December 2020

COSACC: Cloud-Based Speed Advisory for Connected Vehicles in a Signalized Corridor

Hsien-Wen Deng
Clemson University, hsienwd@clemson.edu

COSACC: CLOUD-BASED SPEED ADVISORY FOR CONNECTED VEHICLES IN A SIGNALIZED CORRIDOR

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Hsien-Wen Deng
December 2020

Accepted by:
Amy W. Apon, Committee Chair
Mashrur Chowdhury, Co-Chair
Mitch Shue

ABSTRACT

The objective of this study is to assess the feasibility of cloud-based real-time connected vehicle (CV) applications. The author developed a cloud-based speed advisory application for CVs in a signalized corridor (COSACC) to achieve this objective. The contribution of this study is threefold. First, it introduced a serverless cloud computing architecture using Amazon Web Services (AWS) for real-time CV applications. Second, the author developed a real-time optimization-based speed advisory algorithm that is deployable in AWS. Third, this study utilized a cloud-in-the-loop simulation testbed using AWS and Simulation of Urban Mobility (SUMO), which is a microscopic traffic simulator. The author conducted experiments on cloud access at three-hour intervals over 24 hours in one day. These experiments revealed that the total data upload and download time to and from AWS via LTE is on average 92 milliseconds, which meets the allowable delay requirement for real-time CV traffic mobility applications. The author conducted a case study by implementing the COSACC in a cloud-in-the-loop simulation testbed. The analyses revealed that COSACC can reduce vehicle stopped delay at the signalized intersections up to 98% and fuel consumption in the signalized corridor up to 12.7%, compared to the baseline scenario, i.e., no speed advisory on the signalized corridor. Moreover, the authors observed an average end-to-end delay from a CV sending basic safety messages to it receiving a speed advisory from the cloud to be about 443 ms, which is well under the 1000 ms threshold required for any real-time traffic mobility application for connected vehicles.

DEDICATION

To

My beloved parents

Who work all day to support me in achieving my dreams

Dr. Chowdhury and Dr. Apon

For supporting and providing advice for this project from start to end

Prof. Shue

For knowledge and suggestions on cloud computing

Dr. Rahman

For guiding me on every aspect from the beginning

M. Sabbir Salek

For constantly working with me and contributing useful content

ACKNOWLEDGMENTS

This study is partially funded by the Center for Connected Multimodal Mobility (C^2M^2) (a Tier 1 USDOT University Transportation Center) headquartered at Clemson University, Clemson, South Carolina, USA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the USDOT Center for Connected Multimodal Mobility (C^2M^2), and the U.S. Government assumes no liability for the contents or use thereof.

I wish to express my gratitude to Dr. Chowdhury and Dr. Apon for their support of this project, Prof. Shue for his extensive knowledge from the industry and generous help on the thesis, Dr. Rahman for his selfless guidance to me from the start of the study and M. Sabbir Salek for kindly working with me and for his contributions to my methodologies.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

As the sources of connected vehicle (CV) data increase and become more diverse [1], infrastructure to support these data must be scaled up to accommodate many types of very large data sets related to connected and automated vehicles (CAV), such as sensor data, text messages, and images [2]. Furthermore, these infrastructures must securely support real-time processing for many applications, must be trustworthy, and must preserve privacy [3]. Public clouds have evolved to a level capable of supporting applications while also meeting these requirements. Recent advances in public cloud infrastructure address these issues in a cost-competitive manner vis-a-vis in-house infrastructure development and labor costs. For example, the Amazon Web Service (AWS) GovCloud region supports significant levels of high speed and secure data processing, which makes a case for an investigation of CV applications utilizing public cloud technologies. Additionally, Google Cloud Platform (GCP) and Microsoft Azure are also competing in the commercial cloud space [4][5][6].

In CV applications, there are constant streams of data to be processed and analyzed, often in real-time. This means that there is a deadline to complete the analysis, after which the results of the analysis could become useless. Cloud computing could be a cost-effective solution for backend data processing and archiving, which are required for many CV applications. This study utilized a serverless cloud computing architecture with a real-time platoon-based speed advisory algorithm using AWS to assess the feasibility of commercial cloud services for CV mobility applications. The serverless architecture features a pay-asyou-go model without the burden of managing resources and systems as would be the case when using traditional server-based cloud services. This architecture is defined as Function-as-a-Service (FaaS), which can be triggered and executed as per application requirements after deployment [7][8]. Many services such as stream services, data storage, and notification services can stimulate a CV application to be executed in the serverless architecture.

The author developed a cloud-based, platoon-based speed advisory application that functions in real-time for connected vehicles in a signalized corridor (COSACC). CVs and roadway traffic signals send information of the conditions (i.e., location and speed of CVs and traffic signal phasing and timing information) to the serverless cloud computing infrastructure and trigger execution of the speed advisory application automatically. COSACC is more scalable in terms of communication coverage area and number of CVs due to high availability and scalability of cloud computing compared to an application supported by edge computing without any cloud infrastructure.

The author conducted several focused experiments in the field to test the feasibility of COSACC. In addition, the author also developed a cloud-in-the-loop simulation testbed using AWS and Simulation of Urban Mobility (SUMO), which is a microscopic roadway traffic simulator, to evaluate COSACC at a macroscopic level.

In Chapter Two, this thesis discusses three things: 1. the general architecture for a cloud-based CV application, 2. previous works, and 3. real-time speed advisory applications. In Chapter Three, the author delineates the architectural view of COSACC based on the serverless features provided by AWS. Chapter Four discusses some of the author’s attempts to improve the scalability and performance of the application in AWS and lessons learned from them. Chapter Five describes multiple experiments at a microscopic level concerning evaluating the feasibility of cloud computing for CVs. In Chapter Five, the author presents COSACC, a novel platoon-based speed advisory application. Chapter Six discusses the results of a case study with a cloud-in-the-loop simulation to evaluate the feasibility of COSACC macroscopically. Lastly, in Chapter Seven, the author presents the potential of CV applications supported by cloud computing.

CHAPTER TWO

RELATED WORKS

▪ Cloud Computing for CVs

In general, a Connected and Autonomous Vehicle (CAV) can be considered to be a mobile node with a group of sensors (i.e., GPS, Radar, camera, LIDAR). Cloud infrastructure and transportation infrastructure through V2I communication, use wireless (e.g., DSRC, Long-Term Evolution (LTE), 5G) or wired (e.g., optical fiber) communication technologies. Services in the cloud can be used to aggregate and analyze the collected data and generate appropriate responses based on the data.

A CV sends basic safety messages (BSM) to the cloud using wireless communication. On the other hand, traffic control infrastructure (i.e., traffic signals) sends its traffic control information, such as signal phase and timing (SPaT) messages to the cloud through wired or wireless communication. Data included in a BSM are shown in Table 1 [9]. Traffic management centers share their recorded roadway traffic information with the cloud.

TABLE 1
Basic Safety Message [9]

Type	Description	Size (byte)
DSRCmsgID	Data elements used in each message to define the Message type	1
MsgCount	Check the flow of consecutive messages having the same DSRCmsgID received from the same message sender	1

TemporaryID	Temporary device identifier. When used in a mobile on-board unit (OBU) device, this value is periodically changed to ensure anonymity	4
Dsecond	Time information	2
Latitude	Geographic latitude of an object	4
Longtitude	Geographic longtitude of an object	4
Elevation	Altitude measured by the WGS84 coordinate system	2
PositionAccuracy	Various quality parameters used to model the positioning accuracy for each given axis	4
TransmissionAndSpeed	Speed of the vehicle	2
Heading	Current direction value expressed in units of 0.0125 degrees	2
SteeringWheelAngle	Current steering angle of the steering wheel	1
AccelerationSet4Way	Consists of three orthogonal directions of acceleration and yaw rate	7
BrakeSystemStatus	Data element that records various control states related to braking of the vehicle	1
VehicleSize	Length and width of the vehicle	3

Moreover, other transportation-related services, such as weather services, news services, and emergency management centers, can send their information into the cloud, too.

Inside the cloud architecture, a message broker exchanges data from producers (e.g., CVs) and consumers (e.g., CV applications). A typical cloud-based CV application includes three abstraction layers: infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and software-as-a-service (SaaS). CV application developers utilize IaaS to set up low-level requirements of an application, such as data storage, and operating system. Beyond the infrastructure, PaaS provides flexible and scalable services (e.g., database management and computing services) for developers to build their applications. Last but not the least, CV applications are implemented on the SaaS layer. Developers can implement real-time or non-real-time applications according to CV requirements. Figure 2.1 shows a cloud-based architecture for CV applications development. Many studies have now used the cloud as a platform to develop a CV application. Ning et al. utilized a cloud-based fog computing architecture to implement real-time traffic management [10]. Li et. al. provided a maximum value density-based heuristic algorithm through vehicular edge cloud computing to develop a traffic energy efficiency application [11], and Jin et. al. presented a method of constructing cloud-based mobility services for connected and automated vehicles highway [12]. Each of these studies used a traditional server-based approach to develop real-time CV applications. However, according to the author's knowledge, no studies have used a serverless architecture as the platform for the development of a real-time CV application.

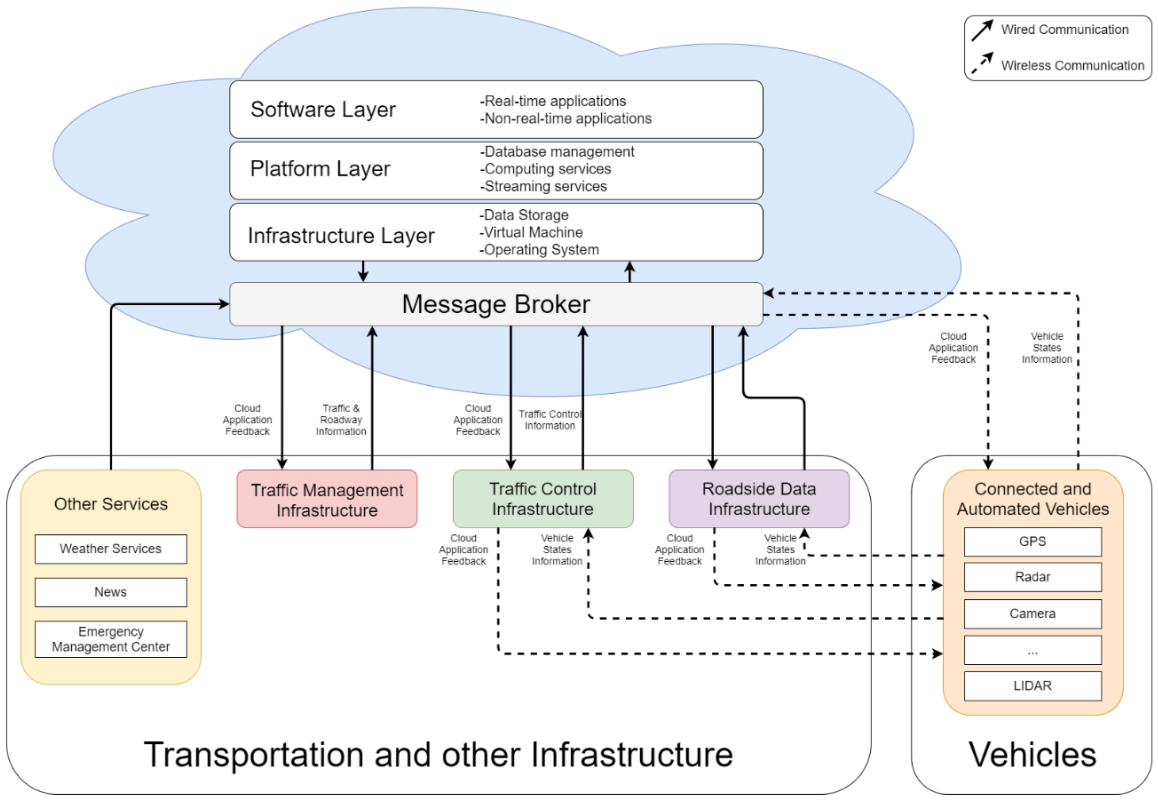
Wired Communication
Wireless Communication
Software Layer
-Real-time applications
-Non-real-time applications
Platform Layer
-Database management
-Computing services
-Streaming services
Infrastructure Layer
-Data Storage
-Virtual Machine
-Operating System
Message Broker
Cloud Application Feedback
Vehicle States Information
Cloud Application Feedback
Traffic & Roadway Information
Cloud Application Feedback
Traffic Control Information
Other Services
Weather Services
News
Emergency Management Center
Traffic Management Infrastructure
Traffic Control Infrastructure
Roadside Data Infrastructure
Connected and Automated Vehicles
GPS
Radar
Camera
...
LIDAR
Cloud Application Feedback
Vehicle States Information
Cloud Application Feedback
Vehicle States Information
Transportation and other Infrastructure
Vehicles

Figure 2.1: Architecture for a cloud-based CV application

- **Cloud-based Real-time Speed Advisory Application**

In the field of traffic engineering, many methods are developed to integrate traffic signal data and vehicular information to enhance traffic operational efficiency [13]. However, with increasing traffic demands and new technology, it is necessary to develop these applications in smarter ways. Intelligent Transportation System (ITS) develops smart solutions for improving traffic operational efficiency and safety. However, investing in developing roadside data infrastructure and maintaining it may not be a cost-effective solution. Thus, the use of cloud computing could be a viable solution for real-time CV application development to improve traffic efficiency and reduce a significant amount of infrastructure deployment cost. Figure 2.2 presents a physical architecture of the cloud-based speed advisory application for CVs. The speed advisory application can be deployed in cloud infrastructure that collects vehicle location and motion information provided by CVs, and signal phase and timing (SPaT) information provided by roadside infrastructures (i.e., traffic signal). By analyzing and aggregating these data, the speed advisory application can generate an appropriate speed advisory for CVs. However, computing time for running the speed advisory application in the cloud and data exchange delay between the cloud and CVs must satisfy real-time requirements.

