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

School of Computer Science and Statistics

Towards the design of a large-scale distributed traffic management system for slot-based driving

Siqi Wei
[redacted]

May 8, 2020

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

This research focused on designing a large-scale distributed traffic management system (TMS). It aims at developing a TMS to support the slot-based driving approach and to control vehicles remotely.

At present, autonomous driving is one of the most promising areas in the automotive domain. This technology consists of sensing, perception, planning and operation to improve road safety by avoiding human mistakes. The European Commission states that the nature of human driving is competitive. In reality, drivers tend to drive faster on the motorway, and they prefer to drive on faster lanes instead of slower lanes. Such phenomenon has a negative impact on road safety, efficiency and traffic congestion. Thus, prior work of slot-based driving, as an example of remote control, underpins this project as a way to reduce traffic congestion, coordinate vehicles and guarantee journey times. To support the slot-based driving approach, a large-scale TMS is proposed in this research. This TMS aims to help traffic management, enable autonomous vehicles, give guidance and control them.

This large-scale distributed traffic management system is proposed to handle large user capacity and it requires high scalability, availability, reliability and low latency. This TMS consists of App, load balancer, TMS centers (TMC), map service, road servers, road-side units (RSU), on-board units (OBU), the overall database and local databases. Users can input their requests through the App which will then reroute requests to the load balancer. Load balancers will distribute loads to TMC that will query the map service to get routes based on users' requests. This is followed by TMC querying road servers to check the road availability. The overall database will provide road servers with slot information which will then be sent back to TMC. When vehicles are travelling on the motorway, OBUs will keep broadcasting vehicle information to RSUs which will constantly update the overall database and local databases to store real-time information. This TMS is designed to be able to book the journey for vehicles and to ensure vehicles to follow the pre-defined trajectories while travelling.

A prototype of the overall design was achieved by using simulator SUMO to simulate the traffic environment. This prototype consists of OBU, RSU and slot information. RSUs will read the slot information and send it to OBUs before vehicles joining the motorway. Vehicles will be allocated a route defined by slots in OBUs. Control commands will be sent to vehicles from OBUs while they are travelling on the motorway. This implemented prototype shows the interaction between RSUs and OBUs. It also allows vehicles to be scheduled before travelling and to be controlled during the journey.

Acknowledgements

I would first like to acknowledge my supervisor Prof Vinny Cahill for his help and encouragement during this research project. He gave me valuable advice on the project itself and also the dissertation writing. Without his help, I wouldn't have accomplished this on my own. I would also like to thank Lara Codeca for helping me configure software used in this project.

Finally, I would like to thank my family and my boyfriend Jerry Yang for enduring support and advice.

Contents

List of Figures

1 Introduction

This chapter gives an account of the aims of this large-scale traffic management system project and the motivations for these aims, as well as the chapter structure of this report.

1.1 Research Motivation

Connected autonomous driving is one of the emerging areas in building the smart city. Vehicles will be able to drive on their own without any human inputs. These driver-less vehicles are designed to combine embedded sensors and software to control and navigate themselves. Unlike human driving, an autonomous driving system utilises inputs from roads, vehicles and environments to automatically control the vehicles' CAN (controller area network) bus (11). CAN is a component embedded in vehicles connecting the braking system, steering system, etc. The autonomous driving system can sense the environment and navigate to the destination, which can potentially avoid traffic accidents and congestion and effectively increase road safety. Past reports suggest there were over 5000 accidents on the M50 motorway in Ireland from 2017 to 2018 and there were also 45 percent increase in the number of drivers' deaths on Irish roads in 2019 (12). These facts show that there is a need to introduce autonomous driving for tolerating traffic incidents.

Slot-based driving approach proposed by prior work (13) is adopted as the theoretical framework in this project to achieve remote autonomous vehicle control. The slot-based driving approach pre-defines slot trajectories and behaviors, which guarantees the arrival time and regulates vehicles' driving patterns. This approach transforms the traditional driver vehicle control to remote system control. Once drivers are removed from the control loop, the overall latency of the system will decrease, allowing slots to be packed more tightly. Slot-based driving is of great benefit for increasing road capacity (14). Also, since trajectories of slots are defined before travelling, the vehicles' travel characteristics such as speed, location and lane are all predefined. Thus, there is less possibility for vehicles to be delayed and have accidents, which further lead to higher capacity and safety for the road.

As a result, a large-scale traffic management system on the motorway is proposed in this project to support the slot-based driving approach, of which enables better management of traffic. A TMS is ideally composed of applications and management tools to integrate communication, sensing and processing technology. As shown in figure 1.1, it usually involves distributed servers, on-board units (OBU), replicated roadside units (RSU) and VANET to achieve the high scalability, availability, reliability and low latency. By implementing this, TMS will be able to book the slot for users and control the vehicles in order to make vehicles follow the trajectories defined by the slot.

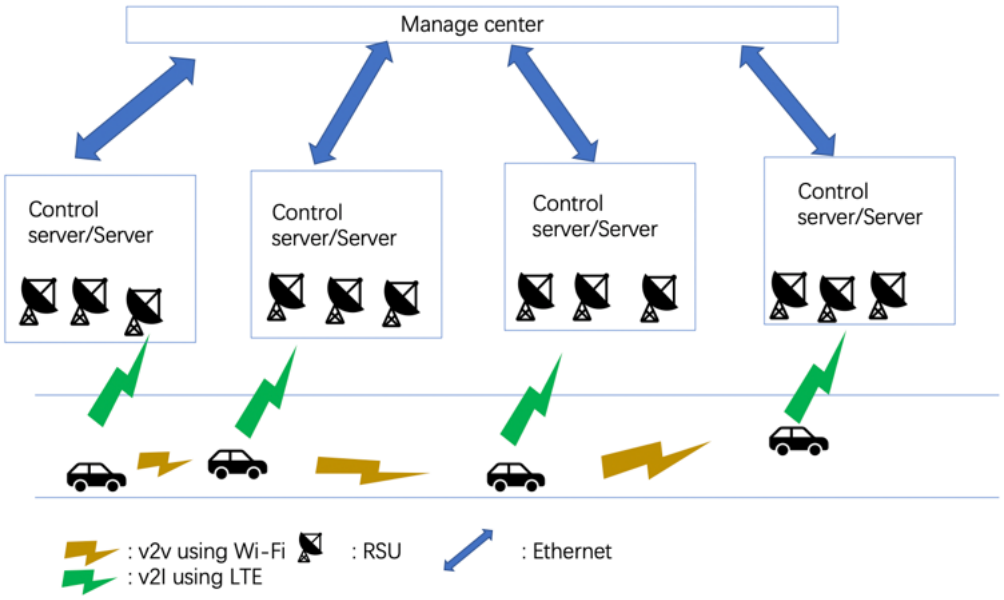


Figure 1.1: Traffic management center

1.2 Research Objectives

This project aims to investigate the design of a large-scale traffic management system on the motorway taking charge of collecting traffic information from RSUs and vehicles, analyzing it and sending according commands to vehicles. An advanced algorithm is also required to be implemented in TMS for controlling the vehicles remotely based on the information gathered. With this, the goal is to allow for the slot-based driving approach to adapt to the various conditions.

There are a few high-level requirements this TMS has to meet. The first requirement is that the TMS needs to have fast response time. Latency is one of the common concerns in large-scale distributed systems (1), as there are multiple duplicated nodes working in consequence. The point of building the traffic management system is to make use of collected real-time information and control vehicles. Therefore low latency is of great importance to the TMS design. Distribution is the second requirement for this TMS, as

there will be lots of components cooperating in the system, and locating them in different places is essential for dealing with requests and computing jobs in large-scale design. High reliability is another requirement for the TMS. It is significant for any large-scale system to do replication and partitioning to tolerate nodes' failures. Scalability is another concern in this TMS design. TMS should be more adaptable and flexible to changing demands from numerous users. A rigid design might work for specific needs, but it will not be optimal if users tend to add in more functionalities in the future. Last but not least, secure communication should also be considered in the design. Because data transition and communication between TMS and servers, RSU and OBU should be secured to prevent any information leak.

1.3 Overview of Achievements

This research presents work on designing a large-scale distributed traffic management system and implementing a prototype of the designed TMS. As shown in figure 1.2, the design of a TMS is usually composed of App, TMS center, load balancers, map service, road servers, OBUs, RSUs, overall database and local databases. App will read in users' requests and pass them to the load balancer which will then distribute these requests to different traffic management centers (TMC). TMC will query the map service (OSM) which can produce multiple routes for users. Given routes from the map service, TMC will then query its corresponding road servers, followed by road servers querying the overall database to check the road availability. If a slot available, the system will reverse the slot in the local databases for this user. When vehicles are running on the road, the OBUs in the vehicle's body will keep broadcasting its real-time information to RSUs which will then update their local database.

This research also presents a prototype of the designed large-scale distributed traffic management system for slot-based driving approach. It is achieved by using the traffic simulator SUMO which simulates the traffic networks and vehicles. Components including RSU and OBU are fully implemented. RSU is able to read the slot information pre-defined in the XML file, reformat it and pass it to OBU. It is also in charge of setting up TCP connections with OBU. OBU is used to allocate the slot information to vehicles and send commands to them. These commands can control the vehicles remotely in SUMO. Slot information is all defined and stored in the XML file for quick retrieval purposes. Before getting on the motorway, vehicles will be assigned a slot and this slot's trajectory and behaviors are both pre-defined in the system. While travelling, vehicles will have to drive in the slot they booked and the slot will be released after vehicles exit.

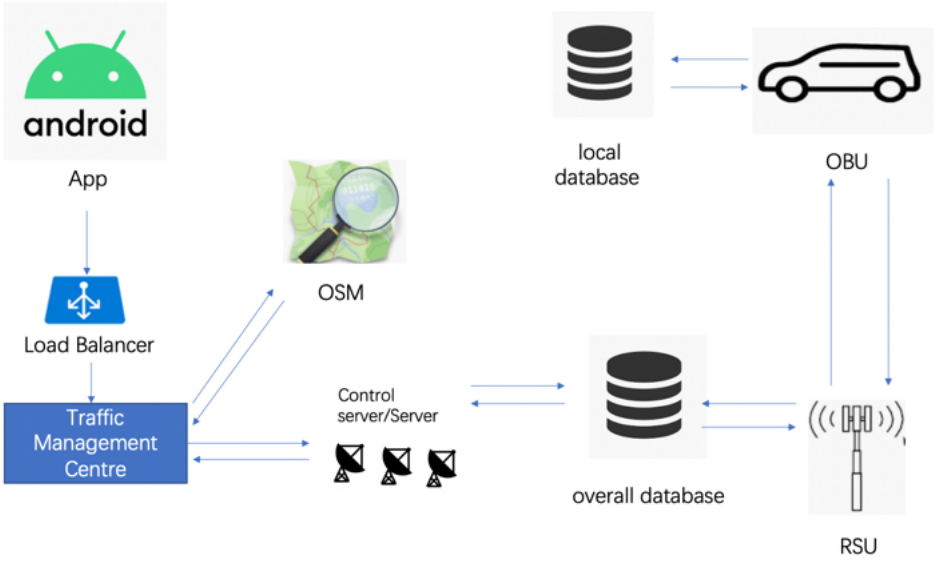


Figure 1.2: TMS architecture

1.4 Improvements and Future work

Even though the prototype is fully implemented, testing of the system's scalability, reliability, availability and latency are yet to be done. The unit tests of functionalities still needs to be done. However, considering the time and scale of this project, unit tests will not be carried out in this report.

Improvements to the individual component of the approach will propagate into this design in the future. In real-world, there are various types of sensors on the road such as in-road sensors, cameras and on-road sensors (15). They all have different roles in the traffic networks. Gathering information from different kinds of sensors can greatly improve the information accuracy. This prototype can be further improved by scaling up the distribution of components and adding in more types of sensors for TMS to make more accurate judgement about the traffic congestion and accidents. Furthermore, weather conditions and vehicle body types will also be considered for slot size adjustment in the further study.

1.5 Thesis Structure

The remainder of this dissertation is structured as follow.

Chapter 2 is entitled Background. It provides the in-depth literature review on the fields of Distributed Systems, Vehicular Communication, Slot-based Driving and Traffic Management System design.

Chapter 3 is entitled System Design. It first describes how the overall system should work in

the real world before detailing the design consideration. Next, OBU and RSU are discussed in detail about their functionalities and how they will be distributed in the system. Thirdly, this chapter discusses the design of data structure of slot information. Finally, it compares multiple vehicle communication methods in terms of reliability and latency, which will potentially be involved in the development of this system.

Chapter 4, entitled Implementation, describes the implementation phase of this project in detail. In phase one, it focuses on the implementation of two essential components, the OBU and RSU. In particular, it involves the communication between the traffic simulator SUMO and OBU. Phase two introduces pre-defined slot information and adds more functionalities to OBU. Phase three then focuses on expanding the scale of this project.

Chapter 5, entitled Evaluation, gives the result, review and in-depth comparison of design and implementation. It also addresses the limitations in this study and improvements that can be propagated to future development.

Finally, Chapter 6 is the Conclusion chapter. It gives the overall conclusion about this project and suggest potential work that should be done in the future to enhance efficiency and scalability.

2 Background

This chapter provides an in-depth literature review of four core topics associated with this project – distributed systems, vehicular communication, slot-based driving and traffic management systems.

2.1 Distributed System

A distributed system is usually a large-scale system where nodes are independent to achieve high availability, reliability and scalability (16). Nodes can communicate, coordinate and share information to one another (17). Figure 2.1 shows an example of a simple distributed system. Servers and databases in this figure are all independent and located in different places. Servers can talk directly to the cloud and they can also communicate with each other.

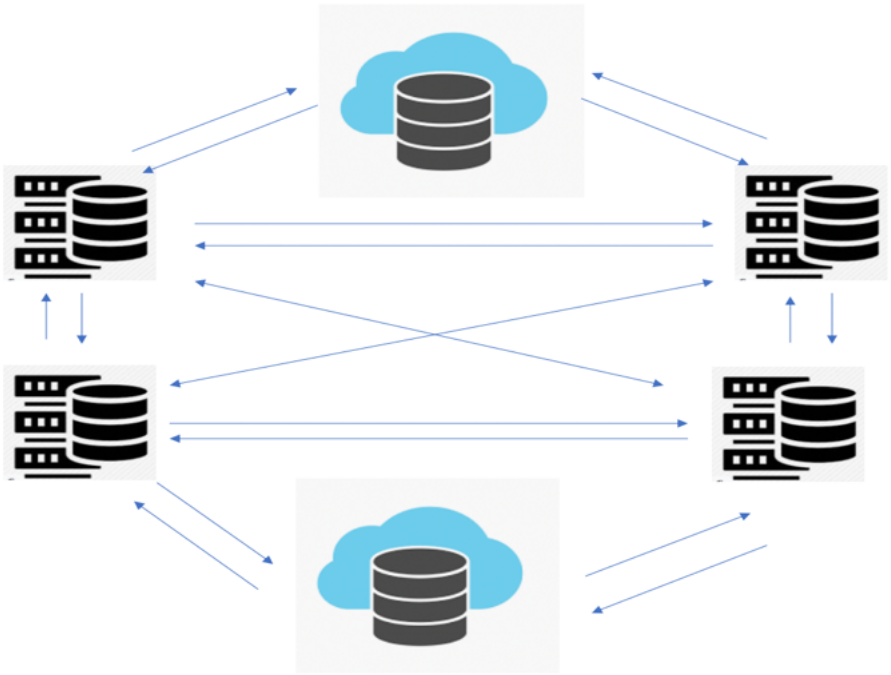


Figure 2.1: Example of distributed system

2.1.1 System Architecture

A client-server model is a distributed architecture structure for large-scale applications. Users can share common data and operations using this model (18). There are two types of client-server model: two-tier systems and three-tier systems. The choice between them is mainly based on the project's scope and implementation complexity (19). The architectures can be defined by how different components are split up among software entities or tiers. Two-tier and three-tier architecture systems will be reviewed in the following content.

