an overview of a typical publish subscribe system is presented, followed by overviews of middleware systems of interest and than finally a comparitive evaluation of performance for the mentioned middlewares is presented to choose the best suited candidate for specific purposes.

2.2.1 An overview of publish subscribe

publish subscribe (pub/sub) systems fundamentally provide producer consumer decoupling and routing logic. producer and consumer are two components common to all pub/sub systems and most of these systems will have a broker that manages the data among different topics/ classes of data, provides error correction and reliability mechanisms, and interact with producers and consumers. routing logic in a pub sub system can be **topic-based**, which involves data streams produced and consumed on a specific topic which is used to identify a service or data type that can be subscribed by consumers and producers to fetch and push data respectively, or can be **content-based**, which involves tagging data with specific keywords similar to topics. decoupling of producer and subscribers is arguably the most fundamental feature of any pub/sub system, that can be decomposed among following three dimensions(1):

- **entity decoupling**, which means producers and consumers need not be aware of each other.
- **time decoupling**, which means producers and consumers do not have to be producing and consuming at the same time.
- **synchronization decoupling**, interactions between pub/sub infrastructure and producers are not blocked by interactions between consumers and pub/sub infrastructure and vice versa.

In addition to these, different pub/sub systems provide varying level of quality of service (QoS) guarantees, following points cover different QoS guarantees that can be expected(1) :

- **Correctness**, in terms of delivery (at least once, at most once, exactly once) and

ordering (no, partial, global ordering) guarantees.

- **Availability**, in terms of characteristics and features to maximize uptime.
- **Transactions**, describing the units of message groups/sequence that the communication takes place.
- **Scalability**, in terms of characteristics and features that enable support to scale the system or extend service to ever increasing users.
- **Efficiency**, in terms of performance metrics describing the practical service implications of the system's deployment. among the most common of such metrics are latency and throughput.

2.2.2 Apache Kafka

Apache Kafka is a widely adopted and highly scalable distributed event streaming platform. Kafka's design has been driven to fulfill high throughput requirements making it an interesting candidate for this section. main components in Kafka's overall architecture are producer, consumer, broker and zookeeper. Kafka employs distributed designs at each of these components as the processing at each of the components is designed to work, distributed on several machines. Figure 2.1, originally presented in (1), shows overall architecture of Apache Kafka.

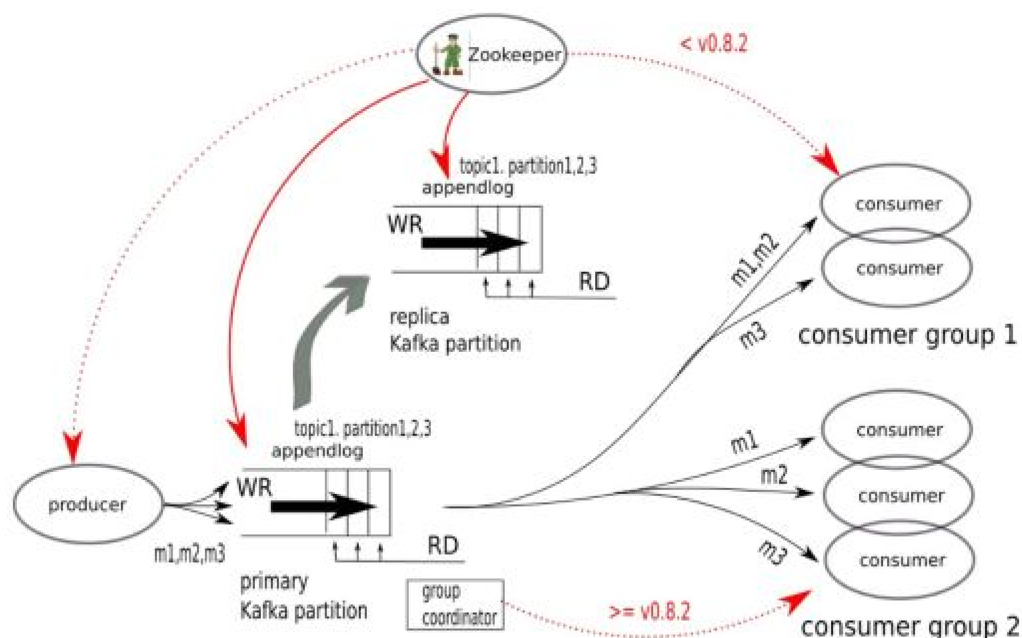

Figure 2.1: Kafka Architecture (1)

In Kafka, the messages published by producers are stored in broker. Kafka provides persistent storage which can be configured to flush old data based on a time limit or a

storage limit. several distributed brokers follow peer to peer broker architecture enabling the availability of all the brokers at all times. zookeeper in Kafka was originally used to manage and coordinate Kafka's brokers and other system components and playing the part of handler, but as Kafka has evolved, involvement of zookeeper has diminished and developer's at Kafka have made clear that zookeeper will eventually be deprecated and control will be given to broker, producers and consumers. starting with v2.8.0, Kafka can be run without zookeeper for basic produce and consume use cases, although the transition is not complete yet and full transition will see zookeeper's exemption from the system. publishers (producers) use push mode to publish, which means data can be published to broker as it is produced by sensor devices. consumers use pull mode, which means consumers can pull data in batches as per the processing capabilities of consumer devices. several consumers can consume data in consumer groups on a topic enabling consumer group to consume all data while each message is only consumed once in a consumer group which means different consumer groups can consume same message but two consumers in the same group can not. Kafka employs partitioning in topics, enabling data in same topic to be stored in different devices, each partition is sequential, immutable message queue that can be added to continuously (9). each topic can be configured individually to have number of partitions and their redundant copies as per requirement. each partition has a sequence number called offset assigned to each message in the queue, which increases linearly. consumers make use of the offset to consume messages in a serial order, and can use it to re-read a message. In a consumer group, each partition can only be consumed by an individual consumer. messages are guaranteed to be in order for each partition. at least once guarantee is default configuration, at most once guarantee can be implemented by configuring producers and exactly once guarantee is also achievable, although comparatively complex to implement (9). to improve throughput, Kafka uses batch processing at all stages(producer, broker, consumers) (1). Kafka is an open source system with a very active user community, new features are ever evolving and Kafka's use in V2X has seen an increasing interest in recent years due to its capability to provide high throughput and low latency service which is useful in range of applications.

2.2.3 RabbitMQ

RabbitMQ is an open source middleware, known to be the best implementation of AMQP protocol. AMQP protocol was first developed to fulfill performance and reliability requirements of finance sector. it is developed with Erlang programming language which inherently supports distributed systems, so, managing systems like zookeeper are not required. RabbitMQ is a lieghtwieght message queuing system in comparison to Kafka in terms of resource requirements. to achieve high availability, RabbitMQ relies on queue mirroring, with one master queue and its mirrors. all the changes are applied to master first

and are then propagated to the mirrored queues (9). the mirrored queues and the master queue essentially contain the same data and are stored in different brokers. Figure 2.2, originally presented in (1), sums up the overall architecture of RabbitMQ's AMQP implementation.

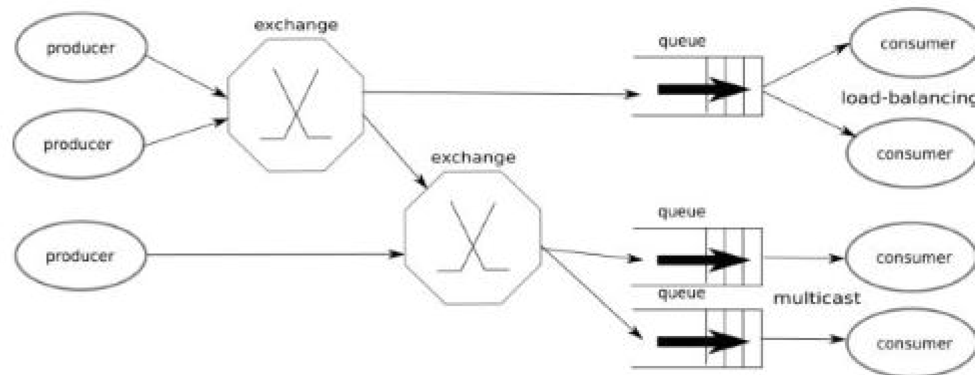

Figure 2.2: RabbitMQ(AMQP) Architecture (1)

exchanges(based on AMQP) are used to route the data received in broker to the queues based on the bindings on queues, which define the criteria associated with the messages in a queue. storage for messages can be configured to be either in memory or in disk. RabbitMQ provides at least once and at most once guarantees. ordering of messages in RabbitMQ is not guaranteed. RabbitMQ is optimized for nearly empty queues and performance degrades with accumulation of messages in queues(1). also, RabbitMQ offers poor scalability as it relies on having redundant data with exact copies of queues in different brokers. RabbitMQ, although, does provide flexibility in terms of its configuration options that Kafka does not, for instance topics can be dynamically created from a producer, which can not be done on Kafka. RabbitMQ also provides a webUI for the system which can be helpful in debugging and monitoring the server(broker) status.

2.2.4 Others

several other middlewares pose viable choices to consider but they either don't offer any significant performance improvements over the aforementioned candidates, or have some limiting factor, due to which they were not considered for this study.
ZeroMQ is a system that is used in high throughput applications. it has a brokerless architecture that supports unlimited queues and batching in messages. ZeroMQ can outperform RabbitMQ at processing message queues containing high amount of data. the challenge however is the technical complexity it demands to develop on and maintain. another limitation of ZeroMQ is that it does not provide persistent storage(10).
ActiveMQ supports a variety of protocols including XMPP and AMQP and provides publish/subscribe communication model using concept of topics and point to point communication model using concept of queues. while ordering guarantee and at least once

guarantee is supported for concept of queues (or point to point model), no guarantees for concept of topics (or publish/subscribe model) are supported. also, ActiveMQ does not support sending messages in batches which poses a limitation when data streams are to be communicated. ActiveMQ has also seen decline in its user community, as user feedbacks and maintenance records have declined over the years(9).