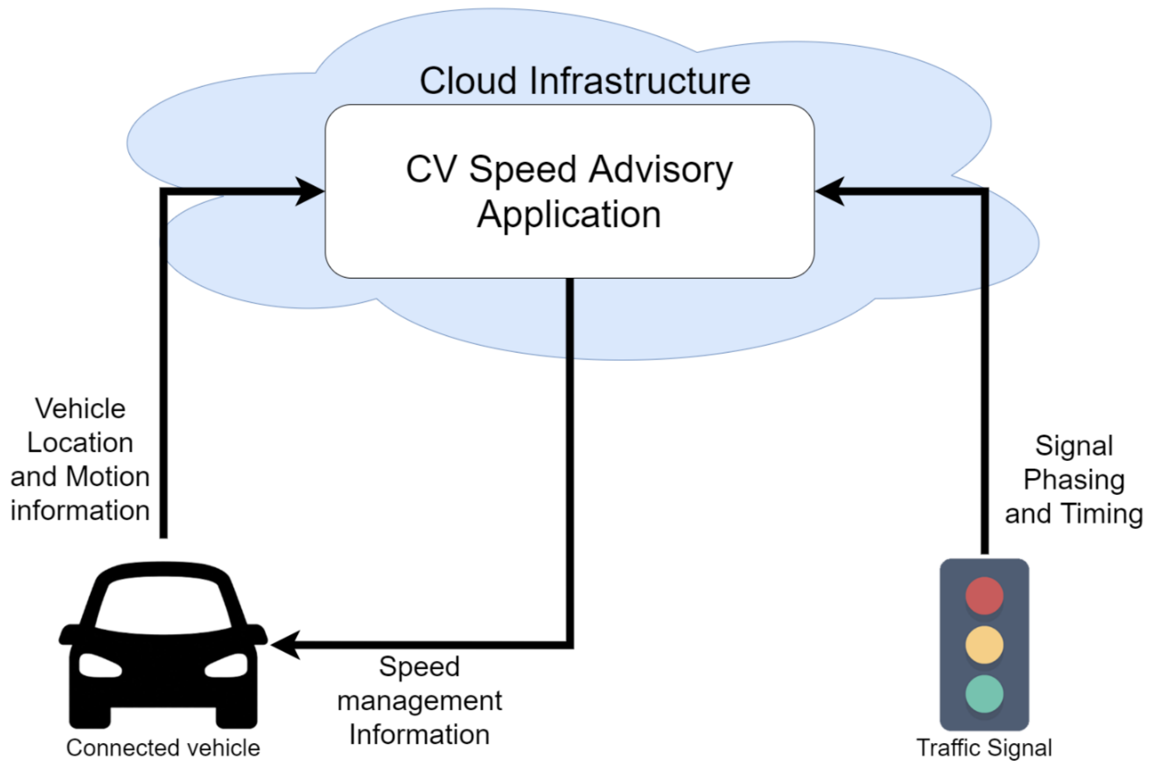
Cloud Infrastructure
CV Speed Advisory Application
Vehicle Location and Motion information
Signal Phasing and Timing
Speed management Information
Connected vehicle
Traffic Signal

Figure 2.2: Cloud-based speed advisory application for CVs

The Green Light Optimal Speed Advisory (GLOSA) algorithm tries to optimize stopped delay, fuel consumption, and CO_2 emission while processing vehicles through signalized intersections. Among the recent works, Suzuki and Marumo developed a GLOSA system that projects a green rectangle on the roadway through the heads-up display of a GLOSA-enabled vehicle [14]. The green rectangle is an advised area for that vehicle to cross the intersection within the allocated green-time. Simchon and Rabinovici developed a dynamic GLOSA system for real-time implementation on roadways [15]. The authors utilized a relaxation procedure to cut the computation time short for real-time implementation. Stebbins et al. combined model predictive control (MPC) with state-space reduction and GLOSA to yield efficient trajectories for the CVs [16]. However, very few studies have considered platoon formation. In [17], Stebbins et al. developed a platoon-based optimization technique for GLOSA. The authors included a safety constraint in consideration of the fact that, in some situations, human drivers may not follow an advised speed if they feel that they would not be able to stop their vehicles while approaching an intersection. Zhao et al. developed a platoon-based MPC to optimize fuel consumption while enabling a platoon of vehicles to pass an intersection within a green interval [18]. The authors evaluated the efficacy of their model for different CV penetration rates. However, none of these studies considered a real-time implementation of a platoon-based system running in the cloud for speed advisory at a signalized corridor. In this study, the authors developed a cloud-based, real-time speed advisory application using a platooning concept.

CHAPTER THREE

CLOUD-BASED SERVERLESS ARCHITECTURE

In a traditional cloud computing architecture, a skilled system administrator is required to create a CV application development and implementation environment (e.g., virtual machines configuration, operating system installation, and computing resource management) manually. Whereas, a serverless cloud architecture provides functionalities to a user without hardcoding integration (embedding data directly into the source code). Furthermore, a CV application developer is not required to create and maintain computing instances or to configure a cloud environment as per CV application requirements. This would significantly reduce application development time, as the developers of CV applications would only need to focus on the development and implementation of the algorithm. Typically, serverless computing services support various languages, such as Node.JS, Python, .NET, and Java. The primary advantage of cloud-based CV application development is that an application deployed using a serverless architecture can work with other cloud services (e.g., data storage, streaming services), the event-driven feature makes the application can be triggered to launch.

Currently available commercial cloud services support a serverless cloud architecture in addition to a more traditional server-based cloud architecture. Microsoft introduced "Azure Functions," which allow users to develop event-driven applications utilizing Visual Studio, as a part of the serverless computing services [19]. Google Cloud Platform (GCP) offers its serverless computing application programming interface (API), called “Knative,” for developers to manage their applications using the serverless cloud computing services along with other GCP services, such as cloud storage and data flow services [20]. AWS provides “Lambda” as its serverless method [21]. A user can deploy any CV application using Lambda cooperated with other AWS services (e.g. DynamoDB, Kinesis Data Stream).

In this study, the author developed a cloud-based serverless architecture for developing a real-time speed advisory application using AWS for CVs along a signalized corridor. However, the same approach can be applied to the development of serverless architecture for CV applications using other commercial cloud services, such as Microsoft Azure and GCP. AWS, as well as Microsoft and Google, maintains a vast cloud infrastructure and services, which makes it highly available and scalable for real-time CV applications.

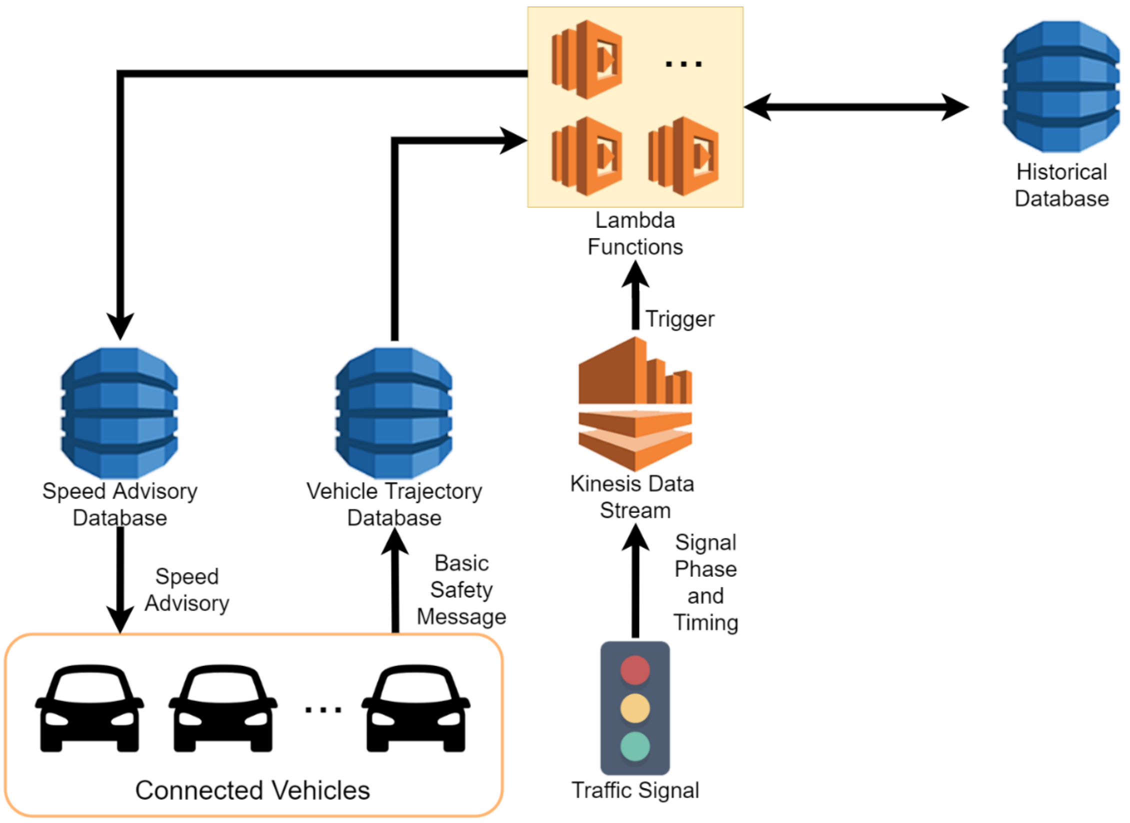

Figure 3.1: Serverless COSACC architecture.

The author presents a serverless architecture that can manage computing resources, databases, and streaming services for COSACC so that the application can run in real-time while guaranteeing Quality of Service (QoS). Figure 3.1 illustrates the COSACC architecture and lists the AWS services used, which include (i) DynamoDB, (ii) Kinesis Data Stream, and (iii) Lambda.

DynamoDB is a NoSQL cloud database service with a key-value structure [22], which the author utilized to develop databases (i.e., a speed advisory database, a vehicle trajectory database, and a historical database). The author created a “vehicle trajectory database” for CVs to update their trajectory information and a speed advisory database to store a set of speed advisories calculated using the algorithm developed in this study. CVs can subscribe to these messages from the speed advisory database. For each traffic signal, the author also created a historical database to save and update the distance between CVs and a traffic signal in real-time.

Kinesis Data Stream is a data stream service that continuously delivers data messages [23] in AWS. For each traffic signal, the author established a Kinesis Data Stream service for sending a streaming message into the cloud every second to trigger the speed advisory algorithm.

Lambda [21], which is the core component of our application, is a computation service for serverless COSACC architecture. The author designed a group of Lambda functions for every traffic signal triggered by the Kinesis Data Stream. Each Lambda function originally stores basic information, such as physical location and signal phase duration of the corresponding traffic signals. Once a Lambda function is triggered, it collects information from traffic signals and CVs, computes speed advisories based on our algorithm, and updates speed advisories for CVs.

In the real world, a CV generates a BSM, and each traffic signal generates SPaT messages. In COSACC, each CV uploads a filtered BSM, which contains the vehicle's ID, location, speed, and the gap between two successive vehicles of a CV into the vehicle trajectory database. Each traffic signal sends a filtered SPaT message containing the current traffic signal phase and the remaining time of that phase to the Kinesis Data Stream. Using this serverless architecture, the author developed a cloud-based optimization algorithm that utilizes these BSMs and SPaT messages to generate speed advisories for CVs in real-time.

CHAPTER FOUR

SCALABILITY OF THE CLOUD FOR INCREASING NUMBER OF CONNECTED VEHICLES

Due to the dynamic scaling characteristics of traffic conditions, COSACC in the cloud must scale up and down in response to changing traffic demands. Although a serverless architecture supports dynamic resource allocation, there are a few restrictions imposed by the commercial cloud providers to prevent applications from exceeding certain cloud infrastructure limits. For instance, AWS allows a Lambda function to utilize up to 3008 MB of memory when processing. Because of this, the cloud application may not be able to support a large number of CVs in real-time.

- **Single Core Single Thread**

The author originally established only one Lambda function (single core) 3008 MB memory configuration to handle all traffic conditions, as Figure 4.1 indicates. The Lambda function processed vehicles in a sequential manner (single thread). The author tested the Lambda function with single core and single thread under different numbers of vehicles (i.e., 50, 100, 150, 200) in simulation, as discussed in Chapter 5. However, as Figure 4.2 shows, the processing time increased as the number of connected vehicles increased. The average processing time was 689 ms in the case of 50 connected vehicles and increased to 1128 ms for 100 connected vehicles, 1694 ms for the 150 connected vehicles scenario, and 2603 ms for 200 connected vehicles. This simulation experiment suggested that using single core single thread to support real-time CV applications would be problematic in AWS.

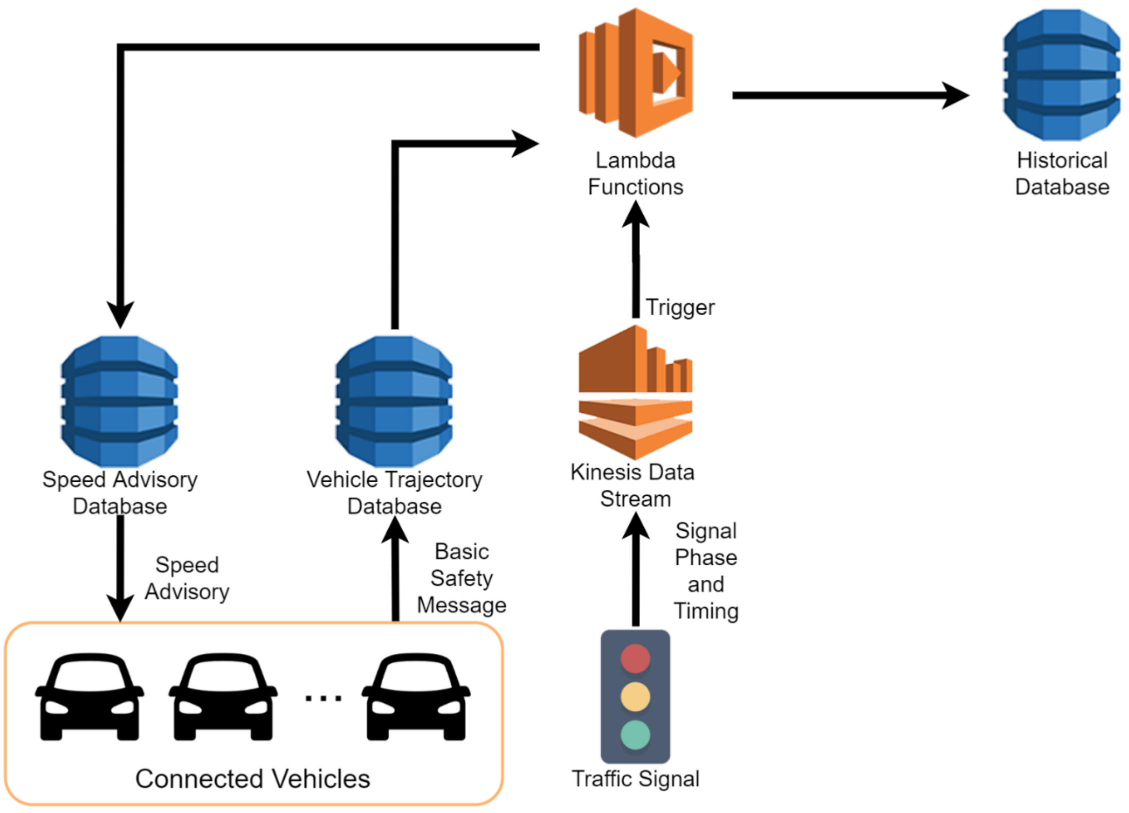

Figure 4.1: Serverless COSACC architecture with single core single thread.