There are several ways to design two-tier client-server systems and the most common implementation is to divide an application into two software entities: client application and database server. As shown in Figure 2.2, clients have to send a request to the database server and this request is usually a SQL request (19). Sending SQL requests requires a tight connection between client and server, meaning clients have to know either the syntax of querying and the structure of the data stored in the database or have the API for translating the SQL query. The location of the server is also essential for access.

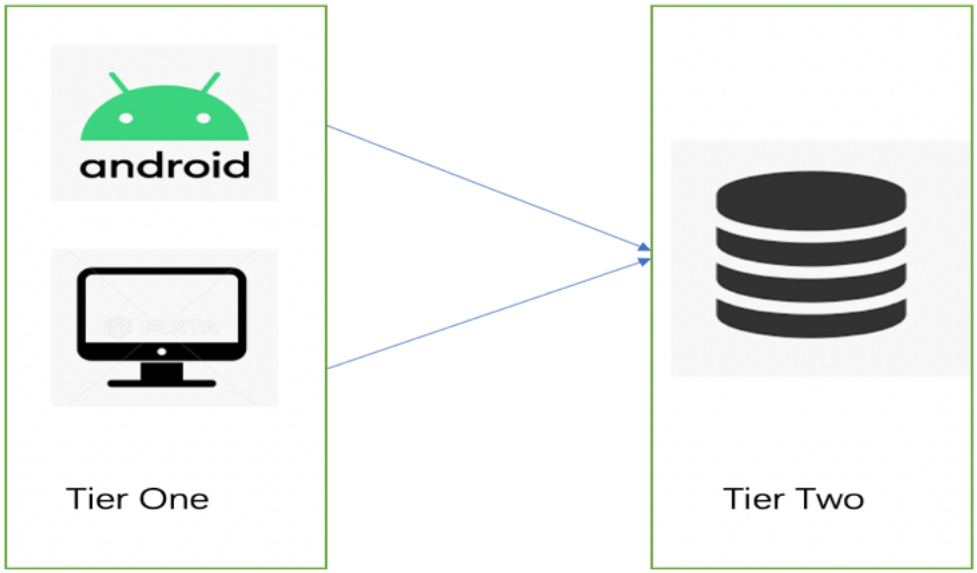


Figure 2.2: Two-tier architecture system

The most obvious advantage of a two-tier architecture system is that it provides high development speed (19). It usually takes less time to develop a two-tier system within a limited period even though the system might not be flexible enough. However, most two-tier systems reply on the logic existing on the client side. This can lead to potential application redistribution problems if a business rule requires a change in the applications' logic (19). On the other hand, security is another concern while using a two-tier architecture because multiple clients can access the database server simultaneously and there are no middleware technologies that offer security checks.

As shown in Figure 2.3, three-tier architecture systems add an application layer to the two-tier system, which helps to develop more complex and flexible applications. Tools for presentation and data processing remain the same and a call is made to the middle-tier functional server while the client is sending a request. Middle-tier servers can be multi-threaded for particular purposes and they can also handle requests for different clients simultaneously (19).

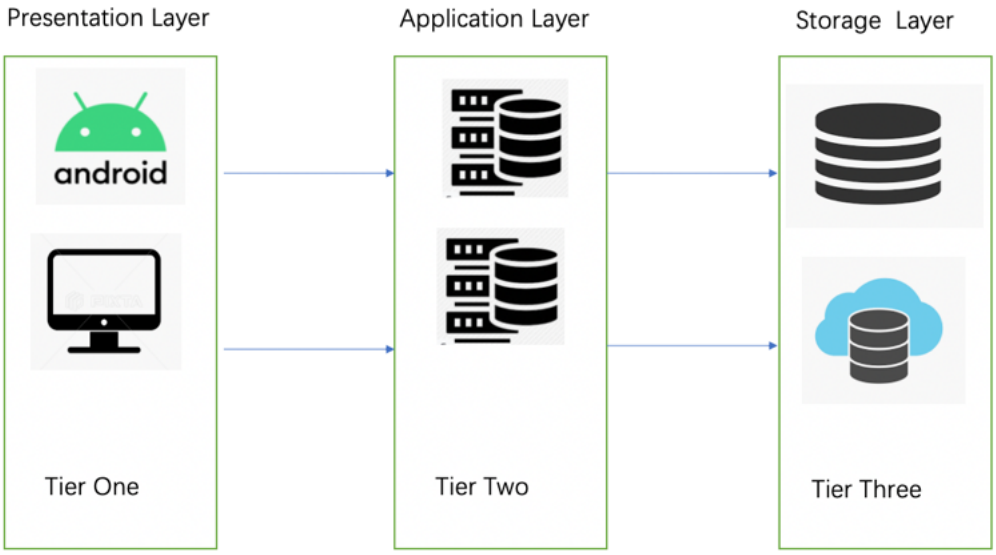


Figure 2.3: Three-tier architecture system

There are three outstanding advantages for three-tier systems.

- One is that the middle-tier server provides higher overall system flexibility (20). In each remote procedure call (RPC), the requesting clients can only pass the parameters instead of SQL queries to the middle-tier server. The server will handle the parameters and make up the SQL requests to query the database. On the client-side, SQL is not required anymore, and the data can be organised in many formats such as relational and object format.
- The second advantage is that three-tier architecture allows for parallel development of individual tiers (21) within independent development requirements. For example, UI, as the presentation layer, requires web framework and UI standards, while the application layer design focuses on the algorithms and functionalities. It is beneficial for a large-scale system in terms of the overall system quality.
- The last advantage is that the three-tier system provides for more flexible resource allocation. As mentioned above, the middle tier can contain different servers, and this will reduce the network traffic burden by splitting data and tasks to one particular service before distributing it to individual clients. Furthermore, multiple requests and

complex data will be sent from the middle tier instead of clients, which can further reduce network traffic. Besides, the load balancer can also be placed between the client and servers to help distribute the overall traffic.

However, three-tier architecture has its weakness as well. As it consists of three basic components, and ideally they are located in different locations, the code and device maintenance is a great challenge for all large-scale systems.

2.1.2 Replication

Scalability, availability and latency are the main concerns of a large-scale distributed system (1). There are three ways of scaling to high load. One is shared-memory architecture. It is also called vertical scaling, which allows the system to install more physical components including RAM chips, CPU and disks on one machine. The second way is a shared-disk architecture where all data is accessible from all cluster nodes (1). The last one is a shared-nothing architecture where data is divided into different sections and stored in different machines. This literature review will only discuss the shared-nothing architecture because this research project focuses on designing a large-scale system in which components should all be distributed.

There are two ways to distribute data across multiple nodes. One is called replication, in which each node keeps a copy of the same data across multiple nodes, ideally in different locations. If some nodes are not available, users can access data from other remaining nodes. Another way to distribute data is named partitioning. It splits the data into multiple partitions which are then assigned to different nodes.

In general, there are three approaches to achieve replication: single-leader replication, multi-leader replication and leaderless replication (22). For the first two methods, they follow the leader-follower model where the leader is one of the nodes (also named replicas) and it is responsible for sending the data change to its followers. When clients want to read the data, they can access either the leader replica or follower replica. However, if they want to write the data, they can only write to the leader replica instead of other nodes. Figure 2.4 below shows the leader-follower model in detail. Users can only write to the leader node which will propagate data changes to its followers.

In the single-leader replication process, only one leader exists in the system and clients are required to send requests and data changes only to this single leader. The leader will propagate data changes to all other replicas. As for multi-leader replication, clients can send write requests to one or more leaders who will send data change streams to each other and any follower replicas. For the leaderless replication, there is no leader in this case and clients can read and write to all the nodes in parallel to detect and correct nodes with updated data.

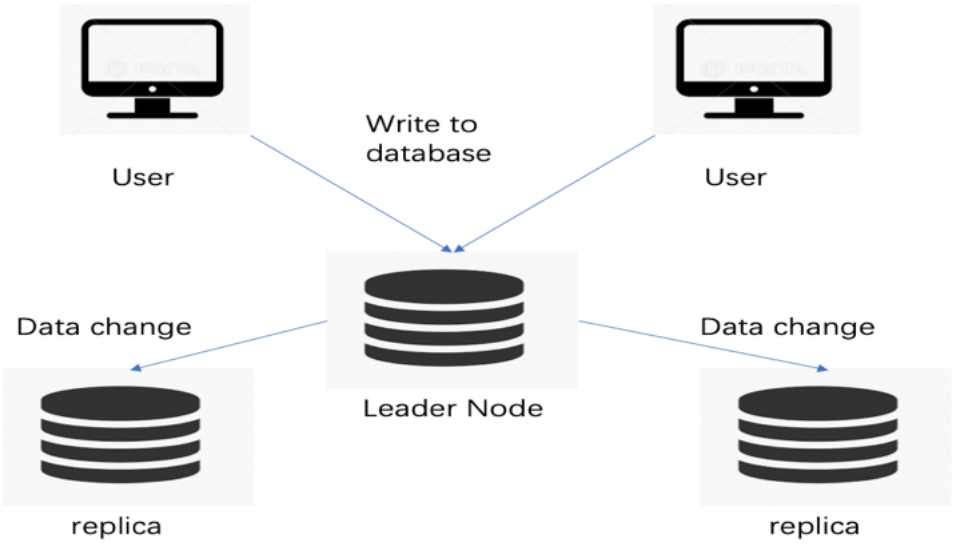


Figure 2.4: Leader-based replication

According to past literature, trade-offs to consider while dealing with replication include propagation methods and the ways to handle failed nodes (1). In general, there are two types of propagation methodologies for data replication. One is synchronous replication and the other is asynchronous replication (1). Synchronous replication requires changes to all nodes within one transaction, while asynchronous replication allows a delay in propagating changes to other nodes. Figure 2.5 below explains the difference in detail. In this figure, follower 1 is synchronous and follower 2 is asynchronous. During asynchronous replication, the leader doesn't need to wait for a response from followers to close the process. This figure shows a significant replication lag before follower 2 processes the data change message. Most of the databases we use such as MySQL (23) and Oracle database (21), provide fast replication services. However, they still cannot guarantee how long the replication process takes. There might also be some occasions that follower's data falls far behind the leaders or it is recovering from failure. Therefore, in reality, it is impractical for systems to have full synchronous replication. Matsunobu pointed out that semi synchronous configuration is applied in practise (24). If one synchronous becomes asynchronous, one asynchronous will be made synchronous to ensure the completion and freshness of data.

2.1.3 Partitioning

It's not sufficient for a large-scale distributed system to only do data replication. Data has to be broken into multiple partitions to increase the system scalability. The process of doing it is called partitioning and it is also known as data sharding. Each partition acts as a single database and stores its own data. Figure 2.6 below shows the partitioning of the car park in the real world. The parking lot has three partitions and each of them parks one type of

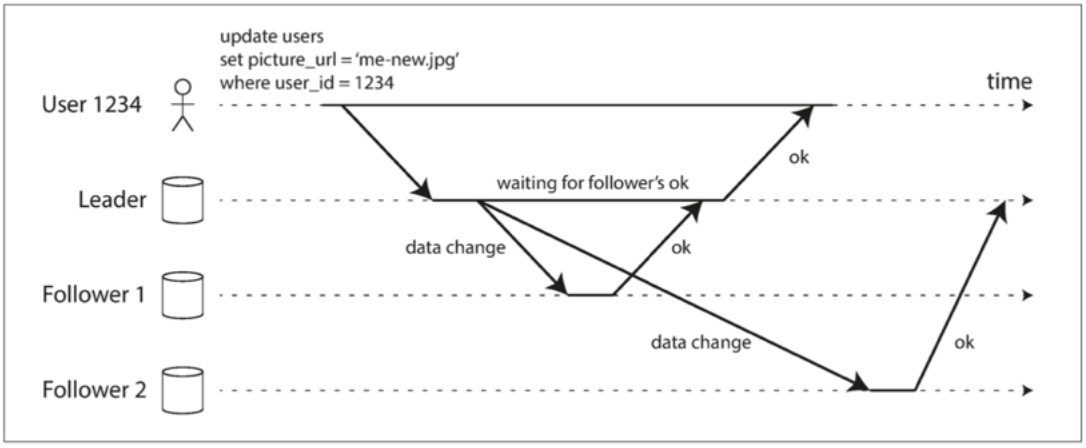


Figure 2.5: Leader-based replication with one synchronous and one asynchronous follower (1)

vehicles. The partitioned database has already been used in distributed execution engines such as Hadoop (25) and MapReduce (26). These systems' data partitioning is designed for different purposes. In this chapter, I will only review the fundamentals of partitioning even though each distributed system has different partitioning techniques.

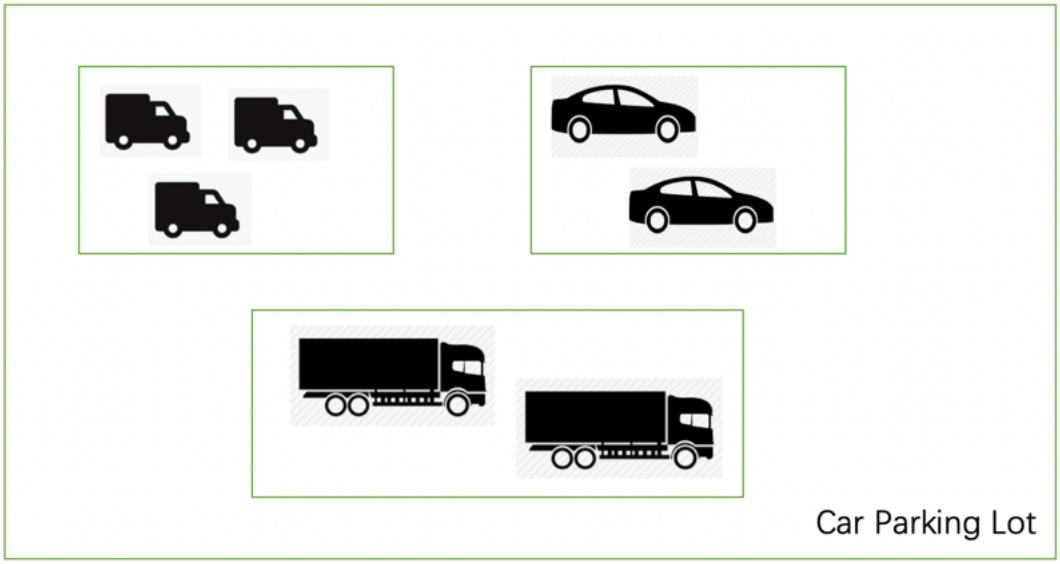


Figure 2.6: Partitioning in the parking lot

System performance will be an issue if a significant number of queries are processed to scan over all the data. Partitioning is a reliable way of solving this problem (27). In practice, partitioning is usually used together with replication, which means the data on one partition can also be found on other replicas. By implementing this, the fault tolerance of the system can be raised. Figure 2.7 below gives an idea of how the system using both techniques looks like. Partition 1 in Node 1 is replicated in Node 3 and Node 4, which means if node 3 is down, users can still query node 4 and node 1.