2.2.5 Comparative evaluation

(9) presents throughput(in MBps) and latency comparisons between various messaging middlewares including Kafka and RabbitMQ. other considered systems are RocketMQ, ActiveMQ and Pulsar. evaluations show Kafka provides significantly larger throughput than all the other systems considered when consumer, producer, number of partitions, number of topics are all set to 1. for both Kafka and RabbitMQ, throughput increases as the message size increases, however Kafka delivers almost 10x or more than 10x throughput than of any system considered for messages of size 100bytes, although, this difference between performance decreases as the size increases. for size of 4kBs, Kafka outperforms every other system with it's throughput being more than 2x of any other system considered followed by RabbitMQ's throughput. the study shows that increase in number of producers and consumers cause increase in throughput of RabbitMQ, however a decrease in Kafka's throughput, but the decrease is attributed to the fact that these consumers are consuming in a consumer group and only one consumer can consume for one partition, while, increase in RabbitMQ's performance is attributed to high concurrency support that Erlang language provides. another evaluation is done by increasing number of partitions for Kafka, which reveals throughput of Kafka increases with increase of partitions up to a certain number and than sees a decline. authors in the study attribute this decline in performance to the zookeeper's capacity to handle number of partitions. latency comparisons in the same study show that Kafka and RabbitMQ provide the worst latency performance among the considered systems. but as the throughput provided by systems other than Kafka and RabbitMQ is not nearly enough to accommodate high throughput advanced V2X requirements, they have been taken out of discussion. among Kafka and RabbitMQ, Kafka outperforms RabbitMQ for varying sizes of messages, although both the systems see increase in latency as the message size increases. the study also shows that Kafka provides lower latency than RabbitMQ for increasing number of producers and consumers up to a certain threshold, after which RabbitMQ is able to provide lower latency than Kafka. and for increasing partitions, similar to throughput evaluation, Kafka sees better performance when number of partitions increase, up to a certain threshold and then decline in performance is observed.

(1) presents latency and throughput(in MBps and pps) comparisons for Kafka and

RabbitMQ. it should be noted that for both Kafka and RabbitMQ, this comparison used testing framework provided by their respective distributions and versions used are Kafka v 0.10.0.1 and RabbitMQ v 3.5.3, latest versions available as of July 2021 are Kafka v 2.8.0 and RabbitMQ v 3.9. latency evaluation for at most once mode in the study present mean and max latency for RabbitMQ and 50 percentile and 99.9 percentile latency for Kafka. the study presents that mean latency of RabbitMQ, with or without replication is between 1 to 4 msecs and 50 percentile latency for Kafka, with or without replication is 1 msecs. max latency of RabbbitMQ for the same setting, with or without replication is 2-17 msecs and Kafka's 99.9 percentile latency is 15 msecs without replication and 30 msecs with replication. the study states that for at least once mode, latency of RabbitMQ is not much affected, although Kafka's latency increases with increase in replication. throughput comparisons for at least one mode show Kafka and RabbitMQ both provide similar throughput for up to message sizes of 2KBs with throughput in MBps uniformly increasing for increasing message sizes for both candidates. throughput in pps is quite stable with slight decline,(between 30pps to 20pps for message size between 500 to 2000 bytes) for both systems. study also reveals throughput trends of Kafka similar to (9) for increasing partition count and for topic count, where increase in throughput performance is seen up to a certain threshold. although it should be noted that earlier versions of Kafka were not optimized for high number of topics and newer versions are expected to deliever better performance in that regard. for at least once mode, study concludes RabbitMQ's performance is expected to drop by 50% and Kafka's performance is expected to drop between 50% to 75% compared to the best efforts.

the comparisons above reveal that Kafka and RabbitMQ are both suitable for high throughput and low latency requirements of V2X. although, Kafka will be able to provide a significantly better performance in a relatively rigid setting in terms of number of producers/consumers/number of topics/number of partitions while RabbitMQ can be more flexible and provide similar performances as it is scaled, although RabbitMQ does not provide exceptionally high scalability which Kafka seems to deliver. this leads to the conclusion that while Kafka can handle high throughput and low latency under rigid settings, RabbitMQ can deliver high reliability under flexible scenarios where high scalability is not needed.

2.3 IOT Protocols

last section gave a look at available middlewares that offer usefulness to this dissertation. Kafka provides unmatched performance in regards with fulfilling high throughput and low latency requirements while being highly scalable and being able to maintain high level of reliability in rigid settings. these characteristics make it a perfect choice to form the backbone of a V2X infrastructure. however, a flexible collection mechanism is required that

could handle collection of data from CVs and deliver it to Kafka endpoints and vice-versa reliably. RabbitMQ with AMQP seems like a good candidate but a look at other available options is needed to make the final choice. even within RabbitMQ, a choice to be made as it supports several messaging protocols on application layer including AMQP(by default), MQTT and HTTP+WebSockets. several IOT protocols have emerged over the years on application layer. so, to arrive at the best choice of protocol for this use case, first an overview of some prevalent protocols and then their comparison is presented in this section.

2.3.1 An overview of prevalent IOT protocols

In this subsection, some prevalent protocols, namely HTTP, CoAP, MQTT, AMQP, DDS and XMPP are briefly reviewed.

HTTP

is the application layer protocol used for data communication, most commonly, on world wide web. HTTP is a request response protocol and uses request response acknowledge cycles for reliability. as HTTP communicates messages in request response cycles, it by default doesn't seem to be a good choice to communicate data streams. additionally, a common request header in a HTTP message is about 800 bytes.

CoAP

CoAP is an application layer protocol that closely resembles HTTP but is intended to be used on resource constrained devices. CoAP provides CRUD operations through HTTP methods and provides status codes very similar to HTTP(11). to provide reliability, CoAP uses confirmable messages(CON) and acknowledgement.

MQTT

MQTT is a publish/subscribe protocol and is consider lighter than HTTP 1.1 which supports near real time message exchange in IOT devices. MQTT supports at most once, at least once and exactly once message delivery guarantees. MQTT is a classic publish subscribe with a broker that manages topics, which producers and consumers can subscribe to in order to publish and consume messages. MQTT is often used in scenarios requiring intermittent connectivity, which will be the case for CVs connecting to road side stations to communicate. so, MQTT poses as a candidate of interest.

AMQP

AMQP is a publish/subscribe protocol generally used in corporate environments. three main components of AMQP broker are exchange, binding and queue. exchange is responsible to route messages to queues based on the bindings. it can do so in four ways, i.e., direct when binding key on message queue exactly matches publisher's routing key, topic when a binding key pattern is used to be matched against routing key, fanout when message needs to be broadcasted to all subscribers unconditionally, and using x-match-expression in headers which supports logical "AND" and "OR" matching. AMQP supports at least once, at most once and exactly once message guarantees.

DDS

DDS is a brokerless publish/subscribe model that follows a data-centric approach for data sharing. messaging model is split in two layers, a) data-centric, publish-subscribe (DCPS) and b) Data Local Reconstruction layer (DLRL). DCPS is responsible for associating data objects for publishers and subscribers for values that need publishing and subscribing, respectively. The DLRL, an optional layer in DDS, acts as a connector for integrating DDS at the application level for interfacing with other external entities (e.g. other protocols, applications, among others(11).

XMPP

XMPP is an open standard protocol used for real time applications like instant messaging, voice and video calls etc. XMPP uses XML as data format. but XMPP is based on fire and forget, and so does not provide high reliability within it's core services. although, XMPP is a highly extensible protocol and several external plugins and features can be explored to fulfil the requirement for reliability, performance trade-offs can be expected. so, XMPP is ruled out for this dissertation due to the nature of the use case requiring high reliability while being able to maintain high throughput and low latency.

2.3.2 Comparison

(12) presents response time and throughput comparisons between HTTP and MQTT. response time comparisons show that response times for HTTP are 12.7 slower than MQTT when used in fog architecture and 4.7 and 2.5 times slower than MQTT when used in cloud, with cloud located in the same country and with cloud located in different country respectively. the throughput comparison in the same study shows that HTTP is able to provide more throughput for low traffic, but HTTP saturates after a certain point while MQTT's throughput keeps increasing with frequency of messages.

MQTT and CoAP are compared in (13) using a middleware with common interface. it shows that with increasing packet loss, MQTT can deliver packets with smaller delays than CoAP up to a certain threshold, after which CoAP is able to provide better delays which is attributed to TCD overheads in MQTT(vs CoAP using UDP in transport layer). the study also shows that with increasing packet loss, MQTT still manages to deliver more data than CoAP.

(14) compared CoAP, AMQP, DDS and MQTT. the latency was compared for transfer of 10 and 1000 messages of sizes 5KBs, 20KBs and 50KBs. the results in the study show that CoAP performs best for 10 messages of size 5KBs, followed by MQTT, then AMQP and then DDS, while, for 1000 messages of same size, DDS performs best followed by CoAP, AMQP and MQTT. for 10 messages of 20KBs, CoAP again showed best performance followed by AMQP, MQTT and then DDS while, the order remained same for 1000 messages as it was in 1000 messages of 5KBs. the order for 1000 messages of 50KBs was again same as that with 1000 messages of 5KBs and 20KBs, while, for 10 messages of 50KBs, DDS performed better than AMQP and MQTT, CoAP performed best and MQTT performed worst. it is noticeable that MQTT always performed the worst with increased number of messages.

(15) conducted a throughput based comparison for HTTP, MQTT and CoAP by recording number of messages transferred using same settings. the study shows that MQTT delivers best throughput by delivering 263314 messages, followed by CoAP which delivered 134235 messages and then HTTP which delivered 3628 messages in the same amount of time. it is noticeable that the difference between throughput is significant.