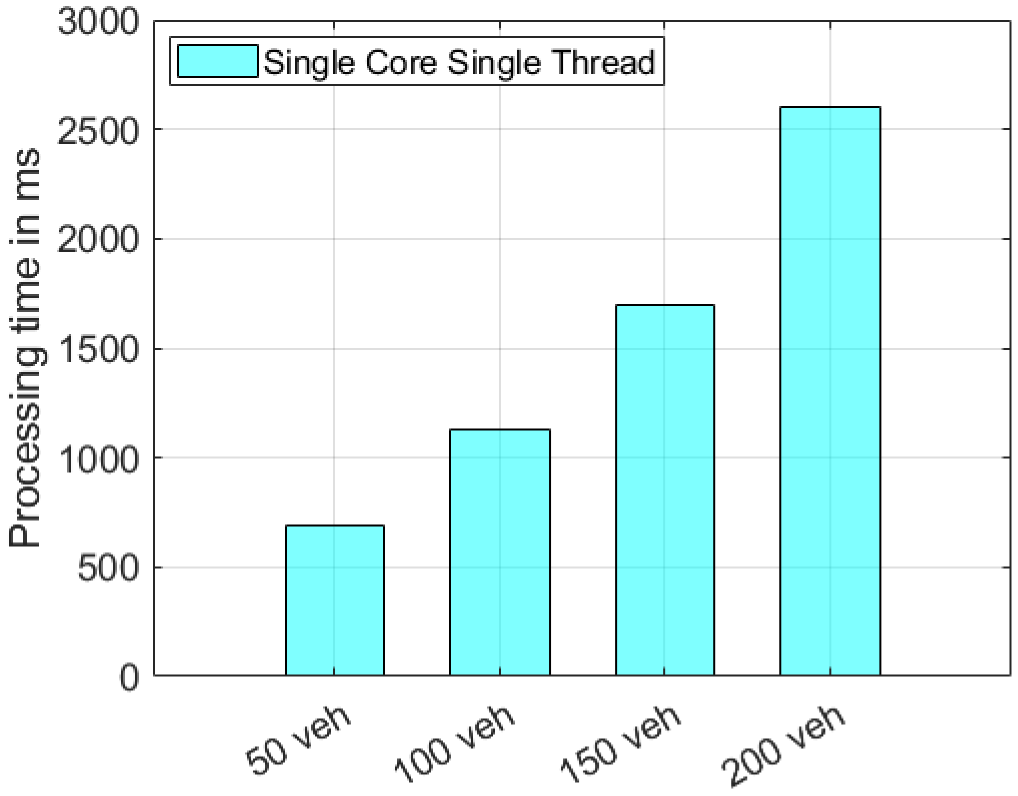

Figure 4.2: Processing time in Lambda with single core single thread.

▪ **Multithreading**

To tackle this issue, the author applied multithreading to AWS Lambda. Rather than adding vehicles into the computation sequentially, COSACC assigns vehicles into multiple threads before the data processing occurs with all vehicle data. Figure 4.3 shows that, compared to the single core single thread solution, applying multithreading significantly reduced the processing time in the cloud infrastructure. On average, the multithreaded Lambda approach was able to process 50 vehicles in 487 ms and 100 vehicles in 778 ms. Figure 4.3 shows the processing time for the multithreaded approach with an increasing number of vehicles. With 150 vehicles, the average processing time was 1193 ms, and with 200 vehicles, the processing time was 1522 ms, which still did not meet the acceptable delay requirement of a real-time traffic mobility application [24]. The experiments revealed that this approach, while an improvement, could not meet the delay requirement of a mobility application.

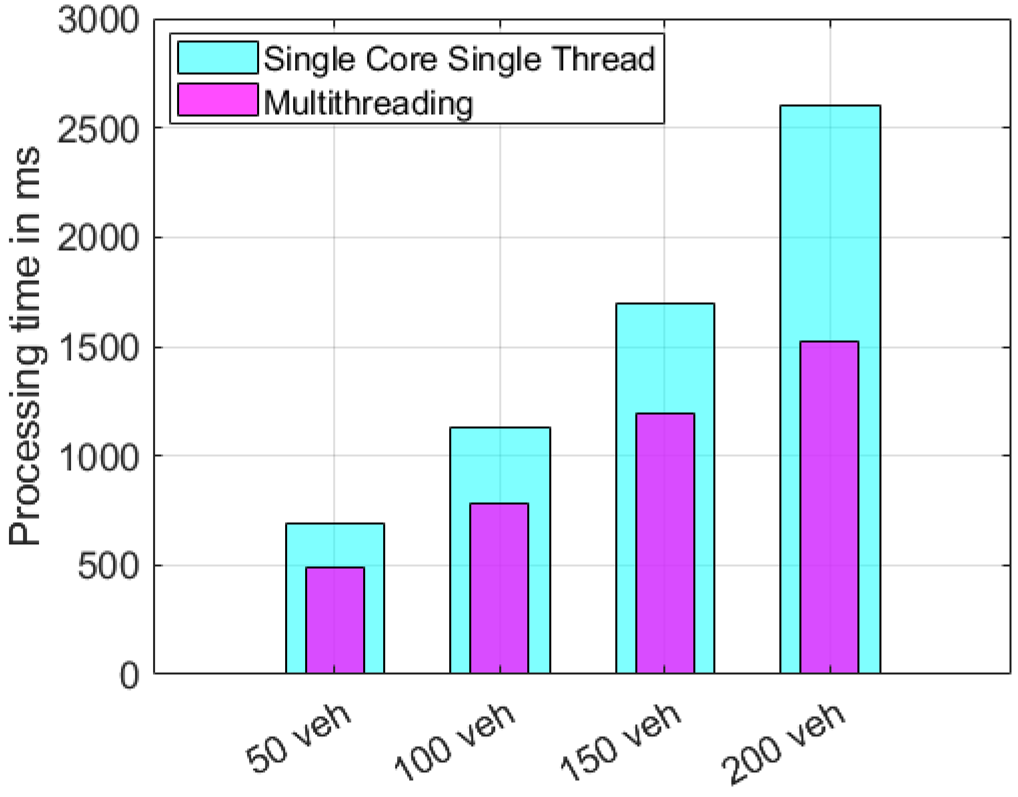

Figure 4.3: Processing time in Lambda with multithreading.

▪ **Parallel Computing**

The author next applied the concept of parallel computing in the cloud, as shown in Figure 3.1 (Chapter 3). Here, instead of employing a single Lambda function, the author established a fleet of Lambda functions for each traffic signal to handle different penetration levels of CVs. To make all Lambda functions work in real-time, the author defined a desired computing capacity for each Lambda function (i.e., the number of vehicles to process) to a maximum of 50 vehicles per Lambda function. All Lambda functions shared one trigger that started their parallel processing. This approach provided more computing power in the cloud for application processing to handle increasing computational demand. As shown in Figure 4.4, applying a parallel computing strategy with multiple Lambda functions achieved consistent and reduced processing time in the cloud even with different penetration levels of connected vehicles. Figure 4.4 shows that the average processing time in the cloud was about 250 ms for 50, 100, 150 and 200 connected vehicles, which is acceptable for a real-time traffic mobility application [24].

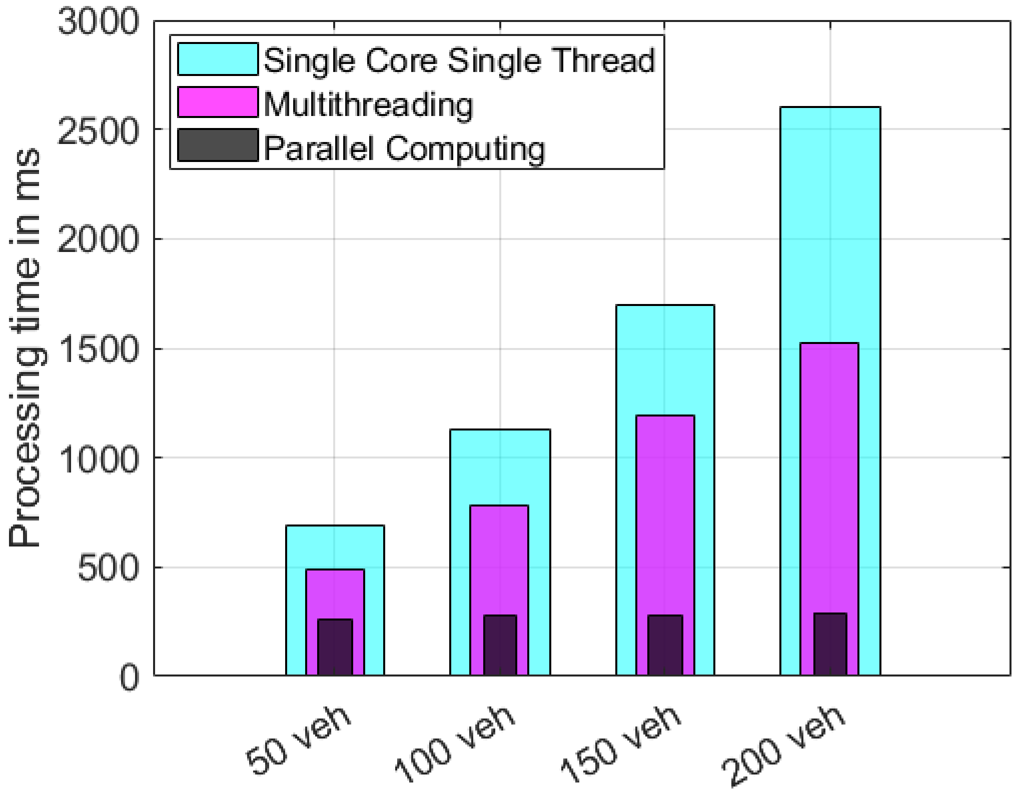

Figure 4.4: Processing time in Lambda with parallel computing.

CHAPTER FIVE

COMMUNICATION WITH THE CLOUD

It is necessary to meet wireless communication acceptable delay requirements between the vehicle and the cloud to guarantee QoS for real-time CV applications. In this section, the author describes several experiments conducted to examine the wireless communication delay when accessing AWS services for CV applications.

▪ Evaluation of cloud round-trip time

▪ *Experiment Setup*

The author established an Elastic Computing Cloud (EC2) instance in the AWS US-EAST-1 region (N. Virginia) to evaluate the communication delay between a CV and the cloud. Both the server (i.e., EC2) and the client (i.e., CV) were equipped with a socket program (see Appendix A) and communicated with each other using Transmission Control Protocol (TCP). To calculate the round-trip time (RTT), the client recorded a timestamp before sending a message to the server and then recorded another timestamp after receiving a message back from the server. Figure 5.1 presents the experimental setup between the server and the client.

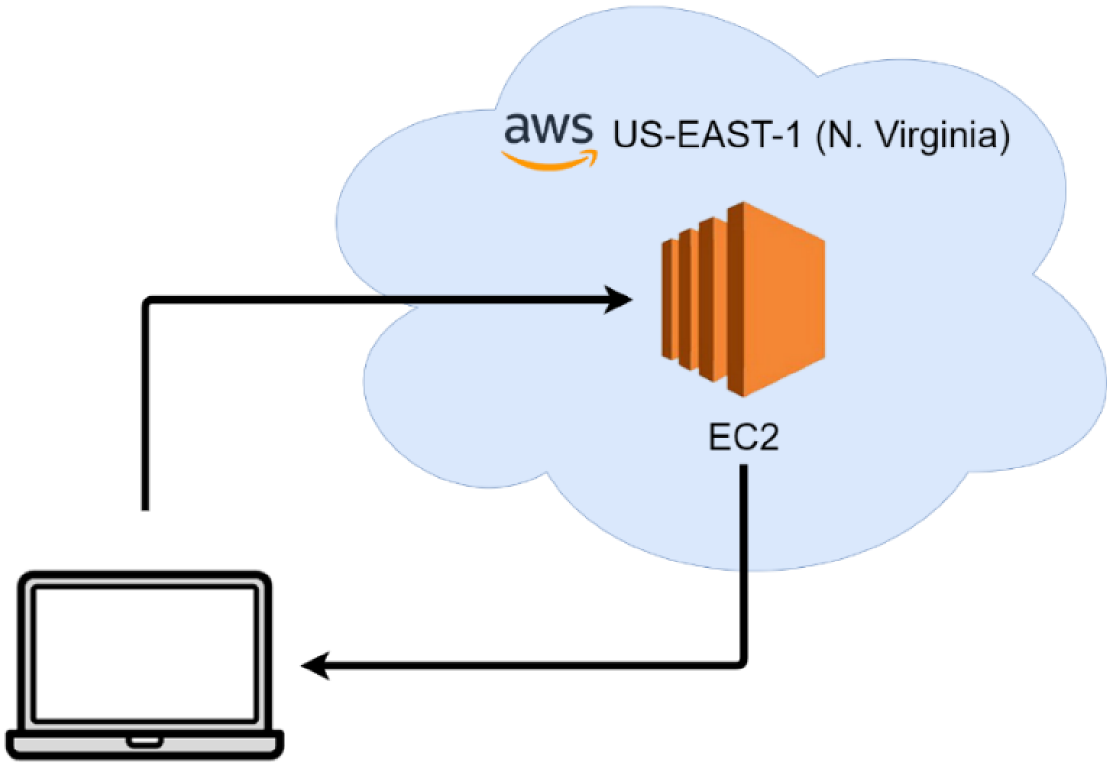

Figure 5.1: Overview of a socket program with AWS EC2

▪ *Evaluation scenarios*

The author conducted each experiment in three-hour increments for 24 hours in one day to determine the peak and off-peak hours of bandwidth usage. For each experiment, the author collected 1000 samples of RTTs.