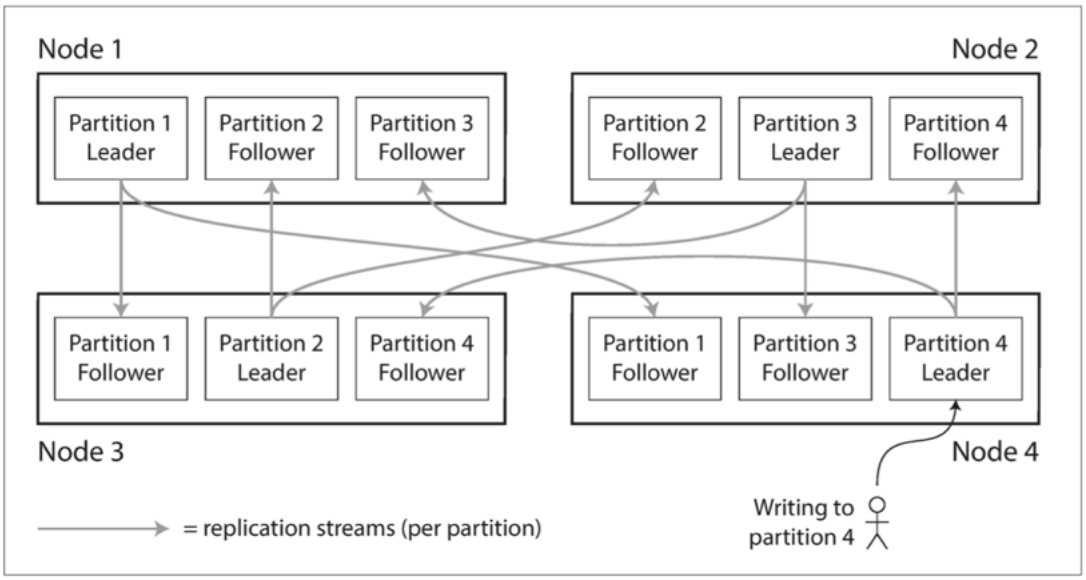


Figure 2.7: Combination of replication and partitioning (1)

The goal of partitioning is to spread the data evenly across nodes for efficient querying. However, there is a possibility that the partitioning is not fair. For example, one extreme case would be one partition has all the data and the rest of nodes have nothing. This situation is called skewed. One way to relieve this problem is key range partitioning. It is based on a key-value partitioning approach and is also used by MongoDB (28) and HBase (29). As shown in figure 2.8 below, a range of keys including A, B and C are allocated to each partition and users can set the key boundaries based on their needs. Administers can also set the key to be sorted for better query performance.

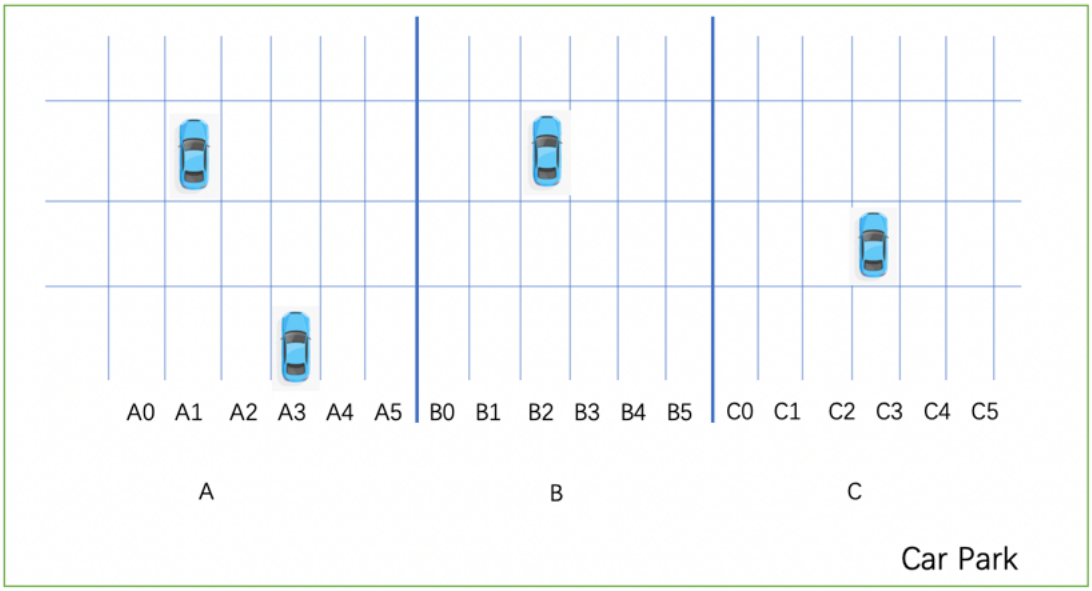


Figure 2.8: A car park partitioned by the key range

One downside of this approach is that key-range still cannot fully avoid hot spots and skew issues. Thus, hash-key partitioning is proposed for a fair key distribution. It takes advantage of a hash function to determine the position of the given key. This hash function doesn't need to be strong, but it has to suit the systems' needs. For example, simple MD5 encryption is used by several non-relational DB such as MongoDB (30) and Cassandra (1). This approach separates data more randomly but it loses the strength of the sorted key, which leads to low efficiency. The concatenated index approach has been proposed to solve this issue (1). It combines multiple fields into one key by adding one column to another. This approach consists of one primary key for location scan and the other concatenated keys for efficient range scan. For example, on a social media website, a user might follow a lot of people. If the primary key is chosen to be follow-timestamp and user iD, it would be easy to find people who were just followed by this user within some time interval.

In reality, the situation might be more complex if partitioning involves secondary indexes that are usually used to search for the occurrences of a value. Due to the high complexity, some databases such as Hbase (31) have avoided using secondary indexes while others such as Riak (32) have started to take advantage of them because they are efficient for data modelling.

There are two ways to partition a database with secondary indexes. One is indexed by document, shown in Figure 2.9. Secondary indexes are saved in the same partition as the primary key. Another approach is to partition the database by terms, where secondary indexes' positions are determined by terms instead of documents. Figure 2.10 gives an example of how it works in detail. All the red cars in two partitions are classified under color:red using secondary index. However, the index is also partitioned and thus colors starting with a-r can be found in Partition 0 and the rest can be found in Partition 1.

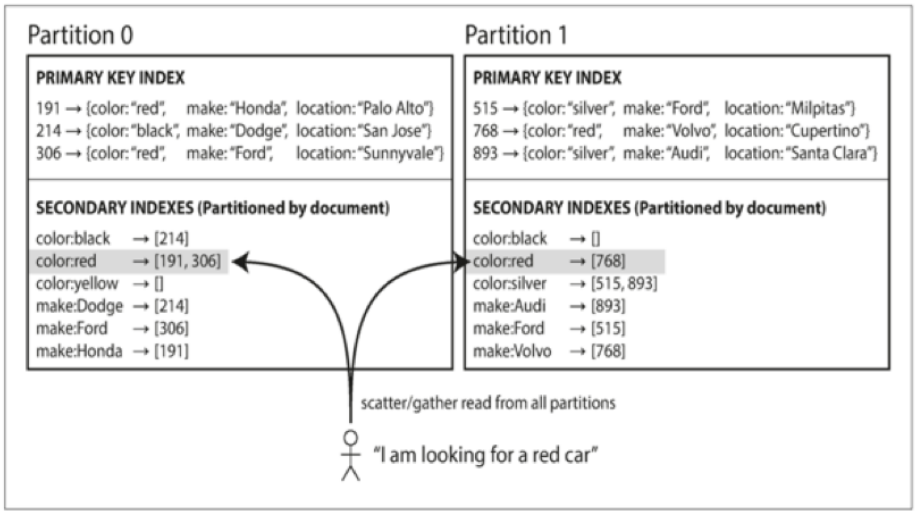


Figure 2.9: Secondary index by document (1)

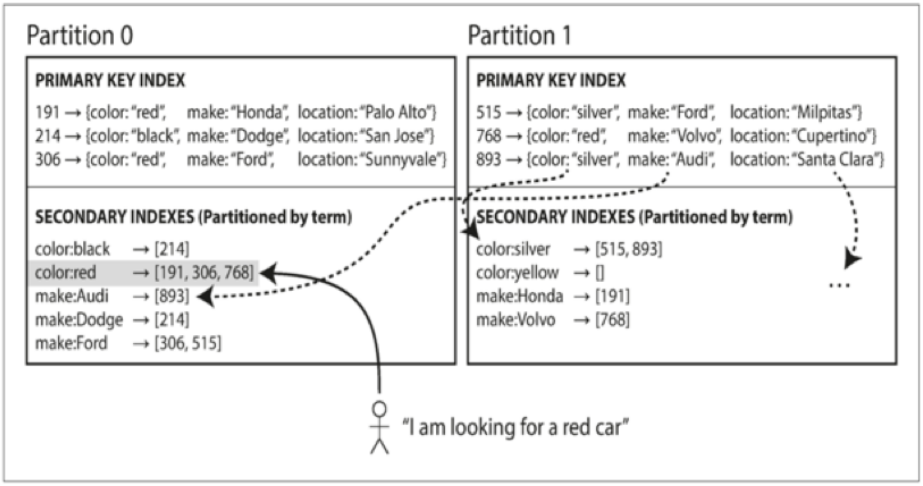


Figure 2.10: Secondary index by term (1)

This section looked at two architectures and several techniques for building a large-scale distributed system. Informed by the advantages and disadvantages listed above, users should choose the one which suits their needs best. Considering the goal of this research project is to build a large scale system, a three-tier architecture and replication technique would be used for later development.

2.2 Vehicular Communication

Vehicular communication is an emerging area of research in the field of building smart traffic management systems to improve safety and efficiency. It is defined as communication between vehicles (33).

VANET, also named Vehicle ad hoc network, is an advanced network which takes advantage of wireless communication to connect vehicles and road sensors for fast and accurate data exchange and transmission (33). It consists of three components: on-board units (OBU), roadside units (RSU) and authentication server (AS). OBUs are installed in each vehicle and RSUs can be any sensors that are deployed on the road as an infrastructure to gather road information. Figure 2.11 below gives a straightforward explanation of what the infrastructure of VANET should look like. As shown in this figure, V2V and V2I both represent vehicular communication. For V2V, it stands for vehicle-to-vehicle communication. In this dissemination, data transmission only happens between vehicles within the valid wireless range. If one vehicle is outside the range, data will be sent to another vehicle as a middle point. Whereas, V2I stands for the vehicle-to-infrastructure communication. In this case, data transmission happens between vehicles and some sensors embedded on the road units, such as RSUs. Both V2V and V2I belong to the V2X, vehicle-to-everything, category.

In VANET, wireless technology is widely used and there are some trade-offs between

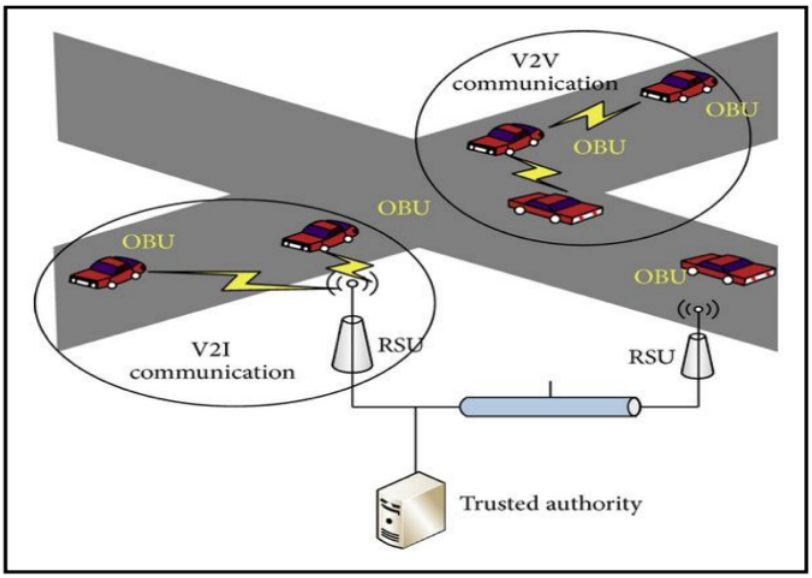


Figure 2.11: VANET (2)

different protocols. Dedicated Short Range Communication (DSRC) and Long Term Evolution (LTE) are two main candidates for Connected Vehicle (CV) and they will be reviewed respectively in the following section. DSRC is an IEEE 802.11p-based (34) wireless technology for safe data transportation and high efficiency (4). It is commonly used in V2V and V2I for data transfer. As for LTE, it is a 4G standard for wireless broadband communication for mobile devices, which is slower than real 4G but much faster than 3G. Figure 2.12 below is a detailed comparison between DSRC and LTE in terms of bandwidth, latency, transmission distance and reliability.

	Bandwidth (Mbps)	Latency (Ping in ms under the same condition)	Transmission Distance (metres)	Reliability
DSRC	Up to 27	150	300	Handle fast and frequent handovers in-vehicle environments
LTE	30 - 75	98	1000	Data is encrypted, identity is authenticated and protected.

Figure 2.12: DSRC vs LTE (3) (4) (5)

There is still a lot of debate between these two protocols. However, most of the researches only used software-based simulation, which might not be reliable or realistic. In the research carried by Zhigang et al (35), a real vehicle test-bed was established for testing real-world scenarios. The study analysed vehicles' behaviours using two protocols mentioned and gave a few experimental results. With two vehicles running using different protocols, the one using DSRC shows the lower Packet Loss Rate (PLR) during the round-trip time (RTT), which means that DSRC is more suitable for V2V traffic safety applications. Another test of two vehicles running past RSUs shows that for DSRC, the PLR is lower while vehicles are closer to the RSU and it will increase immediately while getting further. However, for LTE, the PLR keeps stable for a wide range because LTE has larger bandwidth and longer transmission distances. Within the valid transmission distance of DSRC, 300 meters, the PLR of DSRC is significantly lower than that of LTE. As a result, LTE is suitable for cutting down the density of RSUs on the road because of its high coverage while DSRC is better for safety applications such as car accidents broadcasting or driver warnings.

OBU is one of the essential components in the VANET for data collection and processing. Its hardware platform consists of an ARM11-based embedded development platform, a dCMA-86P2 module and a GPS model to support DSRC/WAVE, WiFi and LTE communication protocols (36). Its software platform is based on the Linux operating system. There is a vehicle communication interface where the message is passed from OBU to CAN bus (11) to control the vehicles.

RSUs act as a router in the VANET, which are connected to the internet and forming an infrastructure that provides the ability to communicate with servers, vehicles and other RSUs (37). RSUs are usually firmly distributed on the road based on the networking protocol transmission distance. Algorithms have already been implemented to position the RSUs in the effective locations in the urban scenarios (38). Considering a situation that a RSU wants to send a data packet to one vehicle, firstly, the RSU needs to get the location, speed and direction of the vehicle from the last packet received from the vehicle. After examining these properties, RSU will work together with other protocols to determine whether to send the message or not (39).

To conclude, this section looked at the several technologies applied in the vehicular communication area and provided an in-depth comparison between them. Both DSRC and LTE support the V2I communication and they are chosen mainly based on their transmission distance. Component in VANET were also reviewed and all components reviewed will be used in later design process of the TMS.