(16) compared HTTP and AMQP by sending messages and storing them, in web server service (with database using JDCB drive with HTTP), and in RabbitMQ (with AMQP). the study shows that web server service was able to send and store messages at the rate of 125.9 messages per second, while, in case of RabbitMQ using AMQP, 226.6 messages per second were sent and stored. the study demonstrates AMQP is able to provide better throughput than HTTP.

(17) compared the performances of different brokers of MQTT. the study measures the time taken to send 1000 messages with QoS 0, 1 and 2, for all the brokers considered in study using a single client subscribed to 1 topic. it can be seen that for QoS 1 and 2, time taken to deliver a single message is approximately 1 second. although, exact size of these messages is not given, it is specified that the messages are temperature and humidity alerts.

it is noticeable in above comparisons that other candidates have outperform HTTP every time. studies (14) and (17) have indicated that MQTT's response times tend to be higher than most of the other candidates when number of messages being transferred is high. CoAP is able to deliver a good performance at low message frequencies but the throughput performance declines with increasing load as indicated in (13). now, the choice is to be made between DDS and AMQP. DDS outperforms AMQP in some scenarios and both candidates seem to be a good choice for bridging the gap between Kafka and CVs. however, DDS has a complex resource intensive implementation with large sized libraries (some more than 2GigaBytes)(11). On the other hand, RabbitMQ with AMQP provides a lightweight and distributive deployment that can prove useful at the edge of the infrastructure. so, to conclude this section, AMQP is the best alternative to go ahead with.

2.4 Similar Works

this section shines light on similar works that have been done in recent years to develop an infrastructure for advanced V2X applications. taking a look at similar works gives a perspective on recent developments in field, utilize any insights on what to avoid and what to explore further and base the scope of work done in this dissertation on fields unexplored.

(18) proposes a V2X infrastructure to be used with central LDM. central LDM is a concept proposed by ETSI aiming to process data between ITS stations (which includes vehicles, roadside infrastructure and mobile devices) and C-ITS environment in real time. LDM defines dynamic information such as road conditions, weather and speed information, CVs on the road as LDM objects. LDM receives data from other external affiliated organizations in addition to ITS stations to provide information (like weather conditions from weather stations). the existing central LDM however, cannot make use of all the data available from moving vehicles as it primarily relies on roadside infrastructure to collect road environment and is configured to collect simple driving information like speed and position from the vehicles on the road. the project proposes that availability to data from vehicle sensors can be utilized by adding support to collect and process this available data to existing infrastructure. the proposed addition to infrastructure is aimed at collecting sensor data from vehicles, provide analysis results from all the data available, including the data from existing C-ITS servers (traffic server etc.) to CVs to enable applications like co-operative autonomous driving. LDM will be updated with the analysis results to provide support to and still utilize existing infrastructure. the proposed architecture mainly consists of a collection system with Kafka at its core, a data store made up of MariaDB and HDFS, and a processing system that uses Apache Spark and Zepllin. the proposed architecture organizes data collected from sensors on vehicles in different data formats, processes raw data from

cameras and LiDAR data sends processed vision data(detected objects etc.) to Kafka, while, GPS and IMU data is collected directly by Kafka and delivered to MariaDB and HDFS, from where Apache spark and Apache Zepllin process the data in HDFS and MariaDB to return analysis back to the CVs. raw data from camera and LiDAR is also sent directly to HDFS. the project goes on to implement the design using a server and a test vehicles with GPS, IMU and camera sensors equipped. autonomous car was used to collect data in ROSBAG(record file of ROS), which was later used to draw results on the design. the project provides latency evaluation, from the time the data is generated by the sensor(simulated by replaying ROSBAG files), to the time it is received in database. the setup provided average latency of 14msec to transfer generated data to a python application through Kafka. it was than converted to different data format by python application and transported to database by employing Kafka connectors which again took 14msec. average latency was hence concluded to be 28msec. minimum and maximum latency were recorded to be 14msec and 213msec respectively. the case of maximum latency occurred when data was first transferred from python application to database as that's the first time connector between Kafka and database was connected. the results are concluded with a guarantees of average latency of 14msec and maximum latency of 200msec after the connection between Kafka and database is connected.

(2) proposes a distributed message delivery infrastructure. the study predicts that CV system is likely to become a part of bigger connected society where city wide disaster management systems, emergency services and smart healthcare will all be connected as described in vision of smart networked systems(19). the authors in the study state that, primary challenge for a CV message delivery system are redistribution of data at different stages of data lifecycle(raw, integrated and processed) while meeting functional requirements in time and spatial contexts. the study proposes distributed delivery of streaming messages by utilizing Kafka as the delivery system. the study bases the topics for data streams on CVRIA(20), developed by USDOT which defined concept for more than 90 CV applications when the study was wrote. the study used VISSIM based microscopic traffic simulation to generate the synthetic data of 62 different types. the study simulated 91.5 miles of roadway network along a freeway scenario. 92 RSUs were simulated with DSRC communication range of 900ft, which means each RSU collected data for 1800 ft. In the simulated connected system, each CV sent data to RSU which was then sent to backend transportation centers. the hardware used for Kafka brokers were 16 machines, each with 256GB DDR4 RAM, 2TB 7,200 RPM SATA HDD hard disks and a 10GBps ethernet connection. the hardware used for consumers and producers(CVs and RSUs in different experimental scenarios that study conducted) were machines with 16GB DDR4 RAM and 100GB 7,200 RPM hard drive and 1GBps Ethernet connection. in each experimental scenario, every producer or consumer had a dedicated machine in all the scenarios. the study conducted a baseline experiment using 92 producers

and 10 consumers(without kafka brokers) and 3 scenarios with 92 producers and varying number of consumers with varying number of brokers. size of messages sent were 200 bytes and number of messages sent to arrive at average latency values was 50,000 in baseline, and 50,000 or more in other 3 scenarios. total latency was calculated as time between producer sending the message and consumer receiving the message. study shows that baseline delivered average and maximum latency of 12msec and 3751msec respectively, while when 2 brokers were employed with same number of producers and consumers, average and maximum latency dropped to 2.03msec and 1463msec respectively. average latency increased to 7.95msec when 50 consumers were used with 92 producers and 2 brokers. in another scenario with 4 brokers and 92 producers, average latency was observed to be 1.67msec for 10 consumers and 6.18msec for 50 consumers. and finally with 16 brokers and 92 producers, average latency for 10 consumers and 50 consumers was observed to be 1.49msec and 3.71msec. the study also shows that throughput increased as the number of brokers employed was increased, while with higher number of consumers, throughput for both a single producer and a single consumer involved decreased compared to when lower number of consumers were used. the study also shows that for a setup with 16 brokers and 92 producers and consumers varying from 10 to 50, average latency and 99th percentile latency differed very slightly(99th percentile latency $<$ 13msecs), while maximum latency could even reach upwards of 400msecs in some cases.

(21) aims to accommodate every vehicle in the road to be able to connect to C-ITS, in case vehicles don't have hardware capability to connect, the authors propose use of passenger's mobile device. to provide service at such a scale, the study proposes scalable infrastructure by using Kafka at core, and MQTT to bridge the gap between Kafka and CVs. as Kafka is not designed to support such a high number of connections, it is proposed that EMQX broker using MQTT is the solution, which can scale to millions of connections. the study uses Kafka's Connect API, by developing sink and source MQTT connectors to Kafka. source connector is proposed to receive messages from CVs and publish them to Kafka for data distribution. wildcard MQTT topics are proposed to provide service to CVs which Kafka is not designed to do. the authors propose unique connector for each message type which will publish to static topic on Kafka corresponding to that message type. MQTT sink connector will create dynamic topics to send messages to CVs, these dynamic topics are contained within messages which follow JSON format. results for a feasibility test of the design are provided. test was done on a regular laptop with 16GB of RAM and a 4 core CPU. Kafka was set up with a single broker, single zookeeper and a Kafka Connect cluster. CAM messages of 2KBs were used to be sinked and sourced from Kafka. the study presents that all messages were delivered from source connector to Kafka, which proves the feasibility of sourcing multiple MQTT topics to a static Kafka topic and all messages from Kafka were delivered to MQTT sink connector, which proves the feasibility of sinking messages from a

static Kafka topic to dynamic MQTT topics. the study also presents latency results by recording timestamps at different phases.
The studies presented above all use Kafka at core of their data collection and delivery mechanisms. this proves the usefulness and feasibility of Kafka as a delivery system. the studies have explored the overall architectural designs of possible solutions to advanced V2X challenges. although, the CVs on the performance in IEEE based standards is also significantly affected by the communication between RSUs and CVs. the CVs are moving at relatively fast speed with intermittent connections to RSUs. so, that is one area this dissertation will aim to add to while designing an overall infrastructure of message delivery system that can be basis of a V2X middleware.

3 Design

This chapter presents design for a V2X middleware that could potentially fulfill advanced V2X requirements. it starts out by highlighting some design considerations in context of components and characteristic of a typical V2X infrastructure and how it influences the design choices. then, Section 3.2 presents early attempts of the design with key shortcomings to highlight findings that lead to final design, which is presented next. section 3.3 presents overall architecture of the proposed design, followed by section 3.4 that explores inner components and processes involved in design by going through lifecycle phases involved in expected communication. section 3.5 is summarizes measures used to enhance performance in the design. section 3.6 shines some light on extensibility of the design and how possible extensions can be made to design for future works

3.1 Design considerations

this section reflects at choices made in last chapter, their implications and points out considerations in design that motivate the final design.