▪ *Evaluation results*

Figure 5.2 illustrates that the average RTT over 24 hours was 72 milliseconds. Moreover, the RTT of 99 percent of the samples was within 500 milliseconds. In addition, the author found that the peak periods were from 11:00 AM to 02:00 PM

(14:00) and from 08:00 PM (20:00) to 11:00 PM (23:00), Eastern Daylight Time (EDT).

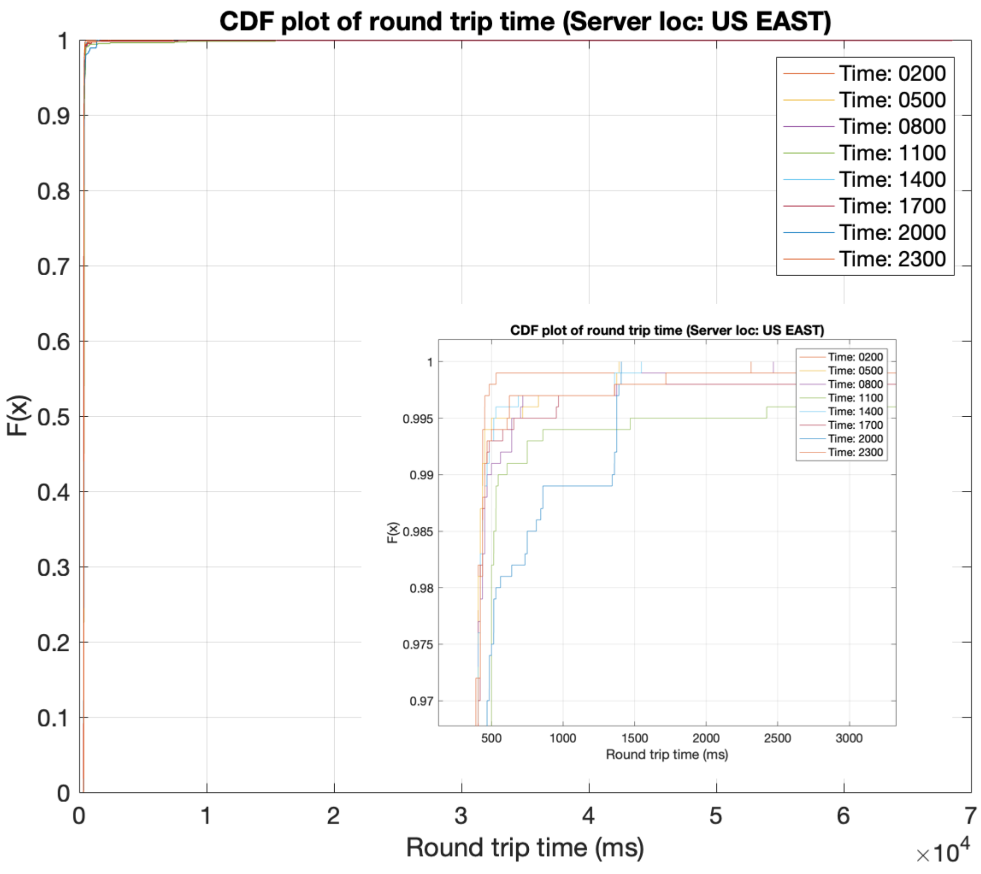

Figure 5.2: CDF results for Round-trip time accessing to AWS US-EAST-1

▪ Evaluation of Communication Delay Between a CV and the Cloud

▪ *Experiment Setup*

After developing the serverless architecture for COSACC using AWS, it was necessary to evaluate the communication delays between a CV and the cloud. As Figure 5.3 shows, two databases were built using DynamoDB in AWS: a vehicle trajectory database and a speed advisory database. A CV sends trajectory updates (i.e., BSMs) to the vehicular trajectory database and downloads its speed advisory from the speed advisory database. The author executed a Python script on a laptop (see Appendix B) to simulate a CV communicating with the cloud using LTE wireless communication. The experiment consisted of two parts: uploading data to the cloud and downloading data from the cloud. For the upload experiment, the author collected upload times for BSMs to the cloud from the computer. For the download experiment, the author collected download times for a speed advisory from the cloud to the computer.

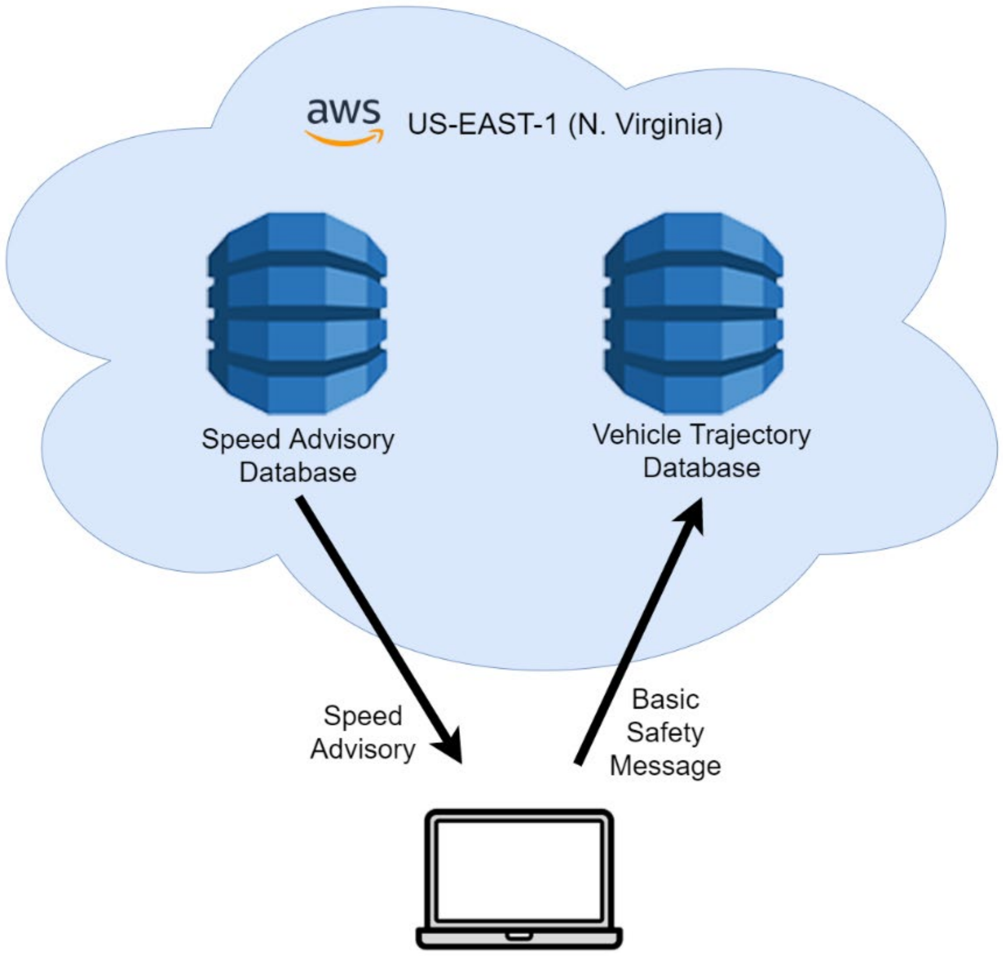

Figure 5.3: Simulation of a CV communicating to the cloud

▪ *Evaluation scenarios*

The author experimented with three-hour increments for 24 hours in one day to estimate the communication delay for uploading and downloading of data. For each experiment, the author collected 1000 samples. The author estimated the 95th percentile of both upload and download delays, which is the threshold for QoS guarantees [25][26].

▪ *Evaluation results*

Figure 5.4 and Figure 5.5 presents the results of each experiment. The maximum upload and download delays were both 100 milliseconds. The author observed that the peak hours for uploading BSMs were from 9:00 AM to 12:00 PM and 06:00 PM to 09:00 PM EDT, and the peak hours for downloading speed advisory were from 3:00 AM to 12:00 PM and 06:00 PM to 09:00 PM EDT. Using the serverless architecture, the author found that the maximum RTT for both upload and download data was 200 milliseconds.

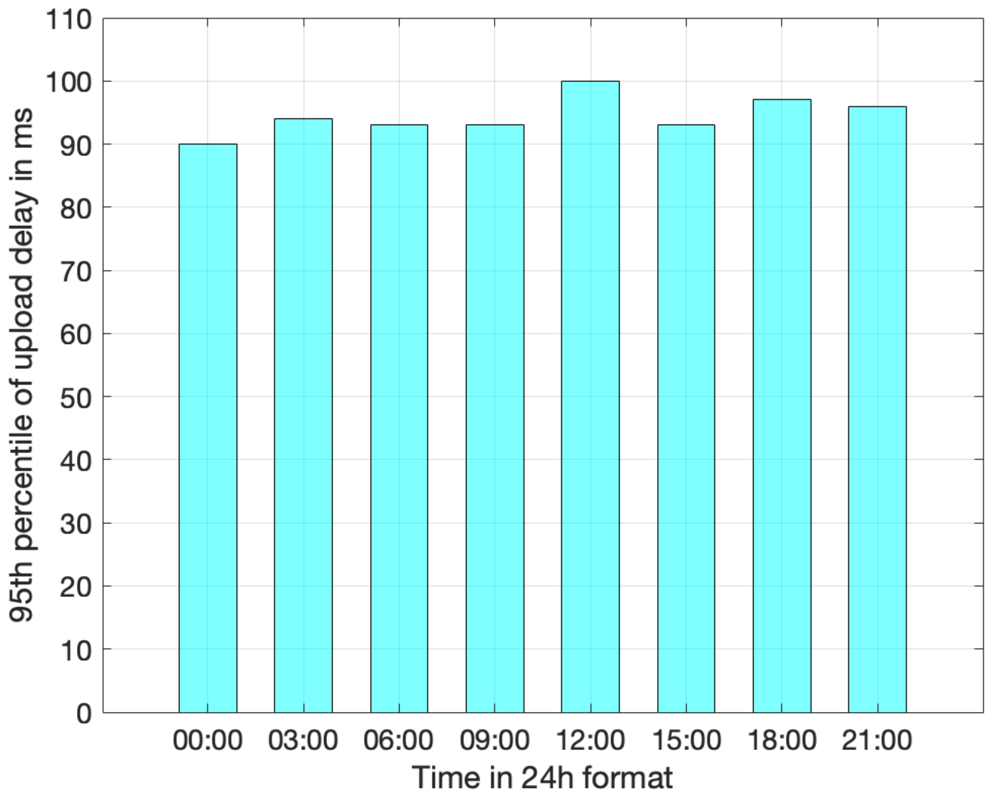

Figure 5.4: 95th percentile of upload delay in 24 hours

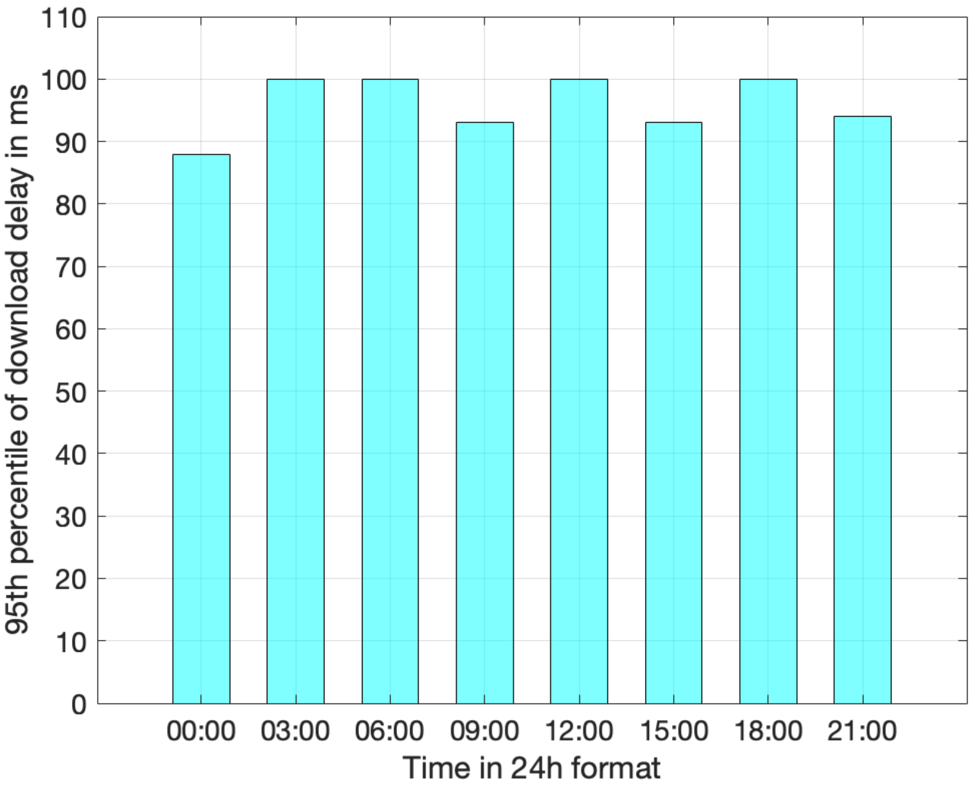

Figure 5.5: 95th percentile of download delay in 24 hours

■ **Field Experiment**

Although the analysis in the previous section shows an RTT under 200 milliseconds for a CV to communicate with the cloud, these experiments were conducted from a fixed physical location. It is necessary to evaluate variations in communication delays with a moving CV. The author performed experiments by uploading BSMs to the Vehicle Trajectory Database and downloading speed advisories from the Speed Advisory Database through LTE to observe the communication delay from a CV traveling on the road.

▪ *Experimental setup*

The author conducted the experiments by driving a vehicle on a fixed route located along Perimeter Road, Clemson, SC, where the LTE signal is available. The speed of the vehicle was 35 mph during the experiment. Figure 5.6 shows a macroscopic view of a CV connected to cloud services through roadside infrastructure.