2.3 Slot-based Driving

Highways are widely implemented in most countries, they provide high-speed routes along frequented on-ramps and corridors. They can also save journey time and travel cost. However, due to its popularity, congestion is the main issue faced by highway infrastructures (40). Unlike other transportation methods such as train and bus, there is no timetable for vehicles on the motorway and thus, motorways cannot guarantee the travel time to commuters. Other than these, the existing road network is becoming increasingly crowded because of the quick increase of vehicles and the facts that there is no space for building large traffic networks. The European commission estimated that traffic congestion costs 1 percent of EU GDP per year, which is 100 million every year (41). The number of cars per thousand persons has increased from 232 in 1975 to 460 in 2002 (42). The overall distance travelled by road vehicles has tripled in the last 30 years, and in the last decade, the volume of road freight grew by 35 percent contributing to 7500 km or 10 percent of the network being affected daily by traffic jams (42). All of these facts indicate that some new approaches for improving road efficiency, road safety and reducing traffic congestion have to be developed and introduced.

Virtual slots, similar as "leaky bucket" (43) can be adapted to the road network by mapping the vehicle to slots. This traffic shaping technique is applied to the vehicles when they are running on the motorway instead of only merging into the motorway.

In 2006, Morla proposed an approach (44) that can lead to congestion-free traffic by cooperating vehicles. His version introduced the slots that represent dynamic time-space corridors and these slots can negotiate between each other to guarantee the congestion-free journey. It aimed at preventing traffic congestion and detecting collisions ahead by coordinating objects including landscape objects and vehicles themselves. The slot-based concept is similar to time-division multiple access (TMDA) data slot. Any vehicle that has been allocated a slot will never experience congestion in his approach. Figure 2.13 shows what the slot-based driving system looks like. White rectangles represent slots. Each road is divided by multiple slots. The blue vehicle in this diagram is following the trajectory (dotted line) defined by slots.

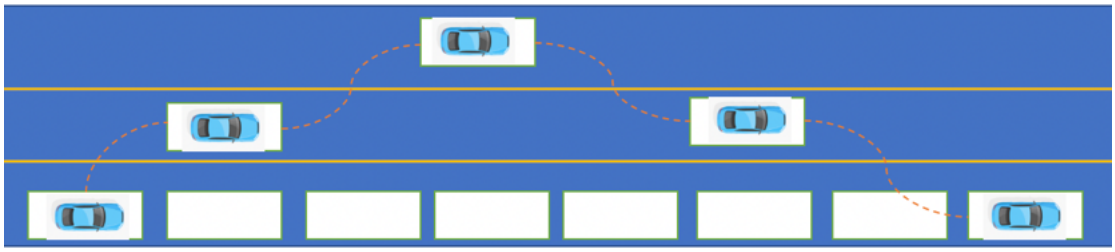


Figure 2.13: Slot-based driving diagram

A similar concept was proposed by Ravi et al (45), but it is a more business-centric version. His approach took advantage of a high-priority lane to combine the slot-based driving system with existing driving.

Real-time vehicle scheduling concept was proposed by Cahill et al (13) in 2008. This concept was also used in Marinsecu's research to achieve efficient on-ramp traffic merging on the motorway (46). Their approach suggested exploiting vehicle-to-vehicle and vehicle-to-infrastructure communication and related technologies. The approach in Cahill's paper divided the road into multiple sections and each slot belongs to one section. Vehicles will be assigned a slot before getting on the road based on their prioritised requests and while travelling, vehicles will have to stay in the slot. This approach is very similar to a channel access method called time-division multiple access (TMDA) which allows users to share the same slot generation frequency and allocate slots to sections in transit through the network. In Cahill's research, he proposed that slot should be defined by size of slot, position at specific time and predefined behaviour as a sequence of accelerate, stop, decelerate and lane changing. Vehicles will have to drive within the slots while running on the road (13).

This section reviewed several slot-based driving approaches proposed by Morla, Ravi and Cahill. These approaches were all designed for reducing the traffic congestion and managing traffic efficiently. Cahill's research gives a detailed explanation on the slot model which will be utilized in the design of this research project.

2.4 Traffic Management System

There are more and more vehicles running on the road nowadays and this fact can lead to heavier traffic congestion which will further cause more accidents, time loss and delay of emergency. To ease traffic pressure, an advanced TMS should be introduced. In Gomides's research (6), he mentioned that congestion can eventually have huge negative impacts on economic development. Thus, he proposed a fully distributed VANET-based traffic management system called FIRE-NRD which takes the information shared between vehicles and makes next step decisions for the vehicles based on the received information. Figure 2.14 gives an insight into what this TMS looks like. It involves three phases including knowledge discovering, knowledge sharing and next road decision. Vehicles will monitor their surrounding environments and estimate the congestion level in the first phase and share this information to their neighbours in the second phase. Final phase uses the received information to find an alternative route with low congestion level for the vehicles.

In Djahel's research in 2015, he reviewed the development phases of modern traffic management systems, existing technologies and some future directions to make current TMS more efficient (7). He pointed out that a typical TMS consists of a few phases as

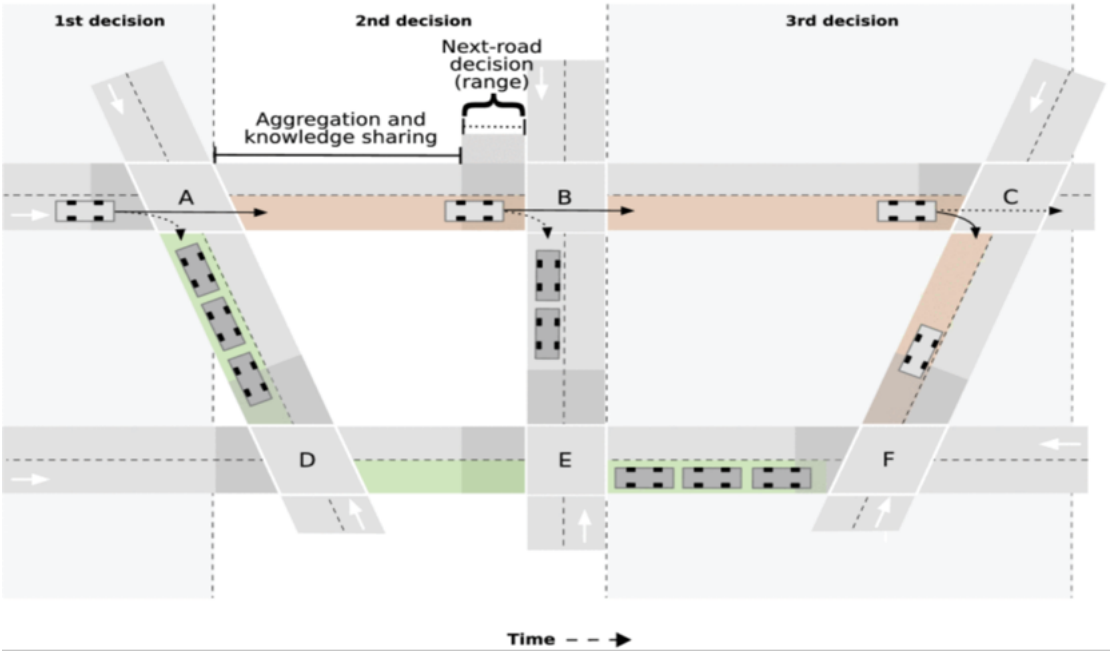


Figure 2.14: Basic Operation of Fire-Nrd (6)

shown in figure 2.15. The first phase, named Data Sensing and Gathering, takes advantage of V2I, V2V and sensing technology to collect data from vehicles and road components. The second phase, named Data Fusion, Processing and Aggregation, is to extract useful information. The third phase, named Data Exploitation, is to compute the optimal routes for users using the processed information. The last phase, named Service Delivery, is to deliver the knowledge to end-users such as drivers via devices.

A similar idea was proposed by Souza et al in 2016 (8). He mentioned that preventing traffic congestion and improving traffic efficiency mainly relies on building modern traffic management systems which usually consists of a set of applications and management tools to integrate communication, sensing and processing technologies. In the TMS he proposed, VANET is the main component and it can enable the V2V and V2I technology which are used for vehicle communications. Figure 2.16 shows the overall architecture of the TMS Souza proposed. From the diagram, we can see that vehicles can collect information from OBU and share it with surrounding vehicles by V2V technology. Vehicles can also send information to the RSUs within the transmission range by V2I technology. RSUs then send the information to the traffic management center and the data will be exploited in the cloud. After data aggregation and exploitation, the according commands or instructions will be sent back to vehicles.

There are three phases in this TMS: information gathering, information processing and service delivery. Information gathering phase is in charge of collecting traffic-related information from various sources such as in-road sensors, road cameras and vehicles. As for the information processing phase, the data can either be processed locally or in the TMC

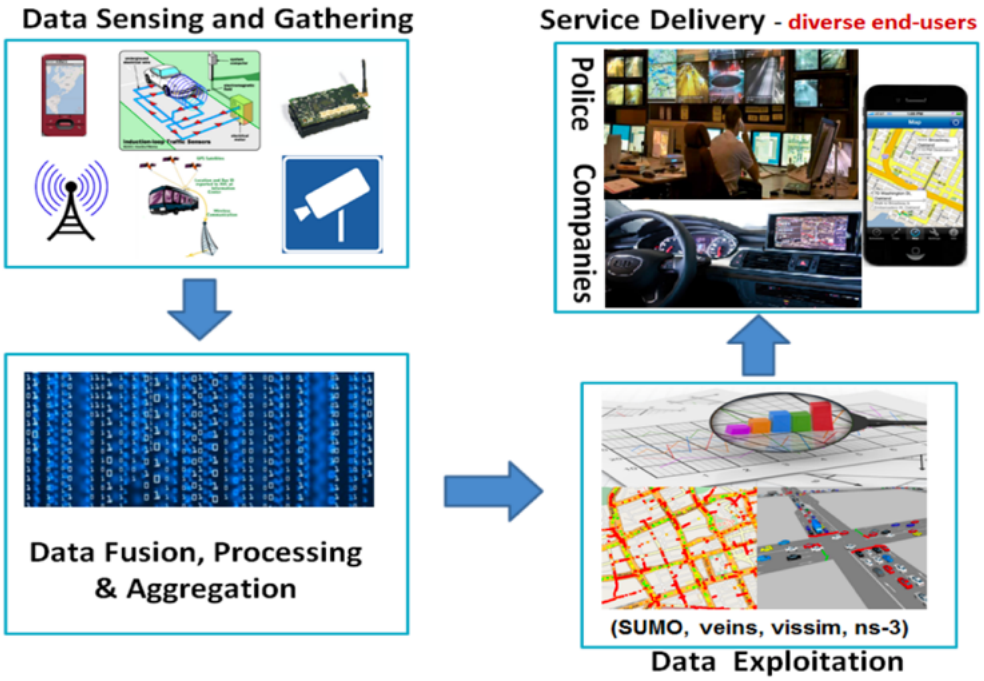


Figure 2.15: Data life cycle in smart transportation (7)

(cloud). The service delivery phase will provide the corresponding services based on the information it got from the previous phase.

There are a variety of researches about TMS and they are mainly divided into two parts: Infrastructure-free TMS and Infrastructure-based TMS. Infrastructure-free TMS (8) represents fully distributed TMSs which consist of congestion detection, avoidance, accident detection and warning, while infrastructure-based TMSs includes traffic light management, congestion detection, route suggestion and speed adjustment.

Many open challenges still exist in the development of advanced traffic management systems. Most significant challenges are related to information gathering, storage and aggregation (8). The challenge in terms of data gathering is that data is collected from many heterogeneous resources, which means the methods used for collecting data are different and the data format may vary. This leads to the issue that it's hard to synchronize the data and share it with other components. Data storage suffers from the same issue as data gathering. An XML-based storage system is currently the most popular approach for dealing with this issue. For data aggregation, the biggest challenge is the data conversion issue because the amount of data is increasing very fast and it is complex for any algorithm to convert heterogeneous data to one within a short time.

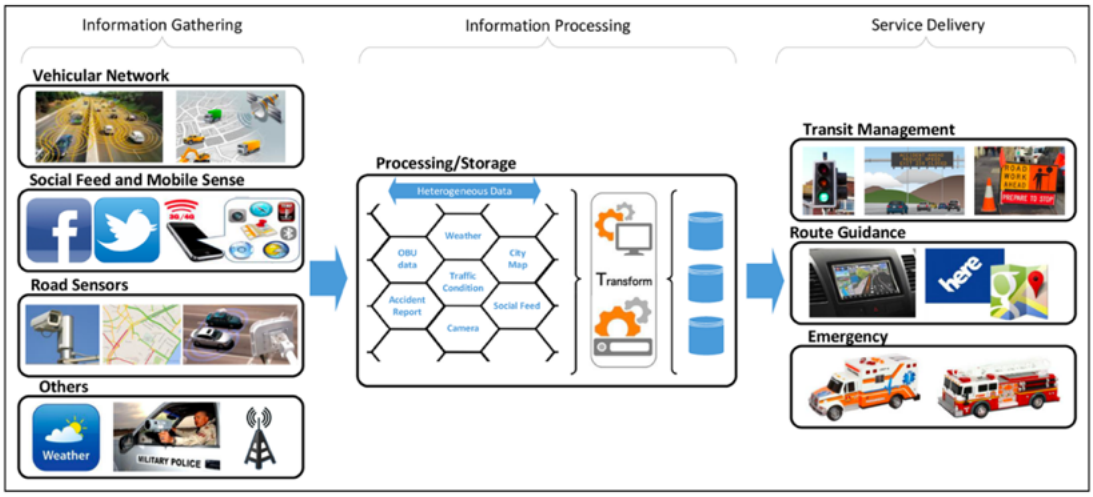


Figure 2.16: Overall TMS architecture (8)

2.5 Conclusion

To conclude, this literature review builds an overview in the field of designing the TMS. The four areas reviewed can provide insights to the composition of the large-scale TMS system in this research project.

Studies that are particularly useful include the slot model proposed by prior work (13), various replication and partitioning techniques, and the TMS design. All the researches can potentially help to build a slot-based large-scale TMS for remote vehicle control in the later design and development.

3 System Design

This chapter gives an in-depth description of how the system and its components are designed. As mentioned in Chapter 1, a good TMS system design should include four key characteristics: reliability, scalability, reliability and low latency. This TMS will be designed towards these requirements.

This chapter will focus on the system design and the implemented prototype will be discussed in Chapter 4.

3.1 Overviews

Before getting into the detailed design, there are some overviews and requirements about the system that the design has to follow.

- This TMS should only remotely control vehicles on the motorway, meaning it should not be responsible for any actions of vehicles before joining the motorway.
- This TMS should be able to handle a huge number of users and requests. For example, if this TMS is in charge of 5 segments of the motorway and each segment is 1000 meters, this TMS can then handle 50 users and their requests at a time.
- To achieve the system reliability and communication efficiency addressed in Chapter 2, the system should be distributed. All components should be located in different locations.
- The system should respond to users' booking requests quickly, with no significant delay.
- This TMS should be adaptable to frequently changing demands.
- The motorway on-ramp is a short section of the road allowing vehicles to enter the highway (47). In this design, one component should be placed at the ramp specifically in charge of checking whether the running vehicles have a booked journey or slot.
- This TMS should get multiple available routes for users and determine the optimal

route which adapts to users' priority.

- Journey in this design is a term representing the period for vehicles to travel from the beginning to the end. The route is made up of edges, such as A->B->C. Trajectory defines the route more specifically and it is made up of a sequence of slot information. A slot has a position at a particular time and slot is moving with time.

With the insights of the above requirements, the detailed design process is carried out below.