3.1.1 Basing on IEEE standards

In previous chapter, the choice was made to build infrastructure up on IEEE based standards as they show capability to provide superior overall performances compared with C-V2X. IEEE based standards rely solely on RSUs to collect data from CVs on the road, two main implications of which are:

- intermittent connectivity between RSUs and CVs poses a challenge if it is to be utilized as an edge node. switching between RSUs have to be handled at application layer when a continuous stream of data is being communicated and re-transmission mechanisms have to be employed at application layer as packet loss can occur when switching from one RSU to next.
- collection of data by RSUs can bring the system to the edge. this opens up scope for pre-processing at the edge by utilizing processing capabilities of RSUs.

3.1.2 Kafka as core of delivery system

As discussed so far, Kafka provides high throughput and low latency capabilities which has made it an ideal choice for basing a message delivery system up on. some implications of choosing Kafka that need consideration in the design are as follows:

- as discussed earlier, Kafka is not designed to handle high number of topics/producers/consumers. if number of brokers is increased, Kafka can be scaled to handle high number of consumers and producers. although, increasing number of topics will still be complex as topics are stored in partitions distributed between all brokers and this distribution can not be configured manually.
- Kafka does not provide support to create dynamic topics as per producer's demands. static topics in Kafka will have to be designed by taking that into considerations.

3.1.3 RabbitMQ as bridge between Kafka and CVs

In previous chapter, it was shown that RabbitMQ with AMQP poses an ideal fit to communicate data from CVs to Kafka endpoints. below are some considerations relating to RabbitMQ that are noteworthy in that context:

- RabbitMQ's ability to create dynamic queues and handle high number of connections is one of the reason that makes it useful in this design. consideration has to be taken to employ proper cleaning mechanisms for dynamic queues created by CVs that are not needed anymore.
- RabbitMQ provides basic inbuilt acknowledgement mechanisms handled by the midddleware. however, for the purposes of this thesis, manual handling of acknowledgement and re-transmissions is of interest considering the challenges introduced by nature of intermittent connection between RSUs and CVs.

3.1.4 Programming language

the implementation of the proposed design chooses Java to be the development language, after considering following points:

- JAVA client APIs are officially maintained as part of Apache Kafka project and can be imported as a dependency in a maven project.
- RabbitMQ also officially maintains JAVA client APIs and provides implementation options as a maven dependency.
- JAVA has range of libraries and tools available to extend functional support which could prove very useful in implementing the design

it should be noted that considerations for programming language here relate to the choice of language used in implementation of the design as part of this thesis. In a real world deployment, there could be issues that might influence choice of the language used. for example, there could be inter-operablity issues within programming already being used in vehicles or RSUs. language used for a middleware must keep such factors in consideration and possibly extend support to multiple languages.

3.1.5 Other considerations

providing scalability, fault tolerance, reliability and extensibility have been major considerations throughout the design. the design is focused on core functions of a delivery message system and thus extensibility is focused on, as to leave scope for secondary features. extensibility is of importance, also because an infrastructure providing support for V2X applications will greatly enhance its resourcefulness if it can be extended to provide continued support as new enhancements surface.

3.2 Initial attempts

Initial design attempts primarily focused on utilizing Kafka at its core and bringing a distributed message collection and delivery system to the edge nodes. the approach was focused on utilizing the distributive properties and high scalability of Kafka to fulfil advanced V2X requirements.
Figure 3.1 shows the overall architecture that initial designs were based on. the design proposed installing Kafka brokers at RSUs, in addition to a backend Kafka cluster for collection and delivery of data between the communication system and CVs. CVs would produce and consume data on Kafka brokers in RSUs, and external public and management services would use the backend cluster and act as producers/consumers to transfer data. the main advantage of such design would be to have scope for pre-processing in RSUs and potentially provide low latency performance, as CVs acting as producers and consumers can directly publish/consume data from message delivery system. the increased number of Kafka brokers will be able to provide high performance even with increased number of connections with CVs. but this design introduces several issues, key issues are summarized in following points:

- Kafka stores partitions of topics in different brokers and each topic has a partition leader, which is responsible for the selection of partition that the data gets added to. distributing brokers in the manner proposed in this initial design does not guarantee that CVs produce data to partitions in the nearest broker. every message produced on a topic is routed to topic leader first, which routes the message to one of the partitions. the likely scenario is the data being produced for different topics will be

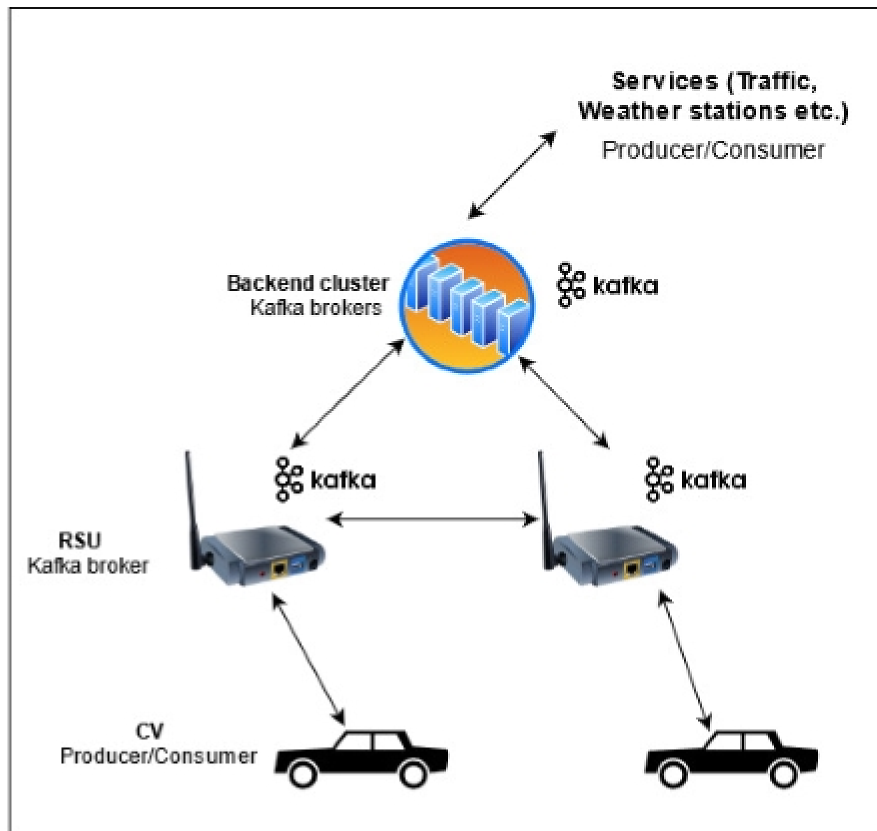

Figure 3.1: Overall architecture of initial designs

transferred to different brokers in the cluster once it is received by the nearest broker and then is transferred to topic's leader partition. this will introduce high traffic in the network, and brokers responsible for collection potentially will not be able to deliver the performance that was expected.

- Creating dynamic topics in this scenario wouldn't have been ideal as the design brings core messaging system to the edge. moreover, it has already been highlighted that Kafka does not support dynamic topics. while CVs could publish data on the brokers and it could have been utilized in a meaningful way, targetting data for consumption of a unique vehicle on the road would have been challenging in this set up and only location specific data would have been delivered efficiently. processing required for delivering data specific to one vehicle would have possibly outweighed latency gains achieved by this design.
- another shortcoming of this design is that Kafka is resource intensive in terms of processing power required and stores data persistently. although there are workarounds to avoid storing data persistently on RSUs, the occupation of processing power would still have posed a key issue.

the issues above could be avoided by separating Kafka based message delivery system on the backend, and collection system formed by brokers in the RSUs. these initial attempts