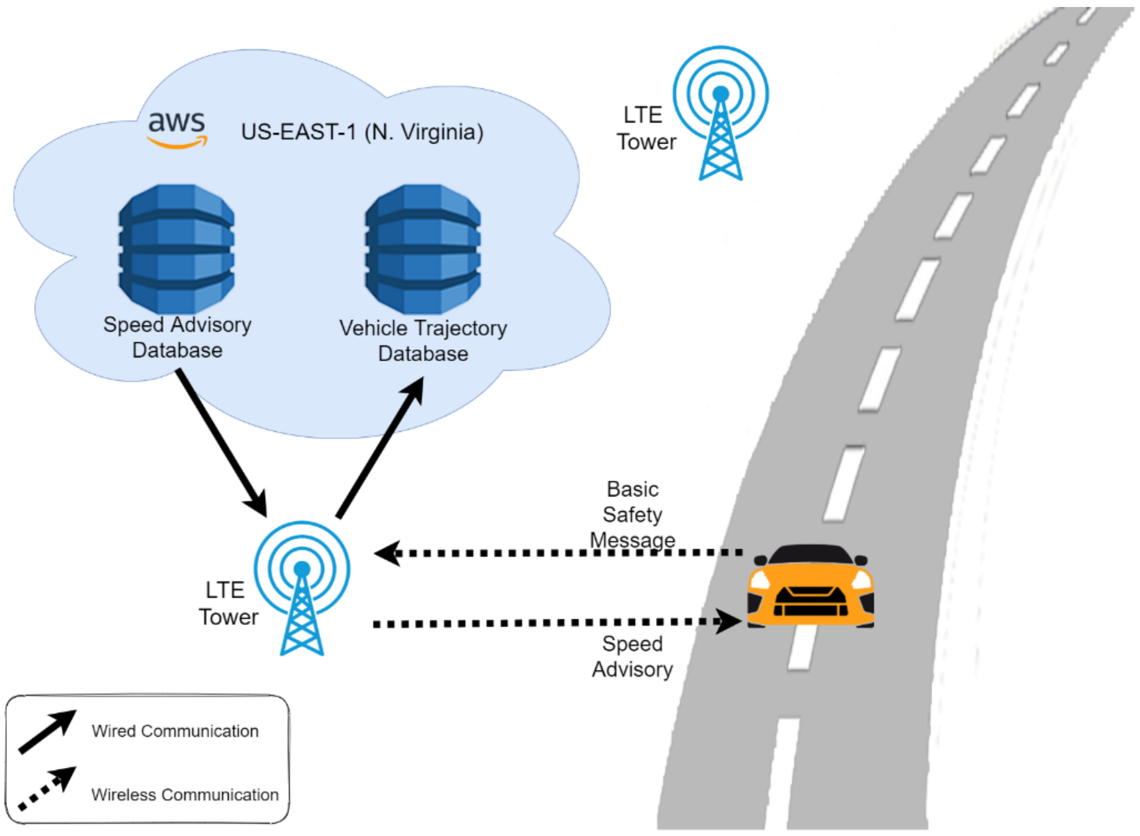

Figure 5.6: Field Experiment Setup

▪ *Evaluation scenarios*

Based on the results in the previous section, the author considered 10:00 AM as the peak hour and 2:00 PM EDT as the off-peak hour for uploading and downloading BSMs. One thousand samples were collected for each experiment.

▪ *Evaluation results*

As shown in Figure 5.7, a moving CV needed a total of 92 milliseconds to upload BSMs and download speed advisories within the 95th percentile. This is well within the acceptable maximum delay of 200 milliseconds, with an average delay of 150 milliseconds RTT. This supports the feasibility of a real-world COSACC implementation in a connected vehicle environment [27].

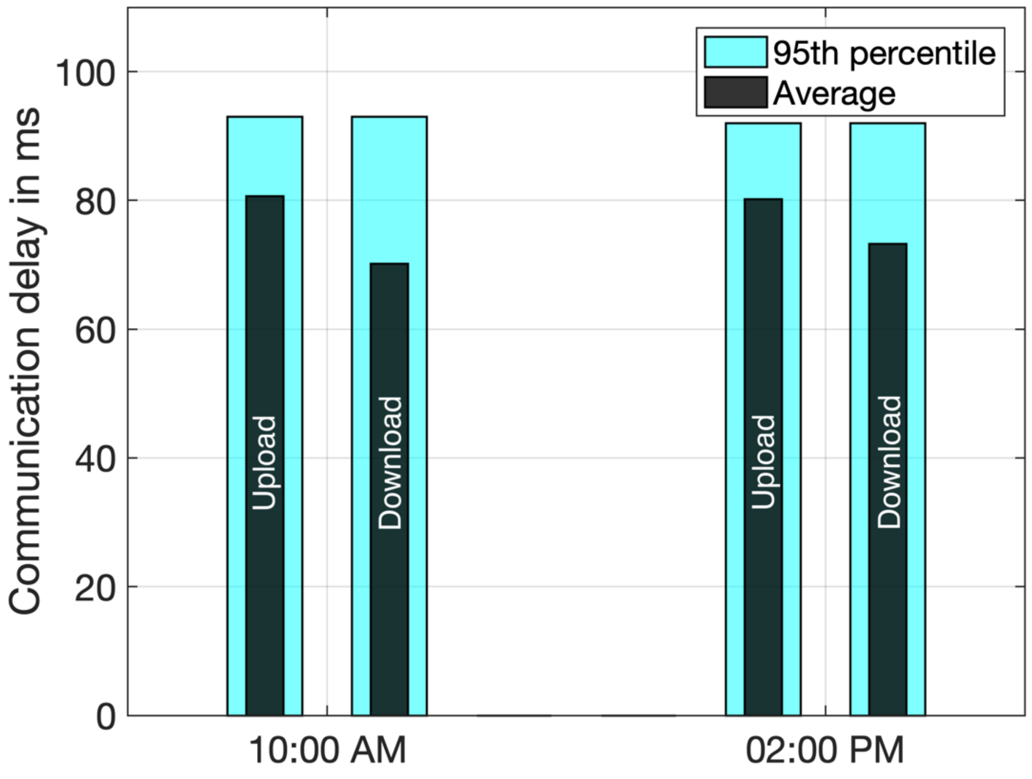

Figure 5.7: Average and 95th percentile communication delay in the field test.

CHAPTER FIVE

CLOUD-BASED SPEED ADVISORY APPLICATION

In this section, the author presents more details about the optimization-based speed advisory application, COSACC, which runs in the AWS cloud. It generates a speed advisory for platoons of CVs to minimize the stopped delay at a signalized intersection. COSACC consists of two parts: (i) vehicle platoon identification and (ii) an optimization-based speed advisory algorithm. Table I presents all the symbols that are required for the application.

TABLE 2
Notations Used in Speed Advisory Algorithm

Symbol	Meaning
S_i^t	Current speed of the i^{th} C
S_{adv}^t	Speed advisory at time t
S_{max}	Maximum speed, same as the speed limit
$t_{S_{adv}}^i$	Time required by the i^{th} CV to reach the intersection after achieving S_{adv}^t
$t_{i,S_{adv}}$	Total time required by the i^{th} CV to reach the intersection from its current state by accelerating to S_{adv}^t and then continuing at S_{adv}^t until it reaches the intersection
$t_{S_{max}}^i$	Time required by i^{th} CV to reach intersection after achieving S_{max}
$t_{i,S_{max}}$	Total time required by the i^{th} CV to reach an intersection from its current state by accelerating to S_{max} and then continuing at S_{max} till it reaches the intersection
t_{remain}	Remaining time of the current green interval
t_{avail}	Available time to pass an intersection
$t_{g,o}$	(Minimum) Green interval of other phases
t_Y	Yellow interval
t_{AR}	All red interval

l_{acc}	Distance covered by a CV while accelerating to a certain speed
$l_{s_{adv}}$	Distance covered by a CV while traveling at the advised speed
$l_{s_{max}}$	Distance covered by a CV while traveling at the maximum speed, S_{max}
$l_{1^{st}\ vehicle}$	Distance from the 1st CV in a platoon to the target signal
d_i^t	Delay of the i^{th} CV
d_{Total}^t	The total delay of all observed CVs
a_{Acc}	Acceleration rate

▪ **Vehicle Platoon Identification**

The author formed a platoon based upon a gap between two successive CVs. If the distance between two successive CVs was less than or equal to 50 meters, then they were considered to be within the same platoon [28]. Moreover, to be identified as a platoon of n CVs, the last (n^{th}) CV of the platoon must have met the following criterion:

$$\min t_{int}^n \leq t_{avail} \tag{1}$$

To estimate the minimum of t_{int}^n, the author considered the total time required by the n^{th} CV to accelerate from its current state to $S_{\max}$ and then continue to operate at $S_{\max}$ until it reaches the intersection, which is given by the following equation,

$$t_{int}^n = t_{Acc}^n + t_{S_{\max}}^n \tag{2}$$

This study considers 100% CV penetration. There are two cases based on the current phase of the signal that CVs are approaching; case I: the platoon can pass the signal within the current green interval, and case II: the platoon can pass the signal in the next green interval. For case I, available time to reach the intersection before the signal turns red is,

$$t_{avail} = t_{remain} \tag{3}$$

and for case II, available time is an accumulation of other intervals until the next cycle, i.e.,

$$t_{avail} = t_{remain} + \sum t_{g.o} + \sum t_Y + \sum t_{AR} \tag{4}$$

For the CVs in case I, the speed advisory tries to assist vehicles across the intersection as fast as possible. For case II, the application first splits CVs into platoons using the gap between any two successive CVs loaded from the vehicle trajectory database. Then, the application provides each platoon with an optimal solution (i.e., speed advisory) based on the objective function and constraints.

▪ **Optimization-based Speed Advisory Model**

To address the objective function of the speed advisory optimization, this study considers the total delay of a platoon of CVs when the platoon travels from an origin to a destination. In this context, "delay" is considered as the additional time required by each CV of the platoon to reach the intersection using the advised speed (S_{adv}^t) compared to the shortest possible time to reach the intersection using the maximum speed (i.e., speed limit), which is given by the following equation,

$$\min_{S_{adv}} d_{Total}^t(S_{adv}) = \sum_{i=1}^{n} d_i^t \tag{5}$$

Where, $d_i^t = t_{i,s_{adv}} - t_{i,s_{\max}}$ (6)

Both $t_{i,s_{adv}}$ and $t_{i,s_{\max}}$ consist of two periods: 1) acceleration period: the time required to accelerate from the CV's current speed S_i^t to S_{adv}^t or $S_{\max}$, and 2) constant speed period: the time required to reach the intersection at a constant speed after achieving S^t_{adv} or $S_{\max}$. Therefore, to determine $t_{i,s_{adv}}$ and $t_{i,s_{\max}}$, we must determine these two time periods. The first step is to determine t^i_{Acc}. To accelerate from S^t_i to a target speed S_{tar} (which can take the value of either S^t_{adv} or $S_{\max}$ here), the required time is given by

$$t^i_{Acc} = \frac{S_{tar} - S^t_i}{a_{Acc}} \tag{7}$$

The next step is to determine the distance covered during the acceleration period. Distance covered while accelerating from S^t_i to S^t_{adv} is calculated as

$$l^i_{Acc} = \frac{{S_{tar}}^2 - {S^t_i}^2}{2a_{Acc}} \tag{8}$$

To determine $t^i_{S_{adv}}$ or $t^i_{S_{\max}}$, the distance to be covered during a constant speed period ($l^i_{S_{adv}}$ or $l^i_{S_{\max}}$) needs to be determined. It can be obtained as follows:

$$l^i_{S_{adv}} = l^t_i - l^i_{Acc} = l^t_i - \frac{(S^t_{adv})^2 - {S^t_i}^2}{2a_{Acc}} \tag{11}$$

$$l^i_{S_{\max}} = l^t_i - l^i_{Acc} = l^t_i - \frac{S^2_{\max} - {S^t_i}^2}{2a_{Acc}} \tag{12}$$

Now, $t^i_{S_{adv}}$ or $t^i_{S_{\max}}$ can be determined as follows:

$$t^i_{S_{adv}} = \frac{l^i_{S_{adv}}}{S^t_{adv}} = \frac{1}{S^t_{adv}}\left[l^t_i - \frac{(S^t_{adv})^2 - {S^t_i}^2}{2a_{Acc}}\right] \tag{13}$$

$$t^i_{S_{\max}} = \frac{l^i_{S_{\max}}}{S_{\max}} = \frac{1}{S_{\max}}\left[l^t_i - \frac{S^2_{\max} - {S^t_i}^2}{2a_{Acc}}\right] \tag{14}$$

Therefore, the delay can be formulated as follows:

$$d_i^t = \left(t_{Acc}^i + t_{S_{adv}}^i\right)_{for\ S_{adv}} - \left(t_{Acc}^i + t_{S_{\max}}^i\right)_{for\ S_{\max}} \tag{15}$$

Substituting the derived terms into Eq. (15) leads to the equation

$$d_i^t = -\frac{S_{adv}^t - S_{\max}}{2a_{Acc}} + \left(l_i^t + \frac{{S_i^t}^2}{2a_{Acc}}\right)\left[\frac{1}{S_{adv}^t} - \frac{1}{S_{\max}}\right] \tag{16}$$

To optimize the speed advisory, this study considered two constraints: 1) constraint 1 ensures that the advised speed does not exceed the speed limit of the road, and 2) constraint 2 ensures that the signal will be green when the first CV of a platoon reaches the intersection. The constraints are formulated as follows:

Constraint 1: $S_{adv}^t \leq S_{max}$ (17)

Constraint 2: $S_{adv}^t \leq \frac{l_{1^{st}\ vehicle}}{t_{avail}}$ (18)

For case I CVs, only constraint 1 is applicable, whereas, for case II platoons, both constraints (1 and 2) are applicable.

CHAPTER SIX

CASE STUDY

The author conducted a case study by developing a cloud-in-the-loop simulation platform to evaluate the feasibility of COSACC at a system level.

▪ Cloud-in-the-loop Simulation

This study used SUMO, a microscopic traffic simulator, in which the author simulated a roadway section including traffic signals and CVs [29]. AWS was integrated with SUMO to develop a cloud-in-the-loop testbed to evaluate COSACC. SUMO TraCI [30] is a Python-based interface compatible with SUMO to extract BSMs (e.g., location and motion of CVs) and SPaT messages (e.g., current interval, remaining green time) from CVs and traffic signals, respectively, and transfer them to AWS services. As described in Chapter three, when AWS Lambda is triggered by the Kinesis Data Stream, it first receives BSMs from the vehicle trajectory database and the historical database to compute the distance from each CV to the target traffic signal in real-time. Current states of the CVs are then transferred as updates to its historical database. Each CV is assigned two possible cases based on their distances from the target signal and the available time, and are split into platoons based on the gap information described in Chapter five. The output of the application is speed advisories that are stored in the speed advisory database.