3.2 Slot Design

Slot model in this design utilises the model proposed by prior studies (13). Each slot is defined by the size of the slot, slot position, speed and pre-defined behaviours. The slot size should consider stopping distance and weather conditions. It should be adjustable based on vehicle conditions and weather. In terms of road safety, the stopping distance is composed of thinking distance and braking distance (9). Based on the formula shown in figure 3.1, the stopping distance of the running vehicle with speed 120 km/h is 95.45 meters. The average vehicle length is 4500 mm (48) , therefore a standard slot size in dry weather is 100 meters. In the wet weather, the slot size should be 200 meters to prevent accidents caused by sliding. The slot model in this design also defines the driving speed of 120km/h which is the maximum driving speed on the motorway in Ireland.

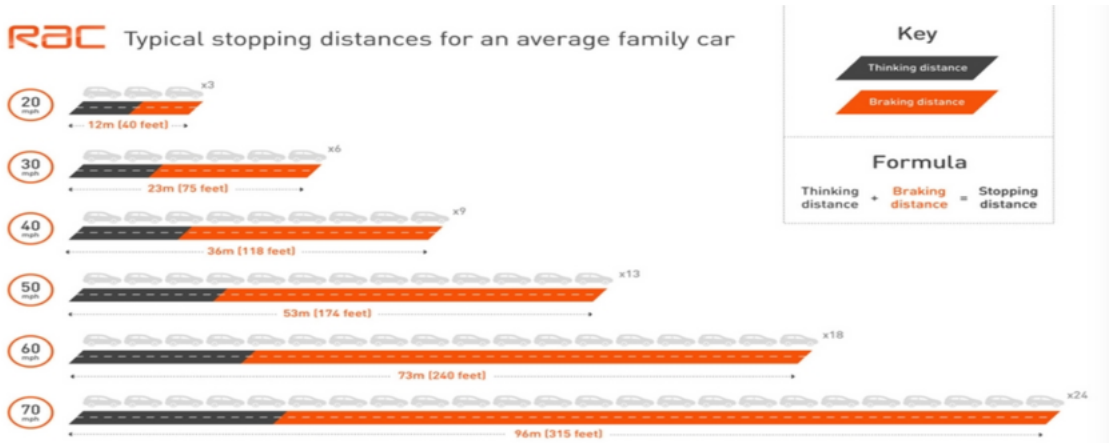


Figure 3.1: Stopping distance formula (9)

In this design, edges define segments of the motorway. Each edge can accommodate multiple slots and each slot are moving with time. Each vehicle will be allocated to only one slot on each edge and it can only drive within this slot. Figure 3.2 demonstrates the trajectory of Slot 1 on the road.

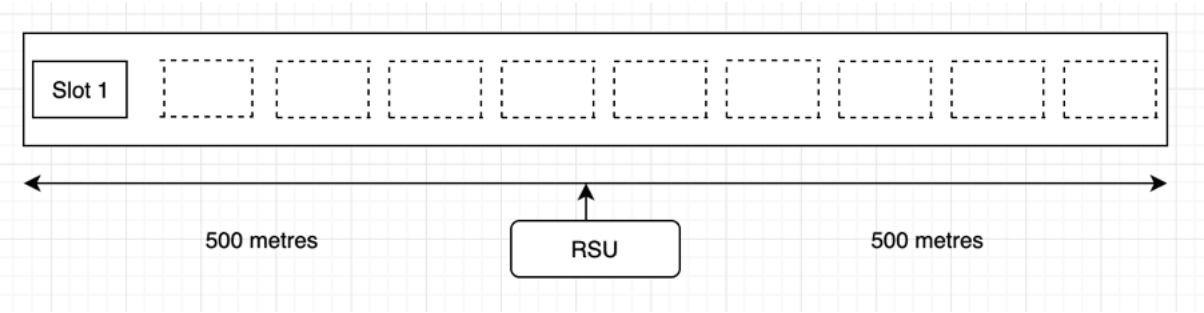


Figure 3.2: Slot trajectory

3.3 Overall Architecture Design

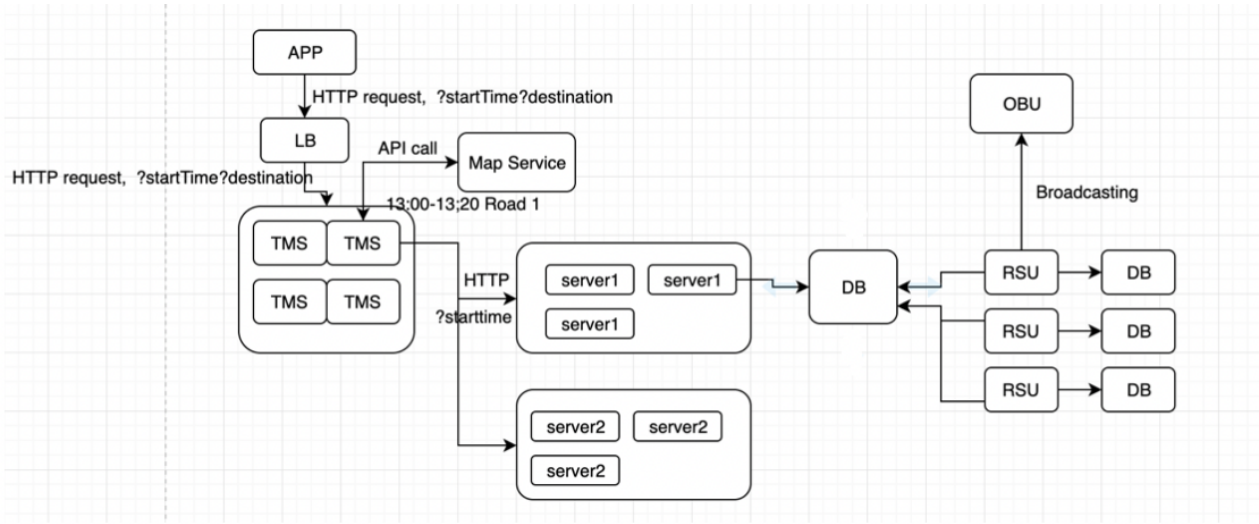


Figure 3.3: Overall system architecture

Figure 3.3 above shows the overall system architecture. This architecture includes components like App, load balancer, TMS centres, Map Service, servers, overall database, RSUs, OBUs and local databases. Each of the components will be discussed in detail in the following sections.

3.3.1 App

The App represents a phone application through which users can interact with the system. It is a user interface designed to take in users' requests. Users' starting positions, preferred starting time and destinations are required as the parameters passed into the App. Requests will ideally be made by HTTP(S) protocol because the header of requests in this design is small and HTTP has the compression capability for faster transfer speed (49). This user interface is the first step for users to book the journey through the TMS.

3.3.2 Load Balancer

The load balancer acts as a reverse proxy (50). It helps to distribute loads across a large number of servers, shown in figure 3.4. It provides higher scalability and flexibility for the system as it ensures no server is overworked (51). Load balancers are usually grouped into two categories: Layer 4 and Layer 7. Layer 4 applies to data transferred within network and transport layers, while Layer 7 applies to the data from the application layer (50). There are various load balancing algorithms providing different benefits. For example, Round Robin is for distributing requests across servers sequentially, while Hash is used to distribute the requests based on the key that developers define (52). In this project, the load balancer is only used to distribute numerous users' HTTP requests from the App, thus IP Hash is sufficient for this system because the distribution of requests is only based on servers' IP address. NGINX, a commonly used HTTP load balancer (53), is decided to be used in this architecture because it is open source and provides IP Hash algorithm.

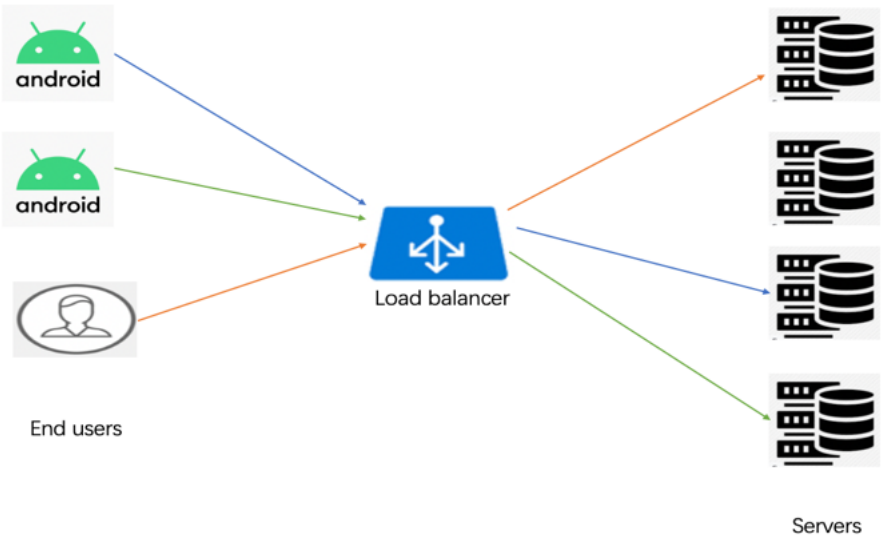


Figure 3.4: Load balancer

Map service makes use of the mapping service on top of OSM to generate routes, which can be accessed by the API call from the TMS. OSM provides the functionalities for users or developers to draw the map based on their needs. In this design, the mapping service takes in users' starting positions and destinations and then generate multiple routes for them. The routes generated from the mapping service are designed to be made up of edge IDs. For example, route 312 represents edge3-> edge1-> edge2.

3.3.3 TMS Centre

As mentioned in Chapter 2, data replication can help to tolerate node failures. Thus, the TMS centre, as shown in figure 3.3, is replicated to achieve scalability and availability. The

single-leader model is used because it can avoid the conflicts by concurrent write (54). In this design, it takes in users' requests from the load balancer and makes an API call to the map service. Multiple routes will be generated by mapping service and each of them will be sent to different TMS centres. TMS centres will then send requests to corresponding road servers to check the road availability for a particular period. When vehicles are running on the motorway, TMS will also frequently query the road servers to check traffic congestion and accidents. Lastly, it is worth mentioning that TMS is also in charge of determining the slot size based on the weather.

3.3.4 Road Servers

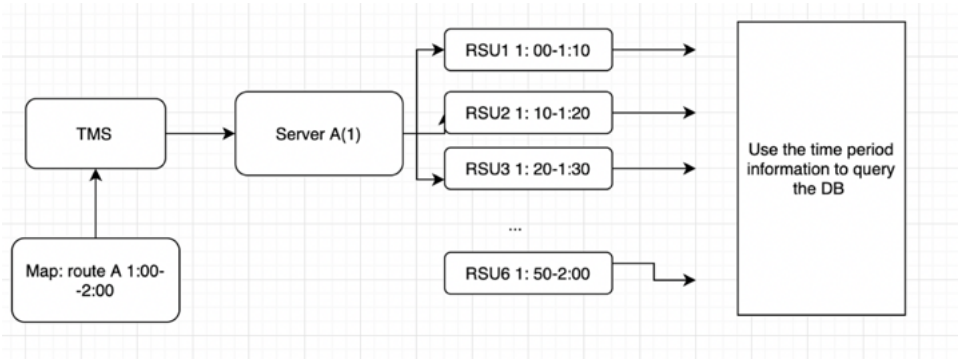


Figure 3.5: Sending requests to the server

Road servers in this traffic management system are used for cooperating TMS centres and RSUs. Servers are replicated and partitioned in this design, meaning that there is only one server on each road. Each server is also in charge of all of the RSUs on that road. Figure 3.5 shows an example of TMS sending requests to servers. In this figure, the route only consists of edge A whose starting and ending times are 1:00 and 2:00 respectively. Therefore the TMS only sends requests to server A, which will further partition the period and send requests to distributed RSUs based on the sub-period.

All the servers can write to the overall database for storing real-time information, which will be discussed in the following sections.

3.3.5 RSU

RSU is one of the most important sensors for gathering real-time information in real life. In this design, RSUs are used to gather information from vehicles. It also acts as an interface for servers to query the local database to book the slot, which will be discussed in detail in the User Scenario section. RSUs are partitioned every 1 km along the motorway based on the LTE transmission distance, RSUs will only gather information within this effective range. As mentioned in the overview section, there should be one component in charge of checking the vehicles' booking information. Thus, one RSU will be placed on the motorway ramp and

it will start querying its local database while vehicles are within its transmission range. If vehicles don't have a booked slot, they will have to book the journey through the App first. Each RSU has its local database which stores slot reservation information and real-time vehicle information. As shown in figure 3.6 below, this RSU is only in charge of 10 slots. Ten slots can be accommodated within the RSU's transmission range and this RSU's local database will only store information of these slots.

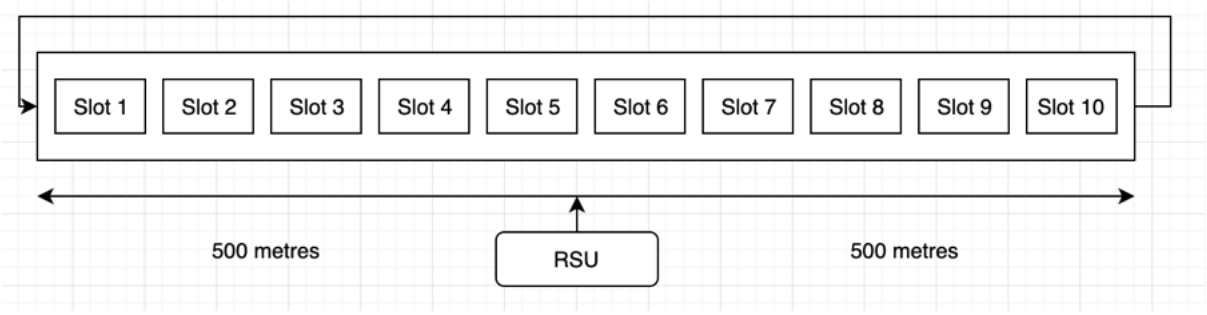


Figure 3.6: Slots within RSU transmission range

3.3.6 OBU

OBU is the component embedded in the vehicle body. It is used for obtaining vehicles information such as speed and positions, as well as broadcast information to the RSUs which will then send back the corresponding commands to control the vehicles.