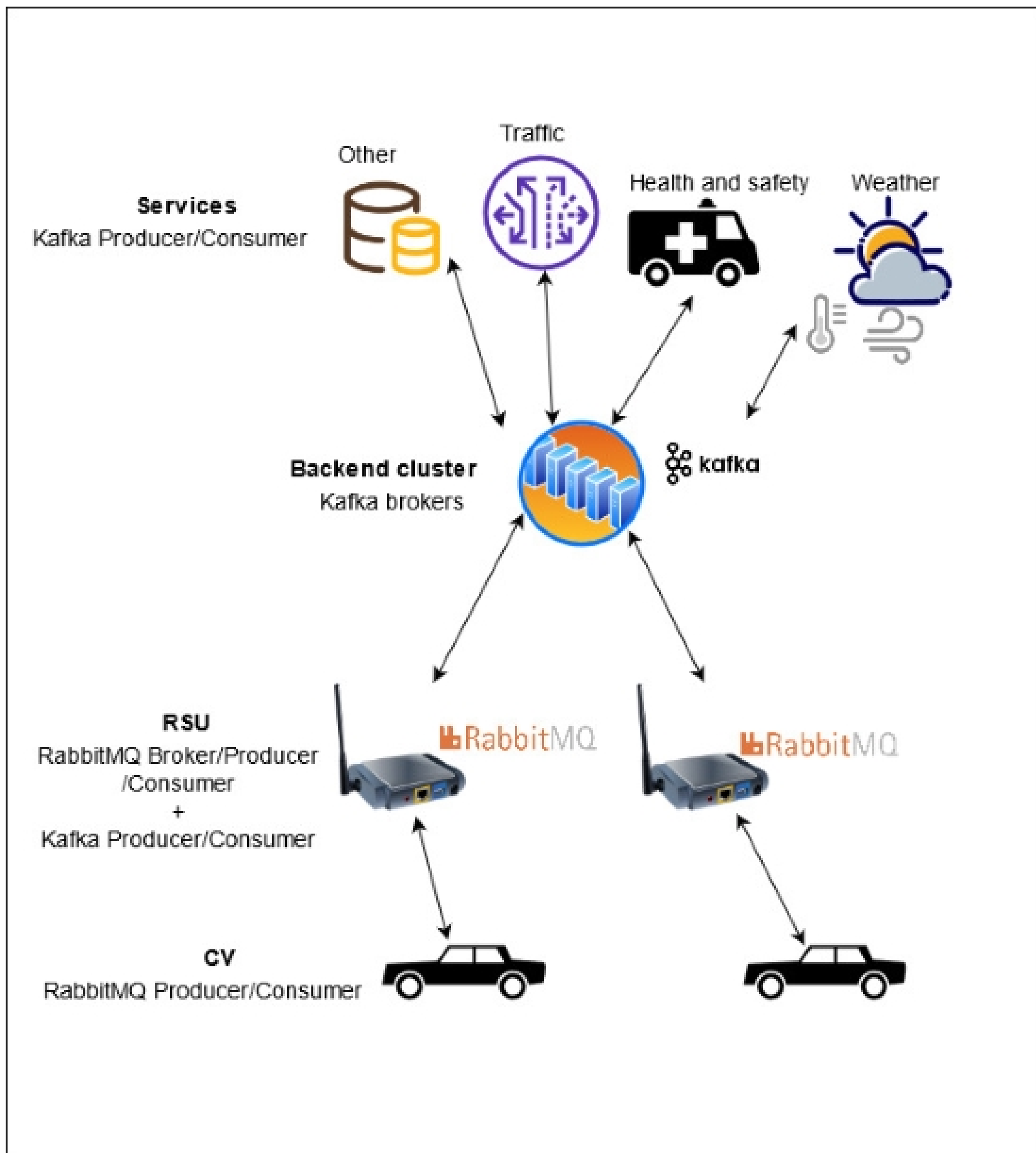

Figure 3.2: Overall Architecture

motivated study for a better suited collection system, which was later decided to be RabbitMQ. this section was worth mentioning as it provides perspective as to why a separate collection system is necessary, especially, if scope for pre-processing in RSUs is to be reserved.

3.3 Overall architecture

This section presents overall architecture of the final design proposed. the design essentially replaces Kafka brokers on RSUs with RabbitMQ brokers to bridge core Kafka based messaging system with CVs.

Figure 3.2 presents the final design proposed. Kafka based messaging system forms the backbone of the proposed infrastructure. traffic, health and safety, weather and other services act as producers and consumers to receive and send notifications and data streams. Kafka's Connect API can also be considered for telemetry of data, and processing functionalities core to the infrastructure but is not explored in this thesis as it's capabilities lie in processing data in bulk. the core Kafka based messaging system is essentially fine tuned implementation of Kafka, and services are essentially Kafka producers and consumers. middleware support is provided as software modules for RSU and CV. Each RSU has a RabbitMQ broker installed, acts as RabbitMQ producer and consumer to communicate data with CVs, and Kafka producer and consumer to send to and receive data from core Kafka based messaging system. CV acts as RabbitMQ producer and consumer to send and receive data. the system aims to collect streams of data generated by IOT sensors, so, middleware support in CVs is designed to re-transmit any packets that are not acknowledged by RabbitMQ broker in RSU. these data streams are stored in a buffer developed in middleware support for RSU. messages from buffer are sent to Kafka by a producer thread, RSU act as producer and consumer for core Kafka based message delivery system. data is consumed from Kafka message delivery system by each RSU independently(every RSU is sole consumer member of its consumer group). the received data is cached in memory by using a buffer mechanism, for which middleware support is provided. this data is sent to CVs as they connect. middleware support uses a locking mechanism on the cached data, taking into consideration that same data may possibly be requested by multiple CVs connected to same RSU. when RSU consumes data streams from Kafka, the data is stored with offset for each message(recall, Kafka stores data in its partition with offset for each message, which consumers use to consume data sequentially and to re-transmit any lost packets), which is same as offset of the message in Kafka partitions. this offset is sent to CVs when they consume data. as they connect to next RSU, the offset is used to continue receiving messages from the data stream. the detailed lifecycle of expected communication, and processes and components involved in it is provided in next section. it should be noted that focus of this dissertation has been on developing a communication system and any processing logic(as in algotithms for processing raw data etc.) is not proposed, but the presented design reserves scope for processing at different stages. middleware support can be extended to include actual processing.

3.4 Lifecycle phases

This section presents typical lifecycle of data stream communication. the design is based on transfer of streams of data between RSUs and CVs to each other, or in other words, data being both produced and consumed by both CV and RSU at high frequencies. this section presents processes involved as a CV connects to a RSU, transfers data, connects to next CV

as connection is lost(or reconnects if packet loss exceed a certain threshold) and continues data transfer.

3.4.1 Overall lifecycle

Figure 3.3 presents the overall lifecycle of communication. once, the connection is established the first step is a handshake protocol on application layer which is initiated to exchange topics that can be produced/consumed between CV and RSU. note that every message sent between RSU and CV uses RabbitMQ with AMQP, the handshake protocol is a high level abstraction to communicate information necessary for declaring relevant queues so as to start producing and consuming on the topic(recall that RabbitMQ uses queues to exchange data on topics, these queues are declared dynamically in different processes throughout the lifecycle. throughout the work so far, it is mentioned that topics in RabbitMQ can be declared dynamically. declaring a queue dynamically and declaring a topic dynamically mean the same in this context). next step is to start actual data transfer between CV and RSU, in parallel with data transfer between RSU and Kafka based core messaging system. once the connection with the current RSU terminates, by moving out of range or due to presence of another RSU closer to CV, the same cycle repeats. subsections below describe each of these phases, namely CV-RSU Handshake, Producing Data, Consuming Data, and Termination of connection and connecting to next RSU . then the next subsections cover how the buffers/caching of data is implemented. choice for format of the messages is presented next and lastly, clean up mechanisms employed throughout the design are overviewed.

3.4.2 CV-RSU Handshake

The application layer handshake is initiated by CV. RSU is always listening for messages to initiate handshake from CVs on a predefined queue(known to CV). CV initiates the communication by sending a message with a unique car Id and name of topics it is producing data on. the topic names sent by car in this message are same as the static topics in core Kafka message delivery system. CV also starts a listener with queue name made by a pre-decided prefix(both RSU and CV know the prefix) with car's unique Id to listen on RSU's response.

On receiveing the message, RSU starts RabbitMQ queues for each of the topic that CV is producing on, and Kafka producer threads for each of the topics. working of these listener threads and Kafka producer threads will be described in next section. these threads are started in parallel by RSU as it responds to CV with message containing static topic names of consumable data topics(as in Kafka based messaging system) that RSU has consumed from Kafka messaging system. RSU also starts dynamic queues for these topics, which will

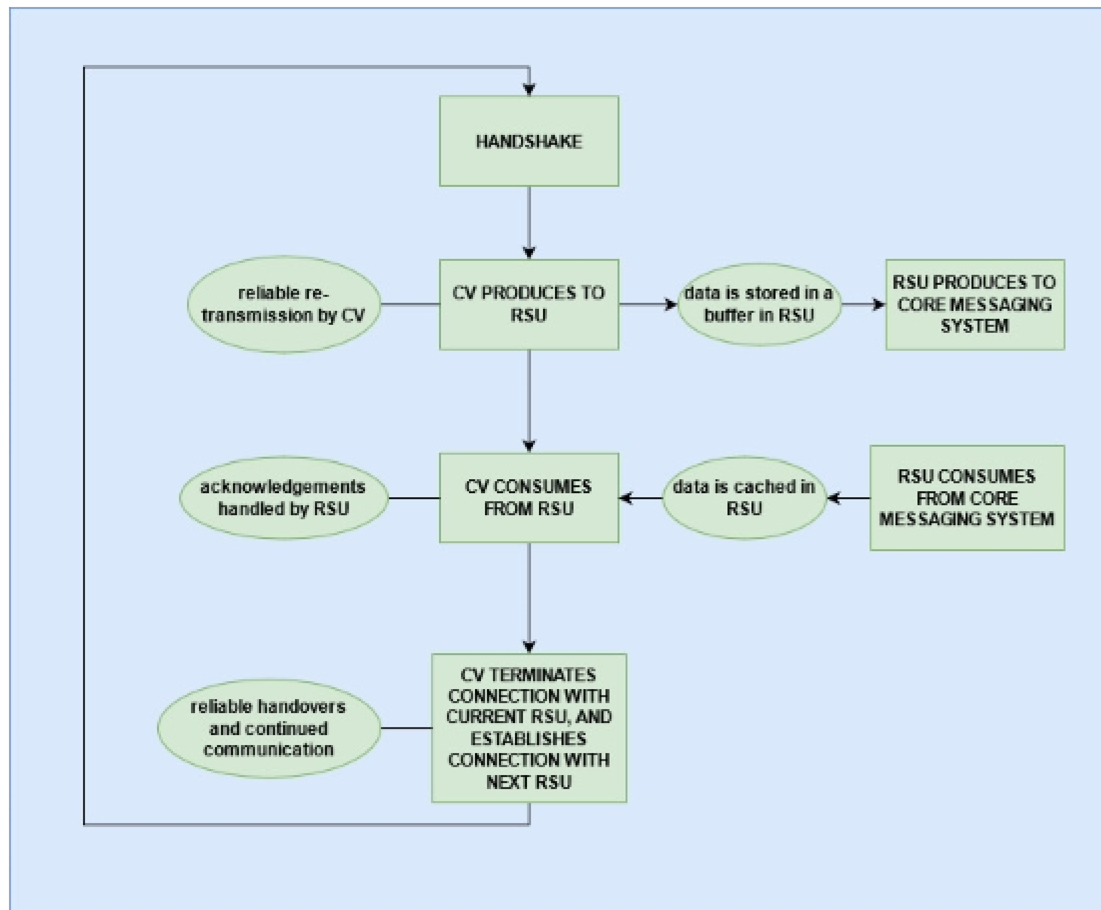

Figure 3.3: Overall lifecycle

be used by RSU to produce messages consumed from Kafka at a later stage after offsets(which are ids to reference messages. the purpose of offsets is described in following sections) are received from CVs, which are required to start producing data to queues which CVs can consume from, sequentially starting from the message at the offset provided.

the response for CV's message to initiate communication is sent by RSU on the listener queue that CV started earlier when it initiated the communication. this message also acts as an acknowledgement for CV to start producing. CV receives the message sent by RSU and starts producing data, and sends offsets to RSU for different topics to start consuming data.

the handshake protocol, basically, involves sending two messages that set up the ground for communication of actual data between RSU and CV. it is proposed that the handshake can be re-tried by CV after a time-out if no response from RSU is received.