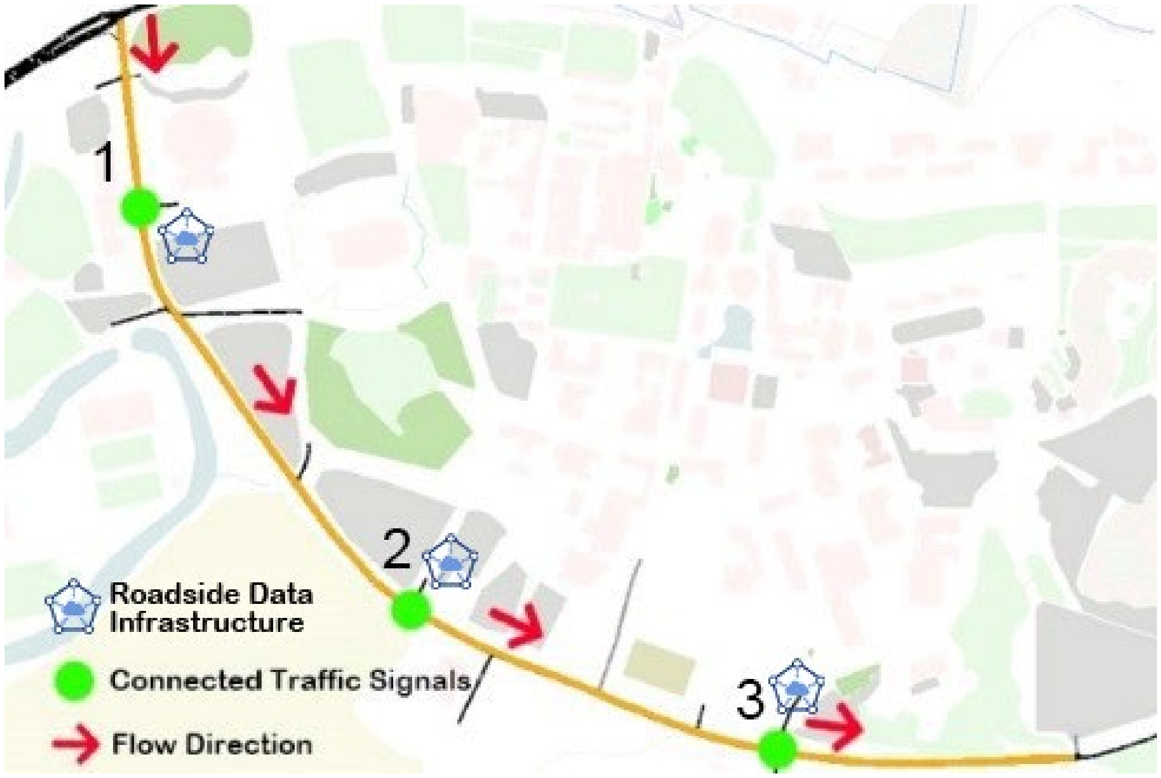

Figure 6.1: Route Configuration.

The author simulated a roadway section including three connected traffic signals along Perimeter Road in Clemson, SC. All CVs followed a speed limit of 35 mph (16m/s). In Figure 6.1, the simulated roadway is in orange and is a 1.5-mile-long 4-lane highway (2 lanes in each direction). The solid green circles indicate connected traffic signals. Traffic signals 1 and 2 maintain 42 seconds green with 3 seconds yellow and all-red intervals for each cycle. Signal 3 has 34 seconds on green for each phase and 5 seconds for yellow and all red. Four cases with different numbers of CVs on the roadway, i.e., 50, 100, 150, and 200 CVs were established. The red arrow represents the traffic flow direction. The author also calculated the number of CVs necessary to simulate traffic at the two-lane roadway capacity of 1900 passenger cars per hour per lane and generated CVs equal to (200 CVs generated in a 15-minute period) or under this rate [31].

For each case, the author evaluated two scenarios in the simulation. The first scenario represents the baseline, i.e., no speed advisory application is implemented. In the second scenario, the author deployed COSACC to evaluate the feasibility of the real-time application through LTE wireless communication.

▪ Evaluation Results and Discussion

To evaluate the performance of COSACC at a system level, this study compared the average stopped delay and the average fuel consumption. The fuel consumption was determined by SUMO's default fuel consumption model. Figure 6.2 shows a significant reduction in stopped delay with COSACC. CVs with speed advisories experienced at most 98% less waiting time or in a traffic queue at the signalized intersections of a corridor compared to the baseline scenario (i.e., no speed advisory) (50 vehicles). COSACC also resulted in less fuel consumption (Figure 6.3). CVs that used COSACC experienced up to 12.7% less fuel consumption compared to the baseline scenario (100 vehicles).

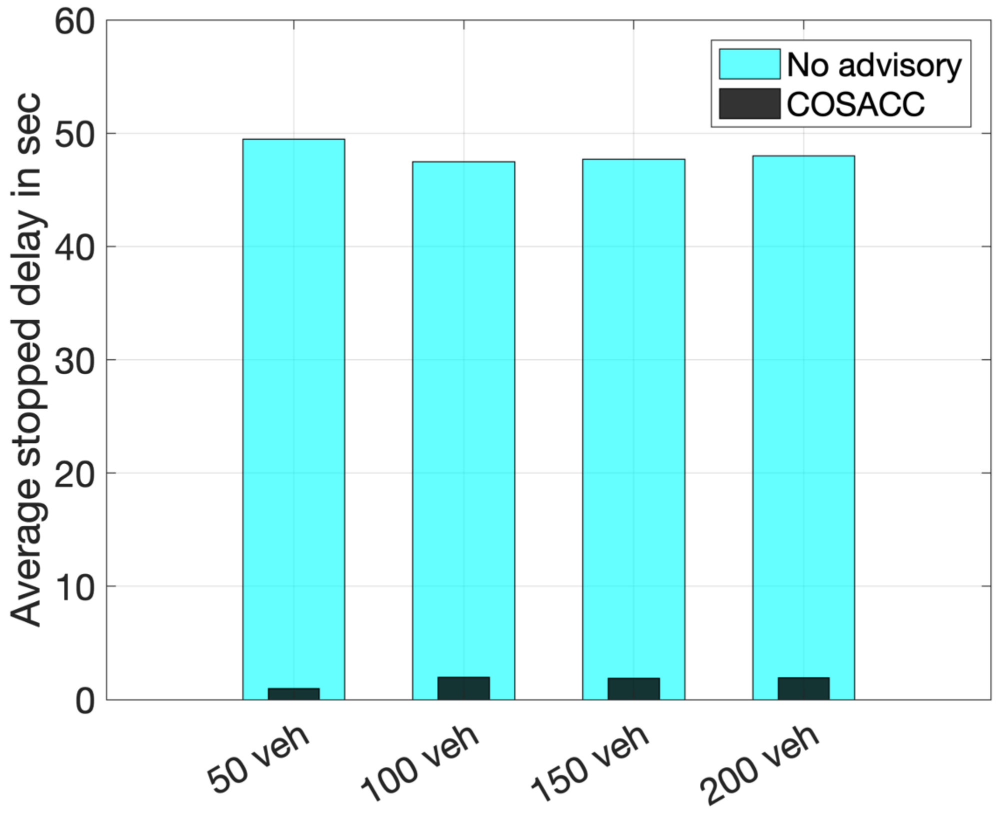

Figure 6.2: Stopped delay

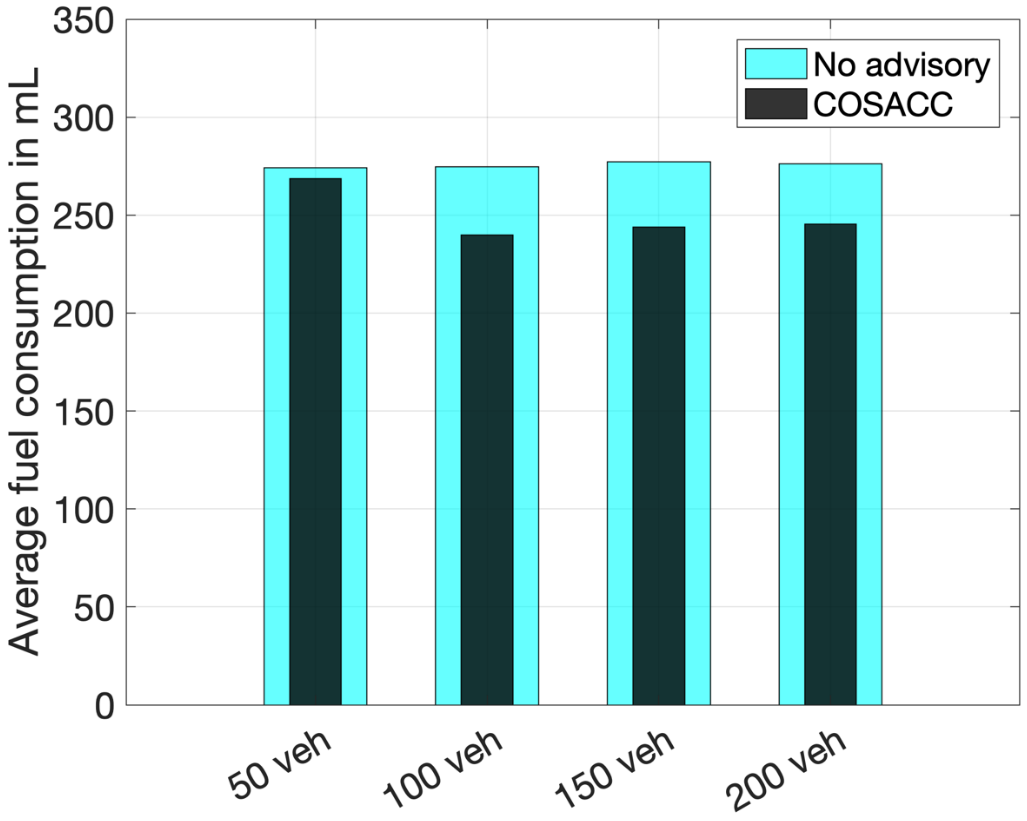

Figure 6.3: Fuel consumption.

Table 3 provides processing time in the cloud and communication delay for each step for the cloud-in-the-loop simulation in all four cases. A communication delay includes upload latency for sending BSMs and SPaT messages to the cloud and download latency of speed advisories from the cloud. As observed from Table 3, the upload delays and download delays per vehicle are under 85 ms on average, and processing time in the cloud is 277 ms on average. This leads to an end-to-end delay of 443 ms on average, which meets the requirement of a real-time traffic mobility application (i.e., a maximum delay of 1000 ms) [24].

TABLE 3
AVERAGE COMMUNICATION DELAY AND PROCESSING TIME

	50 veh.	**100 veh.**	**150 veh.**	**200 veh.**
Upload latency (ms)	80	85	85	84
Download latency (ms)	82	84	84	81
Processing latency (ms)	258	279	281	292
End-to-end Delay (ms)	420	448	450	457

CHAPTER SEVEN

CONCLUSIONS AND RECOMMENDATIONS

▪ Conclusions

In the coming years, connected vehicles will be typical in the transportation system. CVs will contribute to improved mobility and safety, as well as reduced air pollution and energy consumption. However, significant backend computing infrastructure is needed to support CV processing and other related data to support CV applications. Generally, state departments of transportation (DOTs) deploy transportation applications based on servers in their traffic management centers, which requires significant investments in computing and human resources. This study supports the use of cloud infrastructure to meet the dynamic computing needs of CV applications and at lower total costs than traditional infrastructure.

This study confirmed that developing a CV application using a serverless cloud architecture can achieve the same or better results as a server-based cloud architecture. Moreover, a serverless cloud architecture can be more cost-effective and provide even more scalability and flexibility for CV applications.

The study also showed that COSACC improves operational efficiency and fuel consumption for connected vehicles by reducing stopped delay at signalized intersections up to 98% and fuel consumption up to 12.7% compared to the baseline scenario (i.e., no speed advisory). Additionally, field experiments showed that the maximum RTT was under 200 milliseconds and the average RTT was under 100 milliseconds for cloud access from

CVs. The results prove that COSACC is implementable in the real world for CV mobility applications with a serverless architecture.

▪ Recommendations

The following recommendations are made based on this research:

1. Future research should evaluate the relative performance of the serverless and server-based cloud architecture for CV applications in a real-world testbed with different penetration levels of CVs. Evaluation parameters will include cost, delay, and reliability associated with different CV applications.

2. Future research should compare different commercial cloud services, such as AWS, GCP, and Microsoft Azure, for supporting different CV applications in the real world.

3. Cloud services that support CV applications must be secure. Otherwise, real-time CV applications could be compromised. Future research should identify cyber-security risks associated with CV applications using commercial cloud services and possible countermeasures. Critical security elements, which include confidentiality, integrity, availability, authentication, accountability, and privacy, must be evaluated for cloud services accessed by CVs.

4. Although cloud infrastructure supports real-time CV applications, it is not recommended for safety applications (e.g., collision avoidance) in the cloud. Currently, observed RTT exceeds the low communication latency requirements for safety applications.

5. The availability of on-demand computing services and the reliability of all-cloud services make it suitable for developing applications with an increasing number of CVs. Future research should focus on the scalability of cloud computing with increasing penetration levels of CVs on roadways.