3.3.7 Database

There are two databases used in the design: overall database and local databases. Road servers will write to the overall database and each RSU writes to its local database. As for the database chosen, there are two types of databases in production: relation databases and non-relational databases, each with different strengths and weaknesses. Three main factors to consider while choosing the database are data structure, data size and retrieval speed (55). As mentioned earlier, this TMS should be able to handle numerous users and there is a lot of real-time information needed storing in the database, making the data size of users and vehicles quite large. TMS will also have to frequently query road servers to check the traffic congestion and accidents, which means it requires high information retrieval speed. As for the data structure, both the overall database and the local database contains four tables. Figure 3.7, 3.8, 3.9 and 3.10 below give an idea of what attributes the local databases should store. These figures indicates that using a relational database is more suitable because each of the tables has a RSU ID as the primary key and these tables are all related.

iD (same as RSU)	Timestamp	Curr slot 1 info	Curr slot2 info	Curr slot ··· info
		Occupied	Not occupied	

Figure 3.7: Current slot information table

iD	Booked time(datetime)	Slot 1	Slot 2	Slot ···
	14:30-26-Nov-2019 to 14:50-26-Nov-2019	Booked		

Figure 3.8: Slot booking table

iD	Timestamp	Slot 1 size	Slot 1 size
		Size	

Figure 3.9: Current slot size table

iD	Timestamp	Slot 1 speed	Slot 2 speed	Slot ···
		Speed		

Figure 3.10: Current slot speed table

Figure 3.11, 3.12, 3.13 and 3.14 below show the tables that should be stored in the overall database. Similar to the local database, server ID is the primary key of all of the tables and relational database is more suitable here. Both local and overall database have the same structure and data, however, they are used differently. The local database is mainly used for booking the slot information while overall databases focus on storing the real-time information. Two databases are incorporated for faster and more precise data retrieval.

iD (same as server)	timestamp	Slot 1···N
		Occupied/not

Figure 3.11: Current slot information table

iD	datetime period	Slot 1···N
	14:30-26-Nov-2019 to 14:50-26-Nov-2019	Booked

Figure 3.12: Slot booking table

iD	Slot 1···N
	Size in meters

Figure 3.13: Current slot size table

iD	Slot 1···N speed
	Speed

Figure 3.14: Current slot speed table

3.4 User Case Scenario

3.4.1 Booking Journey

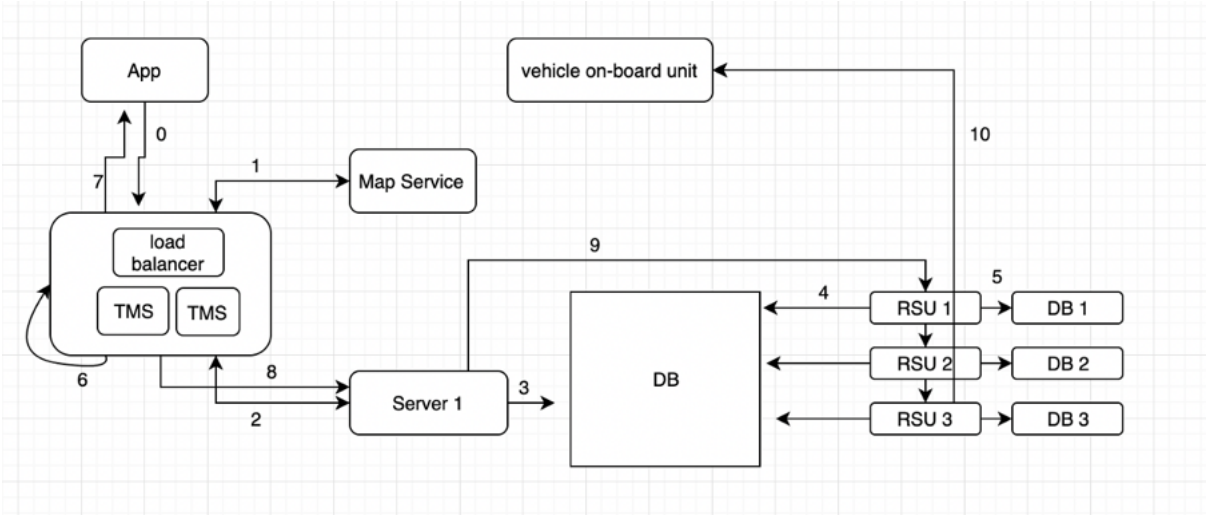


Figure 3.15: Book a journey

Figure 3.15 above shows the process for users to book a journey before joining the motorway. Following step 0 and 1 in this figure, users will input their starting time, starting position as well as destination to the App and this information will be passed to the load balancer. The load balancer will distribute the requests to TMS based on its IP address and TMS will send requests to Map Service which will generate multiple routes and send responses back to TMS. Following Step 2, TMS will start querying corresponding road servers which will further query the overall database to check if the slot is available for that particular period. As shown in Step 6, these steps will keep repeating until optimal route suiting users' priorities is found. After this, road servers will update the overall database and send requests to the RSU which will write to the local database to book the slot. In step 7, the optimal

route will be sent back to the users and route information will be displayed in the App.

Step 4, 5 and 10 are happening at all times. The RSU will continuously update both the local and overall databases. At the same time, it will keep gathering information which is broadcasted by the OBUs.

3.4.2 Driving on Road

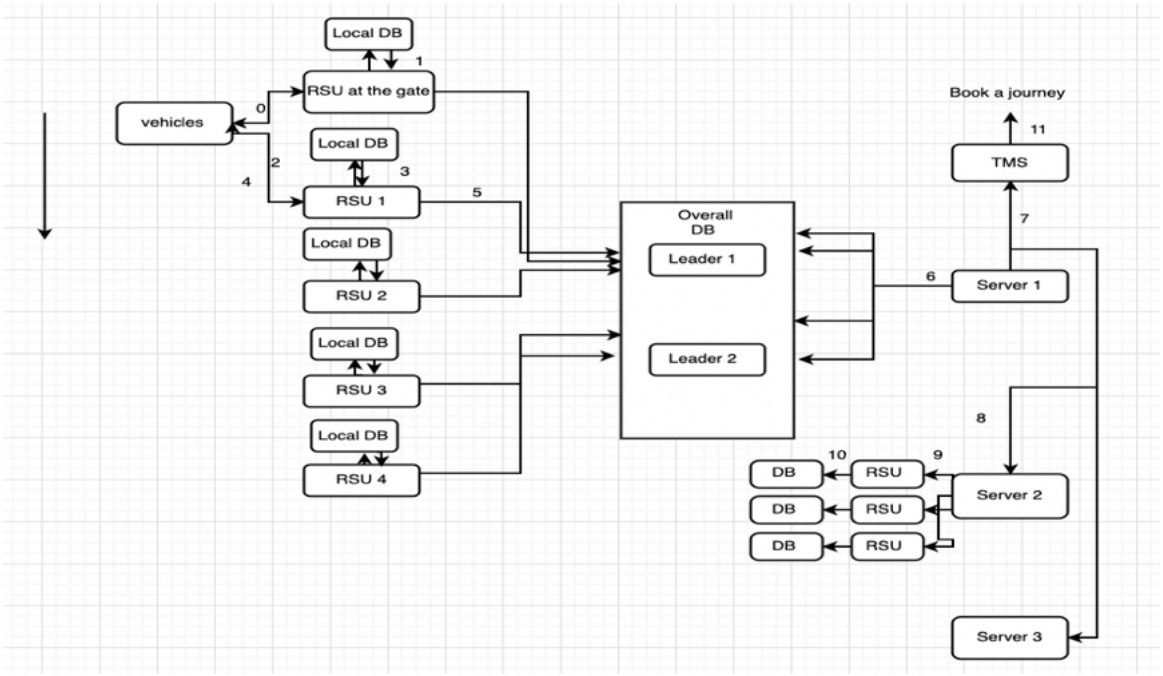


Figure 3.16: Driving on the road

Figure 3.16 above shows an overview of the system when vehicles are running on the road. Before vehicles emerge into the motorway, the RSU at the gate will check its local database to see whether the vehicle has booked slots or not. Upon the detection of a valid booking, the vehicle will join the motorway and start travelling along with the trajectories defined by slots. If no booking is found, it will follow Step 11 to book a journey first. As shown in Step 5, RSU will update the overall database and its local database at the same time while vehicles are travelling on the road. The overall database will follow the leader-follower replication model discussed in Chapter 2. Thus, RSU will only write to the leader nodes and these nodes will propagate the changes to followers. Step 8 shows how the system works when the vehicle is turning to another road. The road server that is currently in charge of this vehicle will inform the server on the next road.

4 Implementation

One prototype of the design was fully implemented in this project with the help of traffic simulator SUMO. Python is the only programming language used in this project and XML is used for slot information storage.

This chapter will give an in-depth description and explanation of the design and implementation process of the prototype. Implementation of the prototype involves three development phases, which will be discussed individually in the following sections. Phase one focused on the initial environment setup and simple components implementation including RSU and OBU, with the aim of getting each component of the system working as a whole. Phase two added in pre-defined slot information and more functionalities, such as vehicle control in OBUs. Phase three focused on scaling up the implementation.

All phases above contribute towards a fast and efficient design prototype, which fulfils the four key characteristics of good TMSs mentioned in chapter 3.

4.1 Phase One

4.1.1 Environments and Software Setup

A map is needed for simulating the traffic network in SUMO. A graphical network editor called NETEDIT (56) is a tool provided by SUMO for users to draw the map based on their needs. The map in NETEDIT can be exported in the format of the SUMO-net file which can be read by SUMO. NETEDIT is a GUI-application and it is designed to be running under the Linux/Windows operating system. To run any GUI-applications under macOS, XQuartz, a component of the X-Windows system, has to be installed first. By using this, NETEDIT and SUMO can be used smoothly in macOS.

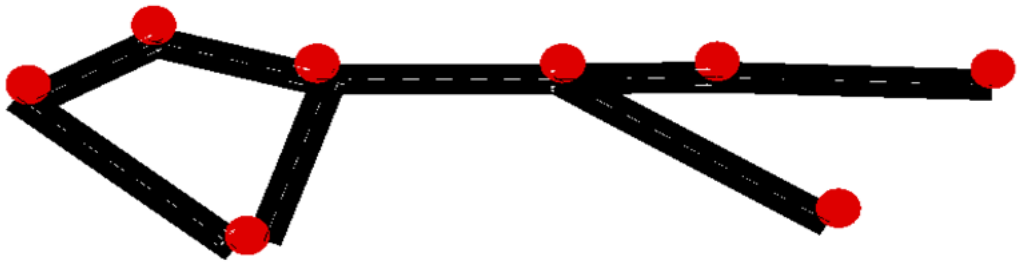


Figure 4.1: Initial map in NETEDIT

As shown in figure 4.1, a map with eight edges and seven junctions was made in NETEDIT as the first version. Each of the edges has two lanes, one direction and a specific edge ID. This map was exported in the format of net.xml (SUMO-net) file which was passed as an input parameter in the SUMO configuration file called sumo.sumocfg. A route file is also required in SUMO for specifying vehicles, their properties and routes taken by vehicles. In figure 4.2, a vehicle with ID 0 has the route starting from the edge -genE36 and finishing at the edge -gneE38. Similar to net.xml, sumo.sumocfg also took the route file as an input parameter. Figure 4.3 shows how net.xml and route file were handled in sumo.sumocfg. Following these previous steps, the sumo.sumocfg file can be opened in SUMO and vehicles were able to run in the virtual environment.

```
<vehicle id="0" depart="0.00">
    <route edges="-gneE36 -gneE25 gneE33 gneE37 -gneE38"/>
</vehicle>
```

Figure 4.2: Route file

```
<input>

    <net-file value="newedit.net.xml"/>
    <route-files value ="newedit.rou.xml"/>
    <additional-files value = "additional.xml"/>
</input>
```

Figure 4.3: sumo.sumocfg file

4.1.2 RSU

As mentioned in Chapter 3, RSUs are designed to gather information from the vehicles. For this prototype, RSUs were implemented for sending slot information to OBUs and setting up communication with them. Phase One only focused on getting RSUs talking to OBUs.

In the prototype, TCP was used for establishing and maintaining the conversation between RSUs and OBUs. It was chosen over UDP for its higher reliability and its ability to guarantee the flow and integrity of data (57). Python provides the socket module which can be used for TCP socket programming. The most common type of socket applications are client-server applications. In this project, RSU acted as a client and OBU acted as a server. Figure 4.4 below shows the process of setting up TCP connections between the server and the client.

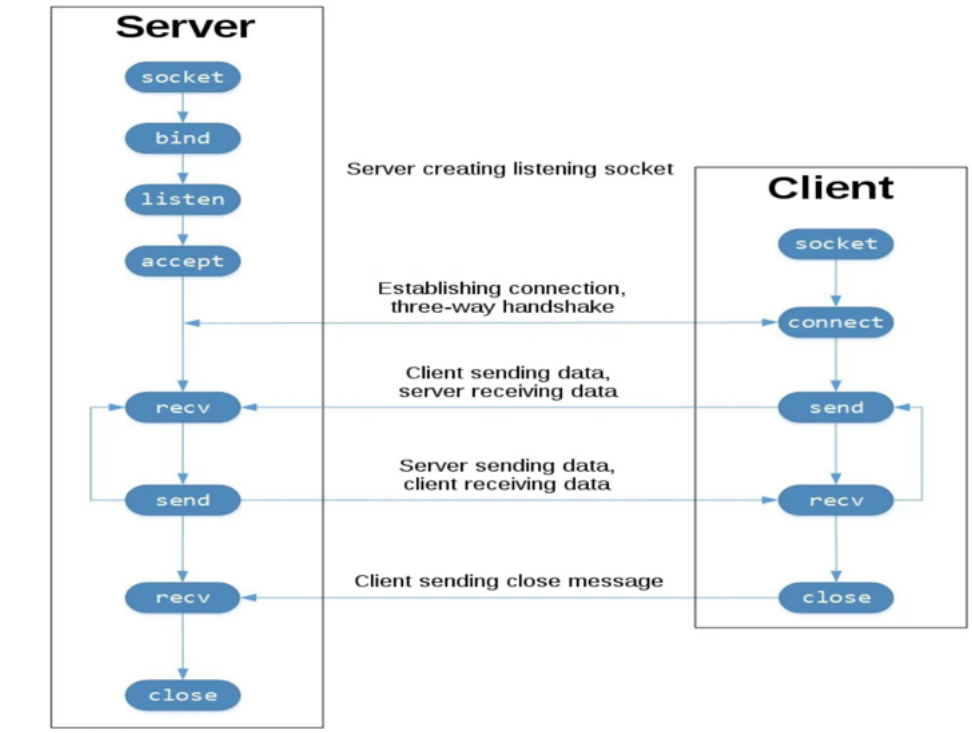


Figure 4.4: TCP three-way handshaking (10)

Figure 4.5 and figure 4.6 below show the process of RSU and OBU setting up the connection respectively on their sides. As mentioned earlier, this TMS should be able to handle a large number of users, meaning that the size of data is large and the data format is complex. Python provides a library called pickle that is specially designed for handling such type of data. Figure 4.7 and 4.8 show how data is serialized and deserialized in RSUs and OBUs. In this phase, only strings with different sizes were used to test the TCP connection between RSUs and OBUs.

```
import socket
HOST = "127.0.0.1"
PORT = 8080
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
```

Figure 4.5: RSU TCP connection

```
import socket
HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 8080        # Port to listen on (non-privileged ports are > 1023)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(2)

conn, addr = s.accept()
```

Figure 4.6: OBU TCP connection

```
request = "what's the position?"
new_request = pickle.dumps(request)
s.send(new_request)
```

Figure 4.7: Data serialization in RSUs

```
data_ = conn.recv(4096)
data = pickle.loads(data_)
```

Figure 4.8: Data deserialization in OBUs

4.1.3 OBU

OBUs in this prototype were in charge of receiving slot information from RSUs, associating slots to vehicles and controlling them. In Phase one, OBUs were implemented for building TCP connection with RSUs, which has been set up in previous content. OBUs were also implemented in this phase for controlling vehicles.