3.4.3 Producing data

CV acts as a RabbitMQ producer to send messages to RSU on the queues declared in previous section. CV buffers the data produced by sensors and sends messages to CV from that buffer. messages produced by CV are acknowledged asynchronously by RabbitMQ broker in RSU. CV sends data in batches of messages by fetching all the messages in buffer. a parallel thread is started to clear the messages fetched and store these message in a different buffer for transmitted messages in the order that messages were sent. a parallel thread works to clear acknowledged messages from the buffer of transmitted messages. when a message is acknowledged out of order that the messages were transmitted in, any previous messages that have not been acknowledged are added to the buffer of sensor data that CV is producing messages from. because of this design choice, the middleware support in CV does not provide ordering guarantee for messages produced but it ensures continued stream of messages from CV to RSU, aiming to improve throughput of the system. timestamps are added as a header in the messages produced, which can be utilized during processing of data when ordering of messages is important.

RSU stores all the received messages in buffers, each of these buffers is unique to each CV and reference-able by the static topic names. Kafka producer threads for all the topics which were started by RSU in Handshake phase send data from this buffer to Kafka messaging system. each producer thread fetches all the data from the buffer for a single topic and clears the buffer, the fetched data is then produced on Kafka by RSU and once all the messages are delivered, producer thread, once again repeats the same cycle. it is proposed that if no data is fetched by producer threads from buffer for 3-5 seconds, the buffers can be removed and producer threads can be terminated. the actual timeout limit can, of course, be decided according to the implementation, for instance, time out for highway scenarios will possibly be less than time out for urban or city scenarios as RSUs on highway will deal with CVs with relatively faster speeds and less connection times. Kafka offers two main acknowledgement mechanisms for producers. with acks=1 configuration option(configuration done while producing messages), acknowledgement of messages is received as soon as the messages reach topic leader and with the second option, acks=all, acknowledgement is received for messages as they reach the specific partition that are supposed to be appended to.

3.4.4 Consuming data

RSU acts as Kafka consumer to consume data from core messaging system. the consumed data is stored in a fixed size buffer(/cache). different RSUs consume all the available data independently and the data is updated as it is available to consume from backend Kafka system and older messages are pushed out. the idea is that data consumed is always the latest information and of relevance to connecting CVs, outdated streams of data are not of

relevance. each message in consumed data, stored in HashMap based buffer, is stored with offset as key which can be used to reference the message. these offsets are same as the offsets these messages had in Kafka partitions. Kafka consumers use these offsets to ensure that messages received are sequential from a partition, and to re-transmit any lost messages. the design proposed offers a similar solution where these offsets are used by CVs as they switch between RSUs to keep receiving the messages continually.

CVs consume data from queues declared in handshake phase in subsection 3.4.2. it was mentioned before that RSU waits for CV to send offset before producing on queues. CV sends a message with TYPE header of value INIT to RSU with offset value for the topic in header marked OFFSET on the same queues that RSU started for producing consumable data for each topic in CV-RSU handshake. RSU on receiving offset value, starts producing messages, starting from the message with offset value received. if offset is not recognized, it is proposed that adjacent offsets can be checked and data should be sent starting from most matured message in adjacent offsets. these messages are consumed by CV from the queues for each topic and offset value for each of the topic is updated in CV as the messages are received. here, the choice was made to use the same queue to send a message with offset value from CV to RSU, that is next used by RSU to produce consumable data for CV. this design choice avoids creation of redundant queues for each topic. these redundant queues would have potentially slowed down the system significantly when high number of CVs are connecting.
here acknowledgement for messages received by CV can be done by using offsets, where if a message is received out of order, re-transmission can be attempted but since, communication of messages at high frequency is considered, it is proposed that using RabbitMQ's in built acknowledgement mechanisms could be a good choice here considering offsets in CVs are reserved while switching between RSUs. the final choice, of course, lies with the nature of implementation and deployment.

3.4.5 Termination of connection and connecting to next RSU

Termination of connection between CV and RSUs can happen in two ways. either the vehicle can travel beyond range and connection might break between RSU and CV, or CV might discover another RSU with better signal strength and connect to it. in any of the cases, the messages being produced will stop getting acknowledged (as the queues being used to produce messages were declared by previous RSU), and after a pre-decided limit of failed attempts, CV-RSU handshake will be retried. for consumer threads in CV, which stop receiving messages too in case of connection termination, time out can be used to stop the threads when messages are not being received, but design proposes that a better alternative is to send a signal(simplest implementation to send a signal would be to use a bool variable

to indicate that connection has broken) to stop the threads when CV-RSU handshake is re-tried.

3.4.6 Buffer/Caching mechanisms

At several stages, buffer systems are used to accommodate high frequency of messages being sent and received. the choice enables continued flow of messages and maximize throughput output. the buffers are designed by using JAVA's HashMap implementation. locking mechanisms are implemented on all the buffer systems and functions are designed that can be called to get access to data in buffers. locking is implemented by using boolean variables to provide conditional access when attempting a write on buffer.

3.4.7 Format of the messages

every message being transmitted uses JSON format. messages are serialized as a string while transmitting through messaging systems and desearialized when received if there is a requirement for accessing headers.(note that the term header, which has been used in previous sections and this section, implies key of a key-value pair in the JSON message.) the messages are tagged with a TYPE header. TYPE header has value INIT for a message used in CV-RSU handshake or to send offset value from CV to RSU and value DATA when messages contain actual data. TYPE header is checked when messages are meant to be carrying control information like in the case of handshakes. messages containing data are also tagged with information relating to data in message. for example, topic name and timestamp are contained within the message headers. the data is contained in these messages under key DATA.

3.4.8 Clean up mechanisms

Below points summarize how several clean up mechanisms are employed in the design:

- RabbitMQ provides inbuilt mechanisms to delete unused queues on a time out basis after their purpose is fulfilled. employing these mechanisms ensures a availability of better processing power which plays significance in latency provided by the system.
- Every thread process employed in the design has a break point, either conditional or based on time-out to ensure that every thread that has started reaches an exit. this again, ensures availability of better processing power.
- Buffers employed are also designed keeping into consideration, that they do not store outdated messages. in each case where buffers are employed, they are cleared as they serve their purpose, for instance, buffer employed in CVs to store sensor data deletes the message that is fetched for transmission and a separate buffer for transmitted

messages is maintained to be used in case of re transmission, which is also cleared as the messages are acknowledged. in cases where latency is of essence, processes responsible to delete message from buffer as they are fetched are done in a separate thread so as to remove redundant overheads while fetching of messages from buffer.

3.5 Other measures to enhance performance

Last section discussed design in detail and gave some perspective of measures taken in design to enhance performance. several key useful measures employed are summarized below:

- Thoughtful multi-threading is employed throughout the design, which means while high level of parallelization of tasks has been done, the design always considered the implications of creating new threads. no compromises have been made in terms of any kind of blocking of data flow. acknowledgments, clean up and other control mechanisms have all been designed by taking that into consideration.
- As found in background study, RabbitMQ performs optimally when the queues in RabbitMQ are empty or nearly empty. it has been a major consideration as queues are only used when data is transferred. in-memory queues are not used for caching in any way throughout the design.
- Special consideration have given to reliability of message delivery. acknowledgements and re-transmission mechanisms have been considered for each of the scenario involving end to end communication.
- Design aimed to present a system which is highly fault tolerant. while, Kafka's distributed architecture provides inherent fault tolerance, the design for communication between RSUs and CVs makes apparent that failure of a RSU doesn't affect the data transfer between CV and other working RSUs. the lifecycle of CV reverts to handshake phase in case connection can not be established with RSU. the data buffers conserve the data produced by CV and offset for consuming data is also conserved by CV.

3.6 Extensibility

The design has essentially proposed core middleware support for V2X communication, so, it is important to have scope for extensibility in the design. key considerations to ensure extensibility in the system designed are summarized in this section.

- Kafka provides inherent extensibility. although, this thesis does not explore usability of Kafka's connect API, it can be used to extend core messaging system's support to include processing of data. Kafka project maintains various connectors to different

technologies.

- The design has reserved scope for processing at RSUs, to provide possibility to have pre-processing capabilities at the edge of the system. the middleware support for RSU is essentially a module developed in JAVA programming language and most of the processing platforms support tools and libraries that can be utilized there.

4 Implementation

This chapter focuses on implementation of the proposed design and its evaluation conducted as part of this thesis. section 4.1 presents an overview of overall implementation and hardware setup employed. section 4.2 points out the key techniques used to simulate V2X environment and finally, section 4.3 presents the evaluation of implementation.

4.1 Overall implementation

The implementation was first done to develop a prototype of proposed design by developing three separate modules for CV, RSU and a generic service. later an evaluation module was added to evaluate the implementation. main performance metrics for a communication system are latency and throughput. so, the conducted evaluation of the system is focused on the same.
the evaluation is done for a somewhat stripped down version of the implementation, results are focused on data collection phase. the specifics of the simulation and each scenario is described in detail in following sections. only one topic is considered for the evaluation, although the implementation provides support for multiple topics due to constrained hardware set up. Kafka message delivery system is implemented by using one broker node and one zookeeper node. Hardware used for implementation is a regular laptop with 16 GB RAM and quad core i7 intel processor.