APPENDICES

Appendix A

Socket Program

- **Socket Program in a server**

Server.cpp

```
#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>


int main()

{

   int server_sockfd; //server socket

   int client_sockfd; // client socket


   int len;

   int speed;


   struct sockaddr_in my_addr; //server address structure

   struct sockaddr_in remote_addr; //client address structure
```

```
int sin_size;
char buff[BUFSIZ]; //buffer
char *buffsend = (char *) "Speed Advisory is XX mph";

memset(&my_addr, 0, sizeof(my_addr)); //Initialization
my_addr.sin_family = AF_INET; //IP communication
my_adddr.sin_addr.s_addr = INADDR_ANY;
my_addr.sin_port = htons(8000); //Port number

cout << "Server Socket program start" << endl;
if((server_sockfd=socket(PF_INET,SOCK_STREAM,0))<0)
{
   perror("Socket Error");
   return 1;
}

if(bind(server_sockfd, (stuct sockaddr *)&my_addr, sizeof(struct sockaddr))<0)
{
   perror("Bind Error");
   return 1;
}

if(listen(server_sockfd,5)<0)
{
        perror("Listen Error");
        return 1;
    }

    sin_size = sizeof(struct sockaddr_in);
    int counter = 0;

    while(1)
    {
        //Wait for client request reach
        if((client_sockfd=accept(server_sockfd, (struct sockaddr *)&remote_addr,
(socklen_t*)&sin_size))<0)
        {
            perror("Accept Error");
            continue;
        }
        cout << "Accept Client " << inet_nota(remote_addr.sin_addr) << endl;
        len = send(client_sockfd,buffsend,strlen(buffsend)+1,0);
        while((len=recv(client_sockfd,buff,BUFSIZ,0))>0)
        {
            buff[len] = '\0';
            cout << "Test No." << counter << " " << buff << endl;
            if(send(client_sockfd,buff,len,0)<0)
            {
                perror("Write Error");
                continue;
            }
        }
        counter++;
        close(client_sockfd);
    }

    close(server_sockfd);
    return 0;
}
```

▪ **Socket Program in a client**

Client.cpp

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS


#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>
```

```
#include <WinSock2.h>

#include <windows.h>

#include <vector>

#include <fstream>

#include <sstream>

#include <ctime>

using namespace std;



#pragma comment(lib, "libws2_32.a")



string int2str(int &val)

{

   string s;

   stringstream ss(s);

   ss << val;

   return ss.str();

}
```

```
int main()

{

   time_t now = time(0);

   tm *ltm = localtime(&now);



   vector<int> RTT;

   WSADATA wsaData;

   char buff[1024];

   bool isSend = false;



   DWORD t_start, t_end;

   char *buffsend = "BSM: 10.00000, 12.00000, 8.56957, 5.12121, 6.87777,
9.68547, 0, 1";



   for(int k = 0; k < 12; k++)

   {
```

```
for(int i = 0; i < 201; i++)

{

   memset(buff,0,sizeof(buff));



   if(WSAStartup(MAKEWORD(2,2),&wsaData) != 0)

   {

      cout << "Failed to load WinSock" << endl;

      return 1;

   }



   SOCKADDR_IN addrsrv;

   addrsrv.sin_family = AF_INET;

   addrsrv.sin_port = htons(8000);

   addrsrv.sin_addr.S_un.S_addr = inet_addr("3.87.202.164");



   SOCKET sockClient = socket(AF_INET, SOCK_STREAM, 0);

   if(SOCKET_ERROR == sockClient)
```

```
        {
            cout << "Socket Error: " << WSAGetLastError() << endl;
            return 1;
        }
        else
        {
            cout << "Socket Initialized" << endl;
        }

        if(connect(sockClient, (struct sockaddr*)&addrsrv, sizeof(addrsrv)) ==
INVALID_SOCKET)
        {
            cout << "Connect Failed: " << WSAGetLastError() << endl;
            continue;
        }
        //send(sockClient, buffsend, strlen(buffsend)+100, 0);
        if((GetTickCount() - t_start) > 10000 && (GetTickCount() - t_start) <
11000)
        {
          cout << "No. " << i << " failed" << endl;
          t_start = GetTickCount();
          send(sockClient, buffsend, strlen(buffsend)+500, 0);
          closesocket(sockClient);
          WSACleanup();
        }else{
          recv(sockClient, buff, sizeof(buff), 0);
          cout << buff << endl;
          t_end = GetTickCount();
          cout << "No. " << i << " RTT is " << t_end - t_start << endl;
          RTT.push_back((int)(t_end-t_start));

          t_start = GetTickCount();
          send(sockClient, buffsend, strlen(buffsend)+500, 0);
```

```
            closesocket(sockClient);

            WSACleanup();

        }

    }



    now = time(0);

    ltm = localtime(&now);



    cout << "Time:"<< ltm->tm_hour << ":" << ltm->tm_min << endl;

    if(k < 11)

    {

        Sleep(1800000);

    }

}

ofstream outFile;

outFile.open("RTT.csv", ios::out);
```

```
    cout << endl << "Round Trip Time Summary:" << endl;

    for(int j = 1; j < RTT.size(); j++)

    {

        cout << "No " << j << " RTT is " << RTT[j] << endl;

        outFile << RTT[j] << endl;

    }



    outFile.close();

    return 0;

}
```

Appendix B

Connected Vehicle communication program

Vehicle.py

```
import boto3

import json

import time

from decimal import *

import csv


def Vehicle_State_upload(Table, ID, pos_x, pos_y, speed, gap):

   Table.put_item(

   Item={

       'vehicle':ID,

       'pos_x':Decimal(pos_x),

       'pos_y':Decimal(pos_y),

       'speed':Decimal(speed),

       'gap':Decimal(gap)

       }

   )

dynamodb = boto3.resource('dynamodb')

state_table = dynamodb.Table('Speed_file')

advisory_table = dynamodb.Table('Speed_advisory')
```

```
ULs = []

DLs = []


#Record upload time

For i in range(1000):

    start_time = time.time()*1000

    Vehicle_State_upload(state_table, '0', '375.28', '442.02', '16.0', '10000.0')

    process_time = int(time.time()*1000 - start_time)

    ULs.append(process_time)

    print('Trip Time: ',process_time, 'ms\n')


print(ULs)


with open('UL.csv', 'w', newline='') as csvfile:

    writer = csv.writer(csvfile)

    writer.writerow(ULs)


#Record download time

For i in range(1000):

    start_time = time.time()*1000

    response = advisory_table.get_item(Key={'vehicle':'0'})

    Speed = float(response['Item']['advisory'])
    process_time = int(time.time()*1000 - start_time)
    DLs.append(process_time)
    print('Trip Time: ',process_time, 'ms\n')

print(DLs)


with open('DL.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(DLs)

print('done')
```

Appendix C

Speed Advisory in Lambda Function

Lambda_function.py

```
import base64

import json

import boto3

import time

import math

from decimal import *

from threading import Thread


dynamodb = boto3.resource('dynamodb')


TLS_position_x = 745.45

TLS_position_y = 1118.61


Distance_Table = {}

Speed_Table = {}

Gap_Table = {}


input_table = dynamodb.Table('Speed_file')
```

```
output_table = dynamodb.Table('Speed_advisory')

Historical_Table = dynamodb.Table('Historical_State_Table_113850262')

queue = sqs.get_queue_by_name(QueueName='TLS_ACK_113850262')

input = ''


def get_Distance(table,ID):

    try:

        response = table.get_item(Key={'vehicle': ID})

    except ClientError as e:

        print(e.response['Error']['Message'])

    else:

        return float(response['Item']['distance'])


def Historical_State_Update(Table, ID, distance):

    Table.put_item(

        Item={

            'vehicle':ID,

            'distance':Decimal(str(distance))

             }

      )


def Assign_Vehicle(pos_x,pos_y,phase,remain,ID,speed,gap):
    X = TLS_position_x - float(pos_x)

    Y = TLS_position_y - float(pos_y)

    distance = round(math.sqrt(X*X+Y*Y),2)


    if phase == 0:

        if distance <= get_Distance(Historical_Table,ID) and distance > remain * 16.0:

            Distance_Table[ID] = distance

            Speed_Table[ID] = speed

            Gap_Table[ID] = gap

    else:

        #Speed Advisories only give to vehicles approaching TLS

        if distance <= get_Distance(Historical_Table,ID):

            Distance_Table[ID] = distance

            Speed_Table[ID] = speed

            Gap_Table[ID] = gap

    Historical_State_Update(Historical_Table,ID,distance)



def compute_advisory(platoon,Distance_Table,Speed_Table,delay_time):

    Platoon_Distance = []

    Platoon_Speed = []
    for vehicle in platoon:
        Platoon_Distance.append(Distance_Table[vehicle])
        Platoon_Speed.append(Speed_Table[vehicle])

    adv = round(min(Platoon_Distance) / delay_time,1)
    if adv > 16.0:
        adv = 16.0

    output_table.put_item(
        Item={
            'vehicle': platoon[0],
            'advisory':Decimal(str(adv))
             }
    )

def scan_and_process_input_table(current_phase,remain):
    thread_list = []
    total_segments = 8 # number of parallel scans
    for i in range(total_segments):
        # Instantiate and store the thread
        thread = Thread(target=parallel_scan_and_process_input_table,
args=(i,total_segments,current_phase,remain))
```

```
        thread_list.append(thread)
    # Start threads
    for thread in thread_list:
        thread.start()
    # Block main thread until all threads are finished
    for thread in thread_list:
        thread.join()


def parallel_scan_and_process_input_table(segment, total_segments, current_phase,
remain):
    threads = []
    thread_number = 0
    #print("Total segments = "+str(total_segments)+" segment "+str(segment))
    vehicles = input_table.scan(
        Segment=segment,
        TotalSegments=total_segments,
        ConsistentRead=True
        )
    print('Looking at segment ' + str(segment) + ' of '+ str(total_segments) + "
"+str(len(vehicles['Items']))+" vehicles\n")

    for i in vehicles['Items']:
        thread = Thread(target=Assign_Vehicle,
args=(i['pos_x'],i['pos_y'],current_phase,remain,i['vehicle'],i['speed'],i['gap']))
        threads.append(thread)
        thread_number += 1
        if thread_number > 8:
            for thread in threads:
                thread.start()
            thread_number = 0
            threads.clear()

    for thread in threads:
        thread.start()




def lambda_handler(event, context):
    payload = ''
    current_phase = 0

    Platoons = []
    Platoon_index = -1
```

```
threads = []
thread_number = 0

test_records =["Test"]

# TODO implement
#for record in event['Records']:
for record in test_records:
    input = str(base64.b64decode(record['kinesis']['data']))[2:-1]

    if ',' == input[1]:
        decoded_message = input.split(',')
        current_phase = int(decoded_message[0])
        remain = int(decoded_message[1])
    else:
        payload = input

    if current_phase == 0:
        delay_time = remain + 48
    elif current_phase == 1:
        delay_time = remain + 45
    elif current_phase == 2:
            delay_time = remain + 3
        else:
            delay_time = remain

        start_time = int(round(time.time() * 1000))

        vehicles = input_table.scan()
        for i in vehicles['Items']:
            if int(i['vehicle']) < 50:
                threads.append(Thread(target=Assign_Vehicle,
args=(i['pos_x'],i['pos_y'],current_phase,remain,i['vehicle'],i['speed'],i['gap'])))
                thread_number += 1
                if thread_number > 12:
                    for thread in threads:
                        thread.start()
                    thread_number = 0
                    threads.clear()
        for thread in threads:
            thread.start()

        end_time = int(round(time.time() * 1000))
        print('Speed file processing time is: ', end_time-start_time)
```

```
    start_time = int(round(time.time() * 1000))
    if Distance_Table:
        Sorted_Distance = sorted(Distance_Table.items(), key=lambda x: x[1])

        for i in range(len(Sorted_Distance)):
            if float(Gap_Table[Sorted_Distance[i][0]]) > 50.0 or i == 0:
                Platoon_index += 1
                Platoons.append([])
                Platoons[Platoon_index].append(Sorted_Distance[i][0])
            else:
                Platoons[Platoon_index].append(Sorted_Distance[i][0])

        threads.clear()
        thread_number = 0
        for platoon in Platoons:

threads.append(Thread(target=compute_advisory,args=(platoon,Distance_Table,Speed_T
able,delay_time,)))
            thread_number += 1
        for thread in threads:
            thread.start()
    end_time = int(round(time.time() * 1000))

    payload += str(end_time-start_time)

    response = queue.send_message(MessageBody=payload)



    print(payload)

    print('Processing time is: ', end_time-start_time)


return 'successfully processed {} records.'.format(len(test_records))
```

Appendix D

Simulation Program

■ **Baseline Scenario**

Baseline.py

```
#!/usr/bin/env python


import os

import sys

import optparse

import time

import json

import math


vechicleID = []

trafficsignalID = []

# we need to import some python modules from the $SUMO_HOME/tools directory

if 'SUMO_HOME' in os.environ:

    tools = os.path.join(os.environ['SUMO_HOME'], 'tools')

    sys.path.append(tools)

else:

    sys.exit("please declare environment variable 'SUMO_HOME'")
```

```
from sumolib import checkBinary  # Checks for the binary in environ vars
import traci


def get_options():
    opt_parser = optparse.OptionParser()
    opt_parser.add_option("--nogui", action="store_true",
                         default=False, help="run the commandline version of sumo")
    options, args = opt_parser.parse_args()
    return options


# contains TraCI control loop
def run():
    step = 0


    Stopping_Steps = [-1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                     -1,-1,-1,-1,-1,
                -1,-1,-1,-1,-1,
                -1,-1,-1,-1,-1,
                -1,-1,-1,-1,-1
               ]

fuel_consumption = [ 0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0,
                    0.0,0.0,0.0,0.0,0.0
                  ]

while traci.simulation.getMinExpectedNumber() > 0 and step < 3600:
    traci.simulationStep()
    print('Step: ',step)

    #Condition for each vehicle
```

```
        vehicleID = traci.vehicle.getIDList()


        for vehicles in vehicleID:
            traci.vehicle.setSpeed(vehicles, Vehicle_Advisory[vehicles])
            fuel_consumption[int(vehicles)] +=
round(traci.vehicle.getFuelConsumption(vehicles),0)
            if traci.vehicle.getSpeed(vehicles) == 0.0:
                Stopping_Steps[int(vehicles)] += 1


    print(Stopping_Steps)
    print(fuel_consumption)


    traci.close()
    sys.stdout.flush()




# main entry point
if __name__ == "__main__":
    options = get_options()


    # check binary
    if options.nogui:
        sumoBinary = checkBinary('sumo')
    else:
        sumoBinary = checkBinary('sumo-gui')

    print("TRaCI Starts Sumo")
    # traci starts sumo as a subprocess and then this script connects and runs
    traci.start([sumoBinary, "-c", "map.sumo.cfg",
                             "--tripinfo-output", "tripinfo.xml"])
    run()
```

▪ **COSACC Scenario**

COSACC_50.py

```
#!/usr/bin/env python

import os
import sys
import optparse
import time
import boto3
import botocore
import json
import math

from decimal import *

from threading import Thread


vechicleID = []

trafficsignalID = []

RRT = []