To control vehicles in SUMO, firstly SUMO_HOME directory had to be set on the python load path, which means the environment variable SUMO_HOME should be set before running the python script. A library called TraCi was used in this study, which provides an interface for Python script to control vehicles in SUMO. It can also be used to retrieve information about vehicles and road (58). This library also needed to be imported into Python to enable the TraCi commands. Figure 4.9 shows the setup of the environment variable and TraCi.

```
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
```

Figure 4.9: SUMO environment setup

To start the SUMO simulation and connect it to the Python script, the code shown in figure 4.10 was needed. After connecting to the simulation, various commands can be sent within the while loop. The official website (59) provides various TraCi APIs for value retrieval, lane changing, state-changing and subscriptions. In this phase, only the commands for retrieving vehicle information and changing vehicles' lanes were used to test system's functionalities.

```
import traci
traci.start([sumoBinary, "-c", "sumo.sumocfg",
                     "--tripinfo-output", "tripinfo.xml"])
step = 0
while step < 1000:
    traci.simulationStep()
    if traci.inductionloop.getLastStepVehicleNumber("0") > 0:
        traci.trafficlight.setRedYellowGreenState("0", "GrGr")
    step += 1

traci.close()
```

Figure 4.10: Starting simulation

4.2 Phase Two

4.2.1 Define Slot

In the design, slot information is stored in both the local and overall database, which can be queried by RSUs and road servers respectively. In Chapter 3, the relational database was determined to be the best choice for retrieving information like speed, data structure and data size. However, this prototype is of small scale compared with the previous design. Therefore XML was used in the prototype to store the slot information. The slot.xml for each vehicle has the following structure shown in figure 4.11. Tag laneiD_slot defines the

slot on one edge. Each slot's position, lane, speed and starting time are all defined within the tag slot_ID. In this prototype, each vehicle would only be assigned one slot on one lane and its slot would change if it turned to another lane or edge.

```
233     <slot>
234     <laneiD_slot>
235         <slot_0></slot_0>
236         ...
237         <slot_4></slot_4>
238     </laneiD_slot>
239     <laneiD_slot></laneiD_slot>
240     </slot>
```

Figure 4.11: slot.xml file

In figure 4.12 below, a slot named E36_0 has a pre-defined trajectory: slot_0 -> slot_1 (the number only represents the position rather than slot ID). It starts from position (-48, 55) and ends at position (-47, 57.6). The slot also has the fixed speed 120 and only runs on lane E36.

Slot files were named using the string "slot_" and the corresponding vehicle ID. For example, the slot information for vehicle 0 should be named slot_0.xml.

```
<slot>
    <E36_0>
        <slot_0>
            <time time="0"></time>
            <pos position="(-48, 55)"></pos>
            <lane lane="E36"></lane>
            <speed speed="120"></speed>

        </slot_0>
        <slot_1>
            <time time="1"></time>
            <pos position="(-47, 57.6)"></pos>
            <lane lane="E36"></lane>
            <speed speed="120"></speed>

        </slot_1>

    </E36_0>
```

Figure 4.12: slot.xml file

As mentioned in Phase One, all the vehicles' routes were defined in rou.xml file. In the prototype, trajectories defined by slots would overwrite the routes in rou.xml when sending TraCi commands from OBUs to SUMO.

4.2.2 RSU

RSUs were responsible for reading the slot information and passing it to OBUs. The development of RSU in this phase focused on getting slot information from slot.xml file, reorganise them and sending them to OBUs.

XML is an inherently hierarchical data format. The most natural way to present it would be using a tree data structure. Python provides the ElementTree XML API (60) for reading the XML file. Figure 4.13 below defined a class called Slot where the ElementTree API helps to read the slot XML file and store slot information. Line 39 parsed the XML file and created an element tree where the root has a tag and a dictionary of attributes. As shown inline 41, the root had children nodes over which we can iterate. Starting from line 41, the code started to iterate over all the information of one slot and append it into a dictionary for readability and faster information retrieval speed.

```
37  class Slot(object):
38      def __init__(self, slot_no, edge_id, number):
39          self.root = ET.parse('slot_'+number+'.xml').getroot()
40          self.slot_info = {}
41          for slot_1 in self.root.findall('./' + edge_id + '/' + slot_no + '/'):
42
43              value = slot_1.attrib
44              count = 0
45              for i, j in value.items():
46                  li = []
47                  if count == 1:
48                      j = j[1:-1]
49                      for x in j.split(","):
50                          li.append(float(x))
51                  else:
52                      li.append(j)
53                  self.slot_info[i] = li
54                  count += 1
55
```

Figure 4.13: Read in the XML file in Python script

Only three vehicles were defined in this phase, therefore three for loops were needed to read three slot.xml files. Figure 4.14 shows the implementation for one of the loops. In this code snippet, the Slot class was initiated 5 timesm which represented there were 5 edges defined in this vehicle's trajectory. After getting all the slot information of one vehicle, a variable called route was utilised to store all the vehicles' information. Route variable was of type list because the list is easy to be serialized and deserialized using pickle, and it is also mutable. All of the slot information was packed and serialised in RSUs and then sent to OBUs using TCP.

```
85      route = []
86      for i in route_1:
87          route_list = []
88          for y in range(5):
89              slot = Slot("slot_" + str(y), i, '0')
90              route_list.append(slot.slot_info)
91
92          trajectory_E36 = []
93          for x in route_list:
94              route.append([i, x['position']])
```

Figure 4.14: Append slot information to route list

4.2.3 OBU

OBUs in this phase is implemented to get the slot information from the RSUs, associate it to vehicles and send corresponding commands to them.

As mentioned in section 4.1.1, slot information was all stored in a list called route. In OBUs, it can be easily deserialised and manipulated using pickle. Figure 4.15 below shows the process of associating vehicles to slots using the information extracted from the slot file. Line 88, 95 and 96 defined the positions that the vehicles should follow in the following step.

```
86          if step < 30:
87              info0 = l_0[step]
88              x0, y0 = float(info0[1][0][1:-1].split(',')[0]), float(info0[1][0][1:-1].split(',')[1])
89              edge_ID_0 = info0[0][:3]
90
91              if step < 15:
92                  info1 = l_1[step]
93                  info2 = l_2[step]
94
95                  x1, y1 = float(info1[1][0][1:-1].split(',')[0]), float(info1[1][0][1:-1].split(',')[1])
96                  x2, y2 = float(info2[1][0][1:-1].split(',')[0]), float(info2[1][0][1:-1].split(',')[1])
97
98                  edge_ID_1 = info1[0][:3]
99                  edge_ID_2 = info2[0][:3]
00
01              elif 15 <= step < 20:
02                  info1 = l_1[step]
03                  x1, y1 = float(info1[1][0][1:-1].split(',')[0]), float(info1[1][0][1:-1].split(',')[1])
```

Figure 4.15: Allocate a vehicle to its slot

TraCi provides various commands for controlling the vehicles. Two of them were used in the prototype. One is moveToXY(self, vehID, edgeID, lane, x, y, angle=-1073741824.0, keepRoute=1) and another is changeLane(self, vehID, laneIndex, duration). The first command was used to keep sending slot positions to vehicles running in SUMO, which can overwrite the route defined in rou.xml. All of the attributes are compulsory for using moveToXY command. Considering there were many vehicles, edges and lanes in the traffic network, multiple if-else loops were used to distinguish the edgeID and send the commands

to vehicles accordingly. This was done in the method called moveVeh, as shown in figure 4.16. In this figure, the moveVeh function differentiates the edge E33 and edge E25. If the vehicle is going to run on edge E33, the code will go to edge E33's if statement and the edge ID and vehicle ID would be passed to moveToXY() as parameters to move the corresponding vehicles. This method differentiated all the edges and sent commands within each edge's if statement.

```
def moveVeh(edge_ID, id, x,y, info):
    if edge_ID == 'E33':
        print("going on E33")
        traci.vehicle.moveToXY(id, 'gne' + edge_ID, 0, x, y, 115, 1)
        print("veh={} speed={}".format(id, traci.vehicle.getSpeed(id)))
    elif edge_ID == 'E25':
        print("going on E25")
        if info[0][3:] == "_1":
            if route_map.index(info) == 10:
                traci.vehicle.changeLane(id, 1, 4000.00)
                print("veh={} speed={}".format(id, traci.vehicle.getSpeed(id)))
            elif step == 14:
                traci.vehicle.changeLane(id, 0, 1000.00)
```

Figure 4.16: Distinguish edge ID and send commands

The second command changeLane was used to change the vehicles' lane. As shown in figure 4.16 above, the vehicle was allocated to switch from lane 0 to lane 1 if that particular vehicle is running on edge E25.

SUMO only recognises the current time, which means only the last command will be used while computing the simulation step. Therefore the moveVeh function for one vehicle could only be called once in each step. To run multiple vehicles simultaneously, the code in figure 4.17 was executed. The figure clearly shows that in each simulation step, the control commands could be sent to different vehicles sequentially to achieve the simultaneity.

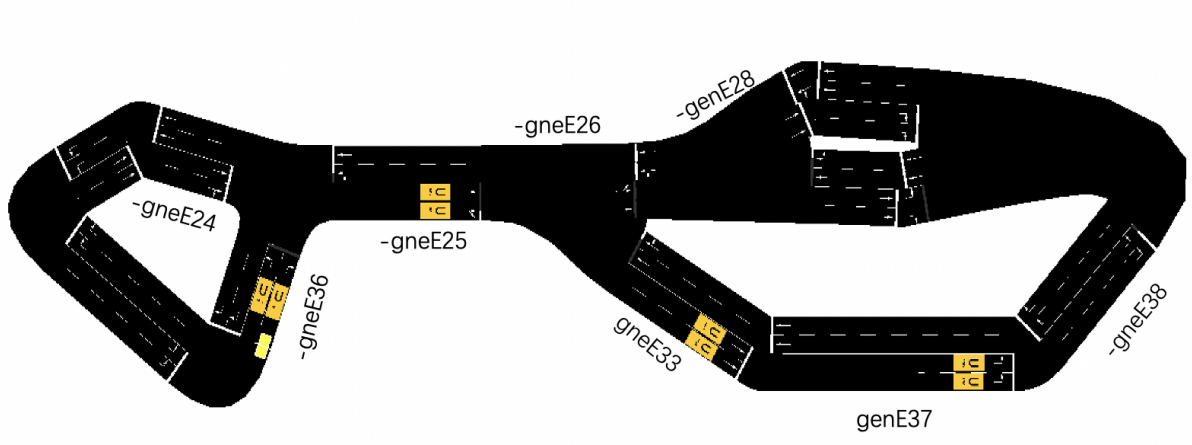
```
if step< 30:
    print("step < 30")
    moveVeh(edge_ID_0, '0', x0, y0, info0)
    if step < 15:
        moveVeh(edge_ID_1, '1', x1, y1, info1)
        moveVeh(edge_ID_2, '2', x2, y2, info2)
    elif step > 15:
        moveVeh(edge_ID_1, '1', x1, y1, info1)
```

Figure 4.17: Move multiple vehicles simultaneously

4.3 Phase Three

Phase Three focused on scaling up the prototype. In Phase Two, a functional TMS has already been implemented. However, the map used was not sufficient for a large-scale design due to its limitation in size. Therefore more edges were added into the traffic network in this

phase, with all edges changed to be bi-directional. Figure 4.18 below shows the final traffic map and vehicle routes implemented in this prototype.



Veh 0: route edges="-gneE36 -gneE25 gneE33 gneE37 -gneE38"
Veh 1: route edges="-gneE24 -gneE25 gneE33 gneE37
Veh 2: route edges="-gneE25 -gneE26 -gneE28"
Veh 3: route edges="-gneE25 gneE24 -gneE23 –gneE35 -gneE36"

Figure 4.18: Map used in the implemented prototype

5 Evaluation

This chapter focuses on the evaluation of the implemented prototype. Section 5.1 details the results obtained from the prototype, presenting images of building traffic network, slot trajectories and code snippets. Section 5.2 evaluates the differences between the design and the implemented prototype. Section 5.3 concludes this chapter, discussing the work that was not done in this project and future work.

5.1 Results

As shown in figure 5.1, there are 4 vehicles (yellow triangle) running in the traffic network. This final map consists of 12 edges and 12 junctions. All edges are bi-directional and have two lanes.

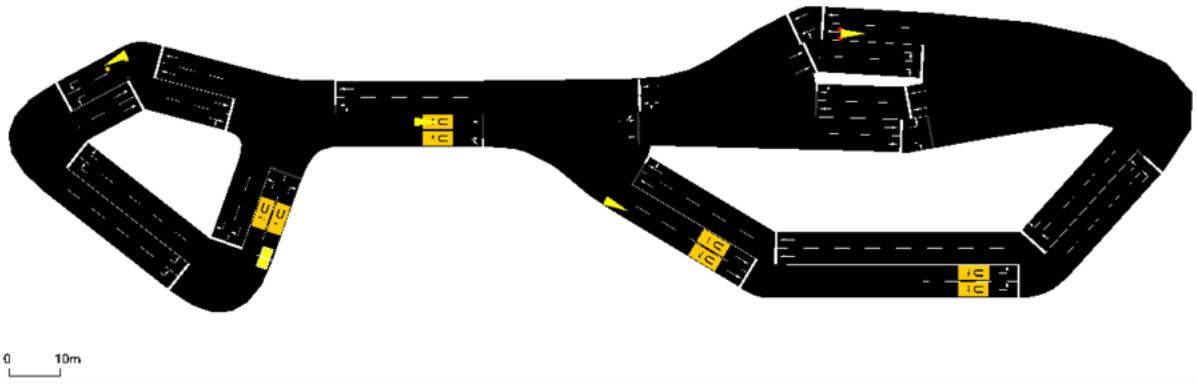


Figure 5.1: Vehicles running in the SUMO traffic network

Accordingly, four slot.xml files were defined and each of them represented one vehicle's trajectory during their journey on the road. The trajectory of one vehicle was made up of slots on different edges. Figure 5.2 below shows the trajectory of vehicle 0. The rectangles marked with number 1 to 7 represent the booked slot on each edge. Rectangles with lighter colours demonstrate the trajectory of slots on each edge. In this figure, a vehicle with ID 0 travelled within the slot 1 on lane 0 of edge -gneE36, slot 2 on lane 0 of edge -gneE25, slot 3 on lane 1 of edge -gneE25, slot 4 on lane 0 of edge -gneE25, slot 5 on lane 0 of edge

gneE33, slot 6 on lane 0 of edge gneE37 and slot 6 on lane 0 of edge -gneE38. Note that it changed the lane while travelling on edge -gneE25, thus the slot was changed correspondingly during that period. Other vehicles' trajectories are shown in figure 5.3. Vehicle 1 (route_2) had the route starting from lane 0 of edge -gneE24 and ending at lane 0 of edge gneE37. Vehicle 2 (route_3) had the route starting from lane 0 of edge -gneE25 and finishing at lane 0 of edge gneE28. Vehicle 3 (route_4) had the route starting from lane 0 of edge -gneE36 and ending at lane 0 of edge -gneE38.

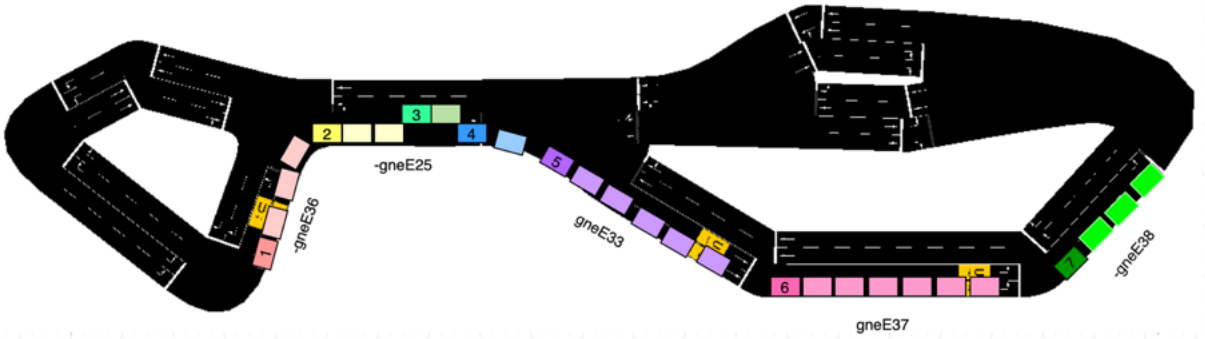


Figure 5.2: Trajectory of vehicle 0

```
route_1 = ['E36_0', 'E25_0', 'E25_1', 'E33_0', 'E37_0', 'E38_0']
route_2 = ['E24_0', 'E25_0', 'E33_0', 'E37_0']
route_3 = ['E25_0', 'E26_0', 'E28_0']
route_4 = ['E36_0', 'E25_0', 'E25_1', 'E33_0', 'E37_0', 'E38_0']
```

Figure 5.3: Trajectories of four vehicles

Slot information was read in RSUs and stored as route information in the list, which is a Python data structure. This information was passed from RSUs to OBUs, which then associate the slot information to vehicles using TraCi.