4.2 Simulating a V2X environment

V2X environment was simulated by hard coding it in implementation. several considerations in implementation were taken to simulate characteristics of V2X environment. CAR module simulates behaviour of a car on a straight road of travelling at a constant speed of 10m/sec. CAR module polls Evaluation module every 300msecs, with its position and other control information. evaluation module reverts with control information necessary for the simulation. the simulation is designed to disconnect(reverting to handshake phase) and connect CAR module to a always active RSU module and simulate message drops. following subsections

describe key key considerations taken in simulation to sum up the test environment.

Simulating sensor data

Simulated sensor data is synthesised by evaluation module and sent to CAR module where CAR stores this data in a buffer, simulating the behaviour of data production as described in the design in last chapter. evaluation module sends messages of size 400KBs every 300 msecs to simulate high frequency data streams.

Switching between RSUs

The switching between RSUs is simulated by control information sent by evaluation module. the evaluation module is updated with CAR's information regularly, and sends a clear signal to CV to make a switch as it moves out of RSU's range. the RSU when indicated to make a switch, stops all the threads in CAR module responsible for producing/consuming data to simulate termination of connection. CV then starts back from handshake phase to simulate connecting to a new RSU. the scenario simulates CAR travelling on 2000m road, with RSUs at every 500m starting from 0m. range of each of the RSU is assumed to be 250 meters.

Simulating packet drops

As the hardware set up is a regular laptop with all the modules communicating with each other in the same localhost, packet loss between any communication is unlikely here. so, packet drops are also simulated to get realistic evaluation of the implementation. simulated packet drops are also controlled by evaluation module. evaluation module calculates the number of max number of consecutive messages to be assumed to be confirmed by RSU, by using following formula,

$$consecutiveSuccessfulMessages = maxConsecutiveAcks * dropRate$$

maxConsecutiveAcks is set at 20 and the drop rate is calculated considering the position of the CAR relative to RSUs by, using the formula,

$$dropRate = ((1 - dropConstant) * (distanceFromNearestRSU)/(rangeOfRSU + 50))$$

value of dropConstant is taken as 0.005, rangeOfRSU is 250 and distanceFromNearestRSU is calculated based on position of the CAR. note that rangeOfRSU + 50 is used in the denominator as even as 250m is reached, the simulation tries to simulate weak signals. although, the switching of RSU is initiated at 250m.
this calculated max number of consecutive successful messages is sent to CV in control information contained in responses to regular updates that evaluation module receives. so,

every 300msecs, CV is updated with this information.

4.3 Simulations and results

4.3.1 Average CV to RSU latency as CV moves between RSUs

This scenario uses evaluation module to draw results for the effect on average latency as CV moves between RSUs. for this scenario, evaluation module uses messages sent by RSU containing timestamps as each message is sent from CV and received at RSU. first latency evaluation presented is shown in Figure 4.1 and Figure 4.2. Figure 4.1 presents the number of average retransmissions attempted by messages as the CV travels from 0 to 500m and switches between RSUs. Figure 4.2 shows the effect on average latency of the messages transmitted. it should be noted that even though the implementation was done to simulate message drops, these drops were simulated by manually ignoring the acknowledgement by RSUs. the messages were forced to retransmit to simulate message drops. all the re transmitted messages had the timestamp of earliest message that was produced from CV to ensure that latency results were as accurate as possible. this evaluation shows that latency of the system drops as the switch between RSU happens, but the system is able to recover quickly.

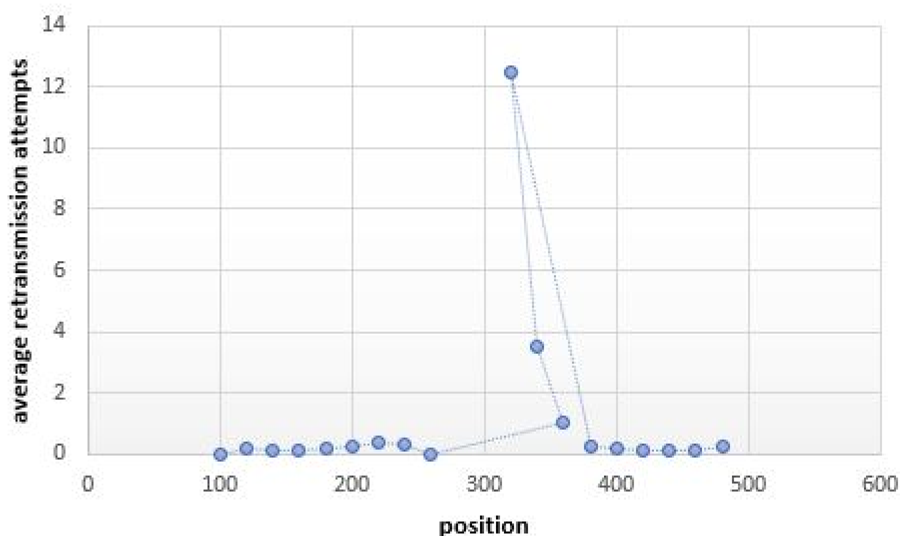

Figure 4.1: Average re-transmission attempts vs position

although, the peak latency in Figure 4.2 is very large, other results have indicated that it is not always the case. although, due to the large peak here, it leaves the impression that latency throughout the communication is stable. to give an objective finding relating to the system's performance, Figure 4.3 is presented, which has some noticeable differences from the previous one. first, it can be seen that latency of the system is unstable throughout the communication which is not profound in previous plot. the proposed design employed buffers

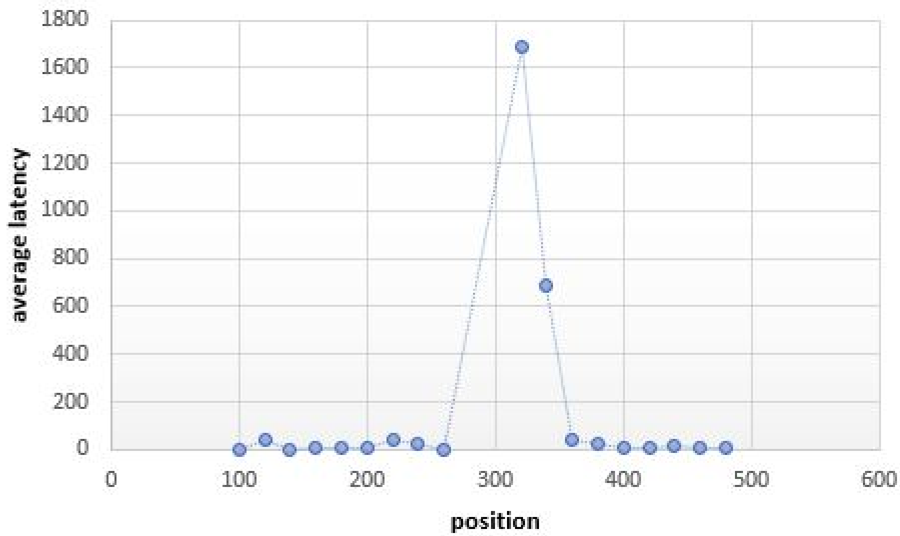

Figure 4.2: average latency vs position

and other design choices to provide a seamless flow of time. an implication of the design choices made was that ordering in messages can not be guaranteed. this means there could be messages in buffers that might potentially cause instability in latency performance. this could be attributed to as the reason behind the instability seen. the other noticeable difference is that peak latency occurs earlier than it occurred in last plot. this demonstrates that system was able to recover faster in this case, it also demonstrates that latency performance of the system could be unstable throughout the communication.

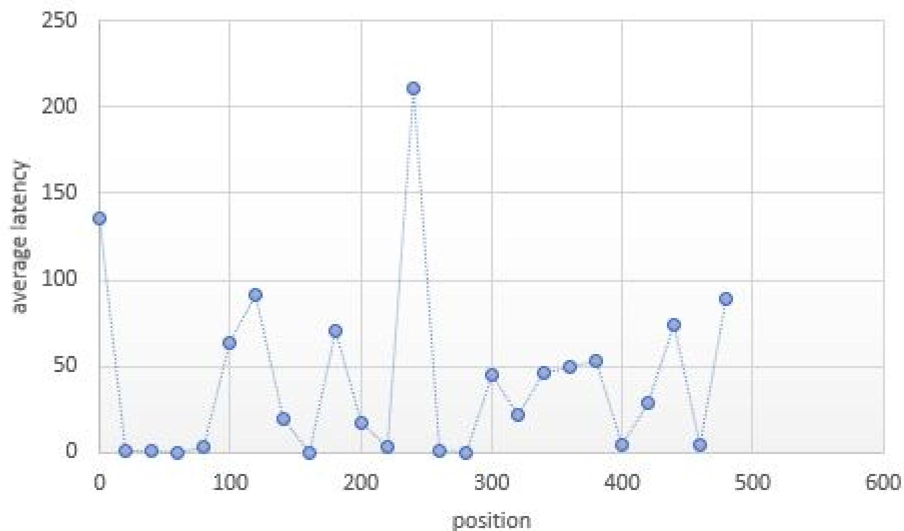

Figure 4.3: average latency vs position

4.3.2 Average CV to RSU latency vs time

Another interesting latency evaluation seemed to be the effect on latency as time progresses. for this scenario, speed of vehicle was slowed down to 5m/seconds while still keeping the same frequency of messages to observe the effect on latency, as communication progresses. the results were again produced using the evaluation module by averaging latency for every 2 minutes. Figure 4.4 presents the results produced. it can be seen that average latency produced is somewhat stable from the 2nd minute on, but the rise in latency between first two minutes suggests that the system is not able to keep up with the frequency of data produced.