MSGS = []

my_stream = 'Speed_Input'

kinesis_client = boto3.client('kinesis',region_name='us-east-1')

sqs = boto3.client('sqs')

queue_url = 'https://sqs.us-east-1.amazonaws.com/233952390740/Speed_Advisory'


record_data = ''

received_data = ''

config = botocore.config.Config(max_pool_connections=100)

dynamodb = boto3.resource('dynamodb',config=config)

Vehicle_State_Table = dynamodb.Table('Speed_file')

Speed_Advisory_Table = dynamodb.Table('Speed_advisory')


# we need to import some python modules from the $SUMO_HOME/tools directory

if 'SUMO_HOME' in os.environ:
    tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
    sys.path.append(tools)
else:
    sys.exit("please declare environment variable 'SUMO_HOME'")


from sumolib import checkBinary  # Checks for the binary in environ vars
import traci


def Vehicle_State_Init(Table, ID):
    Table.put_item(
        Item={
            'vehicle':ID,
            'advisory':Decimal('16.0'),
        }
    )


def Vehicle_State_upload(Table, ID, pos_x, pos_y, speed, gap):
    Table.put_item(
        Item={
            'vehicle':ID,
            'pos_x':Decimal(pos_x),
            'pos_y':Decimal(pos_y),
            'speed':Decimal(speed),
            'gap':Decimal(gap)
        }
    )

def Vehicle_Advisory_Download(DLTable, ID, Table):
    response = DLTable.get_item(Key={'vehicle':ID})
    adv = float(response['Item']['advisory'])
    Table[ID] = adv

def kinesis_upload(stream, data):
    response = kinesis_client.put_record(
        StreamName=stream,
        Data=data.encode(),
        PartitionKey='0',
        ExplicitHashKey='0',
        SequenceNumberForOrdering='0'
    )
    return

def sqs_download(url):
```

```
response = sqs.receive_message(
    QueueUrl = url,
    AttributeNames=[
        'All',
    ],
    MessageAttributeNames=[
        'All',
    ],
    MaxNumberOfMessages=1,
    VisibilityTimeout=0,
    WaitTimeSeconds=5,
    ReceiveRequestAttemptId='string'
)
message = response['Messages'][0]
receipt_handle = message['ReceiptHandle']
message_body = message['Body']
resonsea = sqs.delete_message(
    QueueUrl = url,
    ReceiptHandle = receipt_handle
)

return message_body
```

```
def get_options():
    opt_parser = optparse.OptionParser()
    opt_parser.add_option("--nogui", action="store_true",
                          default=False, help="run the commandline version of sumo")
    options, args = opt_parser.parse_args()
    return options


# contains TraCI control loop
def run():
    step = 0
    record_113913026 = ''
    record_113850262 = ''
    record_113915746 = ''

    current_phase_113913026 = 0
    current_phase_113850262 = 0
    current_phase_113915746 = 0

    tls_timer_113913026 = 0.0
    tls_timer_113850262 = 0.0
    tls_timer_113915746 = 0.0
```

```
Stopping_Steps = [-1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1,
                  -1,-1,-1,-1,-1
                 ]

fuel_consumption = [ 0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                     0.0,0.0,0.0,0.0,0.0,
                  0.0,0.0,0.0,0.0,0.0
                 ]

    threads = []
    threads_number = 0

    while traci.simulation.getMinExpectedNumber() > 0 and step < 3600:
        traci.simulationStep()
        print('Step: ',step)

        #Condition for each vehicle
        vehicleID = traci.vehicle.getIDList()

        #Initialization
        threads.clear()
        for vehicle in vehicleID:

threads.append(Thread(target=Vehicle_State_Init,args=(Speed_Advisory_Table,vehicle))
)
            threads_number += 1
            if threads_number > 8:
                for thread in threads:
```

```
            thread.start()
        threads_number = 0
        threads.clear()
threads_number = 0
for thread in threads:
    thread.start()



#Gathering vehicles' states
threads.clear()
for vehicle in vehicleID:

    Vehicle_Advisory = 16.0

    Vehicle_Location = traci.vehicle.getPosition(vehicle)
    Vehicle_Speed = round(traci.vehicle.getSpeed(vehicle),1)

    if traci.vehicle.getLeader(vehicle):
        Vehicle_Gap = round(traci.vehicle.getLeader(vehicle)[1],2)
    else:
        Vehicle_Gap = 10000.0
```

```
            threads.append(Thread(target=Vehicle_State_upload,args=(Vehicle_State_Table,
vehicle, str(round(Vehicle_Location[0],2)), str(round(Vehicle_Location[1],2)),
str(Vehicle_Speed), str(Vehicle_Gap),)))

            threads_number += 1
            if threads_number > 8:
                for thread in threads:
                    thread.start()
                threads.clear()
                threads_number = 0
        for thread in threads:
            thread.start()

        #TLS timer to get remaining green time
        if current_phase_113913026 != traci.trafficlight.getPhase('113913026'):
            current_phase_113913026 = traci.trafficlight.getPhase('113913026')
            tls_timer_113913026 = 0.0
        record_113913026 = str(current_phase_113913026)
        remain_113913026 = traci.trafficlight.getPhaseDuration('113913026') -
tls_timer_113913026
        record_113913026 += ',' + str(int(remain_113913026))
        #TLS timer to get remaining green time
        if current_phase_113850262 != traci.trafficlight.getPhase('113850262'):
            current_phase_113850262 = traci.trafficlight.getPhase('113850262')
            tls_timer_113850262 = 0.0
        record_113850262 = str(current_phase_113850262)
        remain_113850262 = traci.trafficlight.getPhaseDuration('113850262') -
tls_timer_113850262
        record_113850262 += ',' + str(int(remain_113850262))

        #TLS timer to get remaining green time
        if current_phase_113915746 != traci.trafficlight.getPhase('113915746'):
            current_phase_113915746 = traci.trafficlight.getPhase('113915746')
            tls_timer_113915746 = 0.0
        record_113915746 = str(current_phase_113915746)
        remain_113915746 = traci.trafficlight.getPhaseDuration('113915746') -
tls_timer_113915746
        record_113915746 += ',' + str(int(remain_113915746))

        #Advisory by cloud
        kinesis_upload('TLS_State_113913026',record_113913026)
        print('A:',sqs_download('https://sqs.us-east-
1.amazonaws.com/233952390740/TLS_ACK_113913026'))
```

```
    kinesis_upload('TLS_State_113850262',record_113850262)
    print('B:',sqs_download('https://sqs.us-east-
1.amazonaws.com/233952390740/TLS_ACK_113850262'))
    kinesis_upload('TLS_State_113915746',record_113915746)
    print('C:',sqs_download('https://sqs.us-east-
1.amazonaws.com/233952390740/TLS_ACK_113915746'))

    #Gathering Advisories on cloud
    threads.clear()
    threads_number = 0
    for vehicle in vehicleID:
      threads.append(Thread(target=Vehicle_Advisory_Download,
args=(Speed_Advisory_Table,vehicle,Vehicle_Advisory)))
      threads_number += 1
      if threads_number > 8:
        for thread in threads:
          thread.start()
        threads_number = 0
        threads.clear()
    for thread in threads:
      thread.start()
```

```
        for vehicles in vehicleID:
            traci.vehicle.setSpeed(vehicles, Vehicle_Advisory[vehicles])
            fuel_consumption[int(vehicles)] +=
round(traci.vehicle.getFuelConsumption(vehicles),0)
            Travel_Time[int(vehicles)] += 1
            if traci.vehicle.getSpeed(vehicles) == 0.0:
                Stopping_Steps[int(vehicles)] += 1



        step += 1
        tls_timer_113913026 += 1
        tls_timer_113850262 += 1
        tls_timer_113915746 += 1

    print(Stopping_Steps)
    print(fuel_consumption)

    traci.close()
    sys.stdout.flush()


# main entry point
```

```
if __name__ == "__main__":
    options = get_options()


    # check binary
    if options.nogui:
        sumoBinary = checkBinary('sumo')
    else:
        sumoBinary = checkBinary('sumo-gui')


    print("TRaCI Starts Sumo")
    # traci starts sumo as a subprocess and then this script connects and runs
    traci.start([sumoBinary, "-c", "map.sumo.cfg",
                             "--tripinfo-output", "tripinfo.xml"])
    run()
```

REFERENCES

[1] Y.Du, M. Chowdhury, M. Rahman, K. Dey, A. Apon, A. Luckow, and L. Ngo, "A Distributed Message Delivery Infrastructure for Connected Vehicle Technology Applications," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 3, pp. 787–801, 2018.

[2] A.Sarker, H. Shen, M. Rahman, M. Chowdhury, K. Dey, F. Li, Y. Wang, and H. S. Narman, "A Review of Sensing and Communication, Human Factors, and Controller Aspects for Information-Aware Connected and Automated Vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 21, no. 1, pp. 7–29, 2020.

[3] M.Chowdhury, M.Islam, and Z.Khan, "Security of connected and automated vehicles," *Bridge*, vol. 49, no. 3, pp. 46–56, 2019.

[4] Amazon, "AWS GovCloud (US)," 2000. [Online]. Available: https://aws.amazon.com/govcloud-us/?whats-new-ess.sort-by=item.additionalFields.postDateTime&whats-new-ess.sort-order=desc.

[5] Google, "Government & Public Sector Compliance | Google Cloud." [Online]. Available: https://cloud.google.com/security/compliance/government-public-sector.

[6] Microsoft, "Government Cloud Computing: Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/global-infrastructure/government/.

[7] R. A. P.Rajan, "Serverless Architecture - A Revolution in Cloud Computing," *2018 10th Int. Conf. Adv. Comput. ICoAC 2018*, pp. 88–93, 2018.

[8] H.Shafiei, A.Khonsari, and P.Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges and Applications," pp. 1–13, 2019.

[9] J. W.Kim, J. W.Kim, and D. K.Jeon, "A cooperative communication protocol for QoS provisioning in IEEE 802.11p/wave vehicular networks," *Sensors (Switzerland)*, vol. 18, no. 11, pp. 1–19, 2018.

[10] Z.Ning, J.Huang, and X.Wang, "Vehicular fog computing: Enabling real-time traffic management for smart cities," *IEEE Wirel. Commun.*, vol. 26, no. 1, pp. 87–93, 2019.

[11] X.Li, Y.Dang, M.Aazam, X.Peng, T.Chen, and C.Chen, "Energy-Efficient Computation Offloading in Vehicular Edge Cloud Computing," *IEEE Access*, vol. 8, pp. 37632–37644, 2020.

[12] J.JIng, R.Bin, C.Tianyi, J.XIaowen, Z.Tianya, and Y.Zhenxing, "Cloud-based Technology for Connected and Automated Vehicle Higway Systems," 2019.

[13] A.Stevanovic, J.Stevanovic, and C.Kergaye, "Green light optimized speed advisory systems," *Transp. Res. Rec.*, no. 2390, pp. 53–59, 2013.

[14] H.Suzuki and Y.Marumo, "A New Approach to Green Light Optimal Speed Advisory (GLOSA) Systems for High-Density Traffic Flowe," *IEEE Conf. Intell. Transp. Syst. Proceedings, ITSC*, vol. 2018-Novem, pp. 362–367, 2018.

[15] L.Simchon and R.Rabinovici, "Real-Time Implementation of Green Light Optimal Speed Advisory for Electric Vehicles," *Vehicles*, vol. 2, no. 1, pp. 35–54, 2020.

[16] S.Stebbins, J.Kim, M.Hickman, and H. L.Vu, "Combining model predictive intersection control with Green Light Optimal Speed Advisory in a connected vehicle environment," in *Australasian Transport Research Forum (ATRF)*, 2016.

[17] S.Stebbins, M.Hickman, J.Kim, and H. L.Vu, "Characterising Green Light Optimal Speed Advisory trajectories for platoon-based optimisation," *Transp. Res. Part C Emerg. Technol.*, vol. 82, pp. 43–62, 2017.

[18] W.Zhao, D.Ngoduy, S.Shepherd, R.Liu, and M.Papageorgiou, "A platoon based cooperative eco-driving model for mixed automated and human-driven vehicles at a signalised intersection," *Transp. Res. Part C Emerg. Technol.*, vol. 95, no. May, pp. 802–821, 2018.

[19] Microsoft, "Azure Functions Serverless Compute | Microsoft Azure," 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed: 07-Jul-2020].

[20] Google Cloud, "Serverless computing | Google Cloud," 2020. [Online]. Available: https://cloud.google.com/serverless?hl=us. [Accessed: 07-Jul-2020].

[21] A. W.Services, "Developer Guide."

[22] Amazon Web Services, "Amazon DynamoDB Developer Guide," 2011.

[23] AWS, "Amazon Kinesis Data Streams - AWS," *Aws*, 2019.

[24] USDOT, "Southeast Michigan Test Bed 2014 Comcept of Operations," 2014.

[25] D. A.Menascé, "QoS issues in web services," *IEEE Internet Comput.*, vol. 6, no. 6, pp. 72–75, 2002.

[26] R.Garg, H.Saran, R. S.Randhawa, and M.Singh, “A SLA framework for QoS provisioning and dynamic capacity allocation,” *IEEE Int. Work. Qual. Serv. IWQoS*, vol. 2002-Janua, no. c, pp. 129–137, 2002.

[27] A.Ashok, P.Steenkiste, and F.Bai, “Adaptive cloud offloading for vehicular applications,” *IEEE Veh. Netw. Conf. VNC*, vol. 0, 2016.

[28] T.Hardes and C.Sommer, “Dynamic Platoon Formation at Urban Intersections,” *Proc. - Conf. Local Comput. Networks, LCN*, vol. 2019-Octob, pp. 101–104, 2019.

[29] P. A.Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y. Fl¨otter¨od, R. Hilbrich, L. L¨ucken, J. Rummel, P. Wagner, and E. Wießner, “Microscopic Traffic Simulation using SUMO,” *IEEE Conf. Intell. Transp. Syst. Proceedings, ITSC*, vol. 2018-Novem, pp. 2575–2582, 2018.

[30] A.Wegener, M.Piórkowski, M.Raya, H.Hellbrück, S.Fischer, and J. P.Hubaux, “TraCI: An interface for coupling road traffic and network simulators,” *Proc. 11th Commun. Netw. Simul. Symp. CNS’08*, pp. 155–163, 2008.

[31] H. C.Manual, “Chapter 14 Multilane Highways,” no. December, 2010.