Using this system, users were allocated a pre-defined route containing slot information before getting on the road. The system would keep sending commands to multiple vehicles simultaneously to move them while the vehicles are travelling on the motorway. These vehicles will strictly follow their trajectories and stay within their pre-defined slot during the journey.

5.2 Comparison between Design and Prototype

The final prototype only implemented part of the design, therefore this section will focus on the differentiation between implemented part and the full scope of the design.

5.2.1 Components

The implemented prototype is composed of three components including RSUs, OBUs and slot information. SUMO provides the virtual traffic network for the prototype to run the simulation. Slot information is defined in the format of XML and it is used for storing the slot's position, speed, ID and starting time. Each vehicle has one slot file as its trajectory information. As for the RSU, it is implemented to be able to read slot information, reformat the data, set up the TCP connection with OBU and send the slot information to OBU. The OBU is responsible for receiving the slot information from RSU, extracting the information and associating them to vehicles based on the vehicle ID. It also takes advantages of TraCi interface to constantly send commands to vehicles running in SUMO and control them.

The overall design consists of App, load balancer, TMS centres, servers, RSUs, OBUs, the overall database and the local databases. In the design, RSUs are partitioned and distributed on the road every 1000 meters while the prototype didn't consider the replication because of its small scale. OBUs in the design represent the vehicles. However, in the prototype, SUMO was used to simulate the vehicles. Thus, OBUs only acted as an interface for TraCi API to control the vehicles in SUMO. As for the slot, the initial design determined the slot size to be 100 meters, while in the prototype, each edge was divided into 5 parts and the length of each part was the length of slots. The XML file was used in prototype instead of the relational database in the design because the data size of the prototype is small and the data structure can be easily stored in the format of XML.

5.2.2 System Functionalities

In Chapter 3, two scenarios were discussed. First one is for booking a journey and the second is for vehicles driving on the road. The overall system is designed to be able to handle both scenarios. However, in the implemented prototype, the system was not responsible for booking a slot for the users based on the road capacity and slot availability. These routes were generated randomly and the prototype would only allocate the pre-defined routes to vehicles. While vehicles are running on the motorway, the system in this prototype would keep sending commands to vehicles. However, for the design, TMS will frequently query the database to check the traffic congestion and accidents, and send move commands to vehicles only if no accidents are happening on the road.

5.3 Future Work

As addressed in Chapter 3, high scalability, reliability, availability and low latency are the requirements of this TMS. Even though the prototype was fully implemented, the testing of these features is yet to be done. Thus, these testing work will all be propagated to the

future work for a system with a larger scale.

The current prototype is of small scale and it is necessary to scale up the overall system to adapt to the real-world situation in the future. There are many components which haven't been implemented in the prototype, such as databases and road servers. These are all essential for building up a large-scale system and all the unimplemented components should all be involved in the system implementation in the future.

6 Conclusion

This research aimed to develop a large-scale traffic management system using a slot-based driving approach to remotely control vehicles and ease the traffic congestion on the motorway. Based on the quantitative and qualitative analysis of prior work of four domains including TMS, vehicular communication, large-scale system design and slot-based driving, it can be concluded that such large-scale TMS design was proposed to be able to meet the system requirements and one prototype was implemented to achieve some of the required system functionalities. The implementation of prototype allows the TMS to assign users the routes defined by slot trajectories and to control the vehicles while they are running on the motorway.

There are a few requirements for the overall TMS design including high scalability, availability, reliability and low latency. Enumerating user cases greatly helps to identify each component's characteristics and functionalities, which can further help to meet the system requirements in the design. In the prototype, instead of using real-world vehicles and sensors, traffic simulator SUMO was used to provide the virtual traffic environments and to simulate the vehicles. It greatly reduces the implementation cost and increases development efficiency. An open-source Python script (61) was also used to generate route.xml file to prevent route overlap which will potentially cause traffic congestion.

Based on these conclusions, some work is suggested to be propagated into the future development process. Scaling up the system is the next step for developing this large-scale TMS. The current prototype is of small scale and it only simulates a few components. In future work, simulating more real-world components such as in-road sensors and cameras will greatly improve the system's accuracy of detecting traffic congestion and accidents. The current prototype also lacks testing for requirements that was were used previously. Therefore it is necessary to add testing in the future work to ensure the quality, consistency and performance of the system. Furthermore, the slot-based driving approach should also be considered to be developed to adapt to various environment conditions in the future.

The design demonstrated by this research project proposed a way to implement the traffic management system using a slot-based driving approach, which is yet to be developed in a real-world situation. It provides the approach to combine both advanced TMS and

autonomous driving methodology to reduce the traffic congestion caused by human-driving. The prior slot model is utilised in this design and it is also used to define the trajectory of vehicles (13). The use and redevelopment of this prior model greatly help to standardise vehicles' behaviours, which potentially helps to tolerate human-driver mistakes. This research project represents an initial exploration into integrating TMS and slot-based driving, with the overall aim of developing an advanced large-scale distributed TMS for building a smart traffic environment with higher efficiency and lower accident rate.

Bibliography

[1] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[2] Draft standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks –specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 7: Wireless access in vehicular environments. *IEEE Unapproved Draft Std P802.11p/D9.0, July 2009*, 2009.

[3] clemson vehicular electronics laboratory: dedicated short range communications_2020, 2020. URL `https://cecas.clemson.edu/cvel/auto/systems/DSRC.html`. [Online; accessed 06-March-2020].

[4] J. B. Kenney. Dedicated short-range communications (dsrc) standards in the united states. *Proceedings of the IEEE*, 99(7):1162–1182, 2011.

[5] 2020. URL `https://www.cablefree.net/wirelesstechnology/4glte/lte-network-latency/`. [Online; accessed 06-March-2020].

[6] Thiago S Gomides, Massilon L Fernandes, Fernanda Sumika Hojo de Souza, Leandro Villas, and Daniel L Guidoni. Fire-nrd: A fully-distributed and vanets-based traffic management system for next road decision. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 554–561. IEEE, 2019.

[7] Soufiene Djahel, Ronan Doolan, Gabriel-Miro Muntean, and John Murphy. A communications-oriented perspective on traffic management systems for smart cities: Challenges and innovative approaches. *IEEE Communications Surveys & Tutorials*, 17 (1):125–151, 2014.

[8] Allan M De Souza, Celso ARL Brennand, Roberto S Yokoyama, Erick A Donato, Edmundo RM Madeira, and Leandro A Villas. Traffic management systems: A classification, review, challenges, and future perspectives. *International Journal of Distributed Sensor Networks*, 13(4):1550147716683612, 2017.

[9] 2020. URL https://www.rac.co.uk/drive/advice/learning-to-drive/stopping-distances/. [Online; accessed 6-May-2020].

[10] Real Python. Socket programming in python (guide) – real python, 2020. URL https://realpython.com/python-sockets/. [Online; accessed 6-May-2020].

[11] Sameer Tikar and A. D. Jadhav. Can to wi-fi interface for vehicles. 2013.

[12] See Foxe. Over 5,000 accidents on m50 since 2017, 2020. URL https://www.thejournal.ie/m50-crashes-foi-4278822-Oct2018/. [Online; accessed 06-March-2020].

[13] Vinny Cahill, Aline Senart, Douglas C Schmidt, Stefan Weber, Anthony Harrington, Barbara Hughes, Kulpreet Singh, and Mélanie Bouroche. The managed motorway: real-time vehicle scheduling: a research agenda. In *Proceedings of the 9th workshop on Mobile computing systems and applications*, pages 43–48, 2008.

[14] Remi Tachet, Paolo Santi, Stanislav Sobolevsky, Luis Ignacio Reyes-Castro, Emilio Frazzoli, Dirk Helbing, and Carlo Ratti. Revisiting street intersections using slot-based systems. *PloS one*, 11(3), 2016.

[15] Juan Guerrero-Ibáñez, Sherali Zeadally, and Juan Contreras-Castillo. Sensor technologies for intelligent transportation systems. *Sensors*, 18(4):1212, 2018.

[16] Sape Mullender et al. *Distributed systems*, volume 12. acm press United States of America, 1993.

[17] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[18] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Patterns for three-tier client/server applications. *Proceedings of Pattern Languages of Programs (PLoP'96)*, 4 (6), 1996.

[19] John M Gallaugher and Suresh C Ramanathan. Choosing a client/server architecture: a comparison of two-and three-tier systems. *Information Systems Management*, 13(2): 7–13, 1996.

[20] Yingsong Hu, Liwen Peng, and Chubin Chi. Design technology of three-tier architecture on web application based on .net [j]. *Computer Engineering*, 8, 2003.

[21] are you ready to consolidate databases into database clouds?, 2020. URL https://www.oracle.com/database/. [Online; accessed 06-March-2020].

[22] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, 1997.

[23] URL https://www.mysql.com/. [Online; accessed 06-March-2020].

[24] Yoshinori Matsunobu. Semi-synchronous replication at facebook. *Accessed: Jun*, 25: 2019, 2014.

[25] 2020. URL https://www.ibm.com/support/knowledgecenter/en/SSQNUZ_2.5.0/wsj/analyze-data/hadoop-environments.html. [Online; accessed 06-March-2020].

[26] 2020. URL https://www.ibm.com/analytics/hadoop/mapreduce. [Online; accessed 06-March-2020].

[27] Yas Alsultanny. Database management and partitioning to improve database processing performance. *Journal of Database Marketing & Customer Strategy Management*, 17 (3-4):271–276, 2010.

[28] data partitioning with chunks — mongodb manual_2020, 2020. URL https://docs.mongodb.com/manual/core/sharding-data-partitioning/. [Online; accessed 06-March-2020].

[29] Lars Hofhansl, Lars Hofhansl, and View profile. Introduction to hbase, 2020. URL http://hadoop-hbase.blogspot.com/2011/12/introduction-to-hbase.html. [Online; accessed 06-March-2020].

[30] 2020. URL https://docs.mongodb.com/manual/reference/command/filemd5/. [Online; accessed 06-March-2020].

[31] 2020. URL https://www.agilelab.it/secondary-indexing-on-hbase-a-k-a-nosql-is-good-but-the-world-is-not-key-value/. [Online; accessed 06-March-2020].

[32] Rusty Klophaus, Pavel Hardak, Dorothy Pults, Dorothy Pults, and Dorothy Pults. Secondary indexes in riak, 2020. URL https://riak.com/posts/technical/secondary-indexes-in-riak/index.html. [Online; accessed 06-March-2020].

[33] Bhawna Dhawan and Tanu Preet Singh. Efficient data dissemination techniques in vanets: a review. *International Journal of Computer Applications*, 116(7), 2015.

[34] Daniel Jiang and Luca Delgrossi. Ieee 802.11 p: Towards an international standard for wireless access in vehicular environments. In *VTC Spring 2008-IEEE Vehicular Technology Conference*, pages 2036–2040. IEEE, 2008.

[35] Zhigang Xu, Xiaochi Li, Xiangmo Zhao, Michael H Zhang, and Zhongren Wang. Dsrc versus 4g-lte for connected vehicle applications: A study on field experiments of vehicular communication performance. *Journal of Advanced Transportation*, 2017, 2017.

[36] Q. Yang, L. Wang, W. Xia, Y. Wu, and L. Shen. Development of on-board unit in vehicular ad-hoc network for highways. In *2014 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 457–462, 2014.

[37] Yanwen Wu and Qi Luo. *High Performance Networking, Computing, Communication Systems, and Mathematical Foundations: International Conferences, ICHCC 2009-ICTMF 2009, Sanya, Hainan Island, China, December 13-14, 2009. Proceedings*, volume 66. Springer Science & Business Media, 2010.

[38] B. Aslam, F. Amjad, and C. C. Zou. Optimal roadside units placement in urban areas for vehicular networks. In *2012 IEEE Symposium on Computers and Communications (ISCC)*, pages 000423–000429, 2012.

[39] S. Sou and O. K. Tonguz. Enhancing vanet connectivity through roadside units on highways. *IEEE Transactions on Vehicular Technology*, 60(8):3586–3602, 2011.

[40] Phil Goodwin. The economic costs of road traffic congestion. 2004.

[41] 2020. URL `https://ec.europa.eu/transport/media/news/2016-09-16-european-mobility-week_en`. [Online; accessed 06-March-2020].

[42] Guy A Boy. *The handbook of human-machine interaction: a human-centered design approach*. CRC Press, 2017.

[43] Jonathan Turner. New directions in communications(or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, 1986.

[44] Ricardo Morla. Sentient future competition: Vision of congestion-free road traffic and cooperating objects. 2006.

[45] Nishkam Ravi, Stephen Smaldone, Liviu Iftode, and Mario Gerla. Lane reservation for highways (position paper). In *2007 IEEE Intelligent Transportation Systems Conference*, pages 795–800. IEEE, 2007.

[46] Dan Marinescu, Jan Čurn, Mélanie Bouroche, and Vinny Cahill. On-ramp traffic merging using cooperative intelligent vehicles: A slot-based approach. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 900–906. IEEE, 2012.

[47] Wikipedia contributors. Interchange (road) — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Interchange_(road)&oldid=955099555. [Online; accessed 6-May-2020].

[48] Magnus Sellén. Average car length - list of car lengths - mechanic base, 2020. URL https://mechanicbase.com/cars/average-car-length/. [Online; accessed 06-March-2020].

[49] Wikipedia contributors. Http compression — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=HTTP_compression&oldid=951714277. [Online; accessed 6-May-2020].

[50] URL https://www.f5.com/services/resources/glossary/load-balancer. [Online; accessed 6-May-2020].

[51] What is load balancing? how load balancers work. URL https://www.nginx.com/resources/glossary/load-balancing/. [Online; accessed 6-May-2020].

[52] Wikipedia contributors. Load balancing (computing) — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Load_balancing_(computing)&oldid=951440733. [Online; accessed 8-May-2020].

[53] 2020. URL http://nginx.org/en/docs/http/load_balancing.html. [Online; accessed 6-May-2020].

[54] 2020. URL https://www.brianstorti.com/replication/. [Online; accessed 6-May-2020].

[55] 2020. URL https://blog.cloud66.com/3-tips-for-selecting-the-right-database-for-your-app/. [Online; accessed 6-May-2020].

[56] 2020. URL https://sumo.dlr.de/docs/NETEDIT.html. [Online; accessed 6-May-2020].

[57] Margaret Rouse. What is tcp (transmission control protocol)?, Apr 2020. URL https://searchnetworking.techtarget.com/definition/TCP. [Online; accessed 6-May-2020].

[58] URL https://sumo.dlr.de/docs/TraCI/Interfacing_TraCI_from_Python.html.

[59] 2020. URL https://sumo.dlr.de/docs/TraCI.html. [Online; accessed 6-May-2020].

[60] 2020. URL
https://docs.python.org/2/library/xml.etree.elementtree.html. [Online; accessed 6-May-2020].

[61] 2020. URL
https://github.com/eclipse/sumo/blob/master/tools/randomTrips.py. [Online; accessed 6-May-2020].