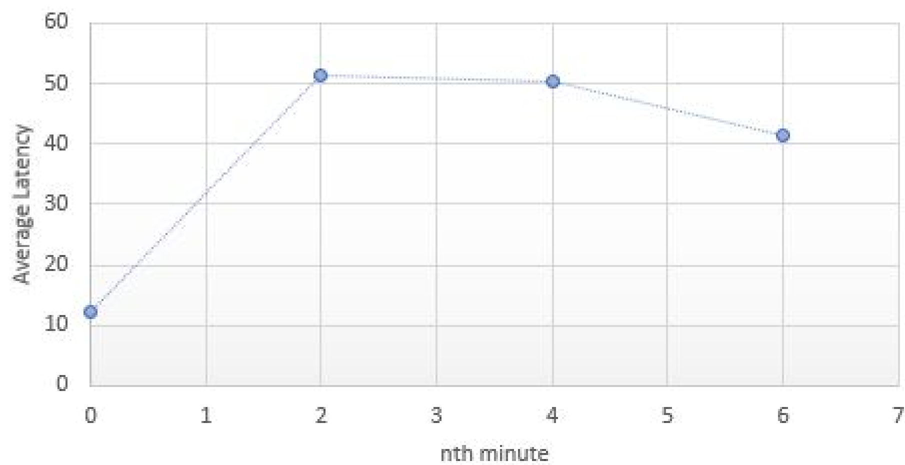

Figure 4.4: average latency vs time

4.3.3 Throughput Analysis

As mentioned before, the hardware set up is a resource constrained system. so, throughput analysis of the system was performed by developing a baseline end to end to communication with RabbitMQ. the baseline produced data on RabbitMQ which was consumed by RabbitMQ consumer and produced to Kafka in continuation, which are consumed by a Kafka consumer. the baseline's implementation produced data at same frequency, i.e., 300msecs with each message containing payload of size 500KBs, which is 100KBs more than the size of packets used in analysis of the system proposed in this thesis, to accommodate for overheads in messages being transmitted in the system's implementation. to perform throughput analysis of baseline, end to end throughput is measured from baseline implementation by recording the number of packets received by the Kafka consumer and the time elapsed since the first packet was transmitted. to perform the throughput analysis of the proposed system's implementation similar technique was used, but at both RSU module

(RabbitMQ consumer) and a generic external service's consumer module (Kafka consumer). the throughput analysis was performed separately for RabbitMQ based collection system's implementation and Kafka based messaging system's implementation by starting the external service after data is already produced to Kafka broker to have a independent result for both parts of the system.
In the baseline, it was observed that 18842 packets were received in 581674msecs, which sums up to a throughput of 32.39pps or 16.195MBps. RabbitMQ based collection system in the implementation of the proposed design observed 2379 packets delivered in 418920msecs, which gives us a throughput of 5.6pps or 2.8MBps(if packet size is considered to be 500KB, including overheads), while, Kafka based messaging system observed delivery of 5825 packets delivered in 451832 msecs, which sums up to 12.89pps or 6.94MBps throughput. these findings are summarized in Table 4.1 and 4.2.

Table 4.1: Baseline End to End throughput

Baseline Throughput
32.39pps/16.195MBps

Table 4.2: Throughput of proposed system

RabbitMQ based Collection	**Kafka based Core Messaging**
5.6pps/2.8MBps	12.89pps/6.94MBps

5 Summary and conclusions

The work on this thesis started with a background study on existing standards and technologies relevant to V2X with aim to present a communication system able to fulfil challenges of advanced V2X applications. the background study in thesis lead to basing the infrastructure's physical and data link layer on IEEE standards which introduced challenges to be addressed between CV and RSU communication. In addition, the study on similar works helped in setting the scope for project and focus on the challenges of a collection and delivery system for CVs and a Kafka based core messaging system. the design proposed took careful considerations for providing seamless flow of data streams reliably to maximize latency and throughput performances. design choices were made taking scalability, fault tolerance and reliability into consideration.
The results in Chapter 4 show that system is able to provide somewhat stable latency performance with quick recovery while switching between RSUs. the design, however, puts a strain on the throughput performance. the optimizations need to be made to the implementation and the design to support advanced V2X applications that the project aimed for. the resource constrained setup is not ideal for the implementation as the simulation of environment utilized processing resources that potentially could have slowed the performance of implementation. a well suited test bed may possibly gather better performances.

5.1 Future Works

Next steps for the project start with implementing the design on a well suited test bed that could capture the capabilities of the design in a real world scenario. possible optimizations to the design can be made, for instance, choice of language can possibly be improved.

The design provided core communication support. it essentially just puts out a blueprint that, as part of this thesis implemented on application level. there is a clear scope to develop a middleware on lower level that could implement the design in a more efficient way. finally, the design can be extended to include actual processing to completely realize the potential it offers.

Bibliography

[1] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 227–238, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350655. doi: 10.1145/3093742.3093908. URL https://doi.org/10.1145/3093742.3093908.

[2] Yuheng Du, Mashrur Chowdhury, Mizanur Rahman, Kakan Dey, Amy Apon, Andre Luckow, and Linh Bao Ngo. A distributed message delivery infrastructure for connected vehicle technology applications. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):787–801, 2018. doi: 10.1109/TITS.2017.2701799.

[3] Ádám Knapp, András Wippelhauser, Dániel Magyar, and Győző Gódor. An overview of current and future vehicular communication technologies. *Periodica Polytechnica Transportation Engineering*, 48(4):341–348, 2020. doi: 10.3311/PPtr.15922. URL https://pp.bme.hu/tr/article/view/15922.

[4] Md. Imrul Hassan, Hai Vu, and Taka Sakurai. Performance analysis of the ieee 802.11 mac protocol for dsrc safety applications. *Vehicular Technology, IEEE Transactions on*, 60:3882–3896, 10 2011. doi: 10.1109/TVT.2011.2162755.

[5] Gaurang Naik, Biplav Choudhury, and Jung-Min Park. Ieee 802.11bd amp; 5g nr v2x: Evolution of radio access technologies for v2x communications. *IEEE Access*, 7: 70169–70184, 2019. doi: 10.1109/ACCESS.2019.2919489.

[6] Rafael Molina-Masegosa and Javier Gozalvez. Lte-v for sidelink 5g v2x vehicular communications: A new 5g technology for short-range vehicle-to-everything communications. *IEEE Vehicular Technology Magazine*, 12(4):30–39, 2017. doi: 10.1109/MVT.2017.2752798.

[7] Waqar Anwar, Norman Franchi, and Gerhard Fettweis. Physical layer evaluation of v2x communications technologies: 5g nr-v2x, lte-v2x, ieee 802.11bd, and ieee 802.11p. In

2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall), pages 1–7, 2019. doi: 10.1109/VTCFall.2019.8891313.

[8] Valerian Mannoni, Vincent Berg, Stefania Sesia, and Eric Perraud. A comparison of the v2x communication systems: Its-g5 and c-v2x. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pages 1–5, 2019. doi: 10.1109/VTCSpring.2019.8746562.

[9] Guo Fu, Yanfeng Zhang, and Ge Yu. A fair comparison of message queuing systems. *IEEE Access*, 9:421–432, 2021. doi: 10.1109/ACCESS.2020.3046503.

[10] Zhenghe Wang, Wei Dai, Feng Wang, Hui Deng, Shoulin Wei, Xiaoli Zhang, and Bo Liang. Kafka and its using in high-throughput and reliable message distribution. In *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pages 117–120, 2015. doi: 10.1109/ICINIS.2015.53.

[11] Eyhab Al-Masri, Karan Raj Kalyanam, John Batts, Jonathan Kim, Sharanjit Singh, Tammy Vo, and Charlotte Yan. Investigating messaging protocols for the internet of things (iot). *IEEE Access*, 8:94880–94911, 2020. doi: 10.1109/ACCESS.2020.2993363.

[12] Istabraq Al-Joboury and Emad Al-Hemiary. Performance analysis of internet of things protocols based fog/cloud over high traffic. *Journal of Fundamental and Applied Sciences*, 10:176–181, 03 2018. doi: 10.4314/jfas.v10i6s.113.

[13] Dinesh Thangavel, Xiaoping Ma, A. Valera, H. Tan, and C. Tan. Performance evaluation of mqtt and coap via a common middleware. *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014.

[14] Pinchen Cui. Comparison of iot application layer protocols. 2017.

[15] Jetendra Joshi, Vishal Rajapriya, S.R. Rahul, Pranith Kumar, Siddhanth Polepally, Rohit Samineni, and D.G. Kamal Tej. Performance enhancement and iot based monitoring for smart home. In *2017 International Conference on Information Networking (ICOIN)*, pages 468–473, 2017. doi: 10.1109/ICOIN.2017.7899537.

[16] Joel L. Fernandes, Ivo C. Lopes, Joel J. P. C. Rodrigues, and Sana Ullah. Performance evaluation of restful web services and amqp protocol. In *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 810–815, 2013. doi: 10.1109/ICUFN.2013.6614932.

[17] Biswajeeban Mishra. *Performance Evaluation of MQTT Broker Servers*, pages 599–609. 07 2018. ISBN 978-3-319-95170-6. doi: 10.1007/978-3-319-95171-3_47.

[18] Aelee Yoo, Sooyeon Shin, Junwon Lee, and Changjoo Moon. Implementation of a sensor big data processing system for autonomous vehicles in the c-its environment. *Applied Sciences*, 10:7858, 11 2020. doi: 10.3390/app10217858.

[19] Eric Simmon, Kyoung sook Kim, Eswaran Subrahmanian, Ryong Lee, Frederic de, Yohei Murakami, Koji Zettsu, and Ram Sriram. A vision of cyber-physical cloud computing for smart networked systems, 2013-08-26 2013.

[20] Connected vehicle reference implementation architecture (cvria), accessed on mar. 2016. [online]. available: http://www.iteris.com/cvria/html/applications/applications.html.

[21] Åsmund Hugo, Brice Morin, and Karl Svantorp. Bridging mqtt and kafka to support c-its: a feasibility study. In *2020 21st IEEE International Conference on Mobile Data Management (MDM)*, pages 371–376, 2020. doi: 10.1109/MDM48529.2020.00080.