

UNIVERSITY OF SOUTHERN DENMARK

MASTER THESIS

Integrating Vehicles With Smart Traffic Solutions

Simon Hjortshøj Larsen
Thomas August Lemqvist

{sila114, thlem14}@student.sdu.dk

Supervised by

Jan Corfixen Sørensen
jcs@mami.sdu.dk

In collaboration with

Swarco Technology



June 2, 2019

Abstract

We all know the frustration of driving from one red light to another. Generally traffic lights do not indicate how long time there is until they change light state. If intersections were able to communicate this information to drivers, it would allow them to optimally adjust their speed to pass the intersections, minimizing the amount of needed accelerations. This, according to studies, would result in a decrease of fuel consumption, decreased time spent idle in traffic and a decrease of air pollution from combustion engines [1, 2].

This report documents the development of a system, which integrates a smartphone application with road infrastructure. This is accomplished by creating a microservice based server system, which exchanges real time traffic data with intersections, through a platform called TLEX. This data is passed on to a smartphone application, where it is used to suggest speeds for passing intersections at green light. The application will in the meantime also send data to the infrastructure, which can be used for vehicle detection and vehicle prioritization.

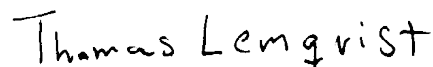
The developed smartphone application is able to provide useful and easily understandable speed recommendations to the drivers. This speed recommendation is visualized to the drivers as a moving area, which they must stay within, to pass the intersection. The server system is able to forward messages from intersections with an average delay of 155 ms, which makes the data current enough to be used for speed recommendations. The server solution is proven to be able to automatically scale to the demands of the users, making it highly available and scalable.

The thesis concludes that it is possible to create a system which integrates with the road infrastructure, to provide drivers with optimal speed recommendations. This could help ease the frustrations of unnecessary accelerations, by reducing the number of stops.

Keywords: [V2X, GLOSA, TLEX, ITS, Microservices, SPaT, MAP, CAM, Service Fabric]

Signatures

It is hereby solemnly declared that the undersigned single-handedly, and independently prepared this report. All citations in the text are marked as such, and the report or substantial parts of it have not previously been subject to assessment.



Simon Hjørtshøj Larsen

date Thomas August Lemqvist

date

Contents

1	Introduction	1
2	Problem	7
3	Related Work	9
4	Requirements	11
5	Analysis	17
6	Design	21
7	Implementation	35
8	Experimental Validation	53
9	Discussion	63
10	Conclusion	67
11	Future Work	69
	Bibliography	70

Acknowledgements

In the process of doing our master thesis we wish to express our gratitude to everybody at Swarco Technology. It has been a pleasure to collaborate and develop a solution at Swarco Technology. A special thank you to Morten Søndergaard for making this collaboration possible and a big thank you to Anders Palm for giving vital support throughout the entire project.

We wish to thank OSS Nokalva for making their ASN.1 software available to us. This software have been instrumental for the development of our system.

We also wish to express our gratitude towards Jan Corfixen Sørensen. He has been our supervisor throughout the project and has given us great advice throughout the project.

Finally, we want to thank all the great people, whom we have lured to proofread our report, and answer questions about the system.

Preface

Reading Guide

This section will describe how various things in the report is setup, in order to make it easier to understand.

The report is divided into chapters. Chapters always start on right sided pages, as it is more reader friendly when printed on double sided paper. This will leave some pages intentionally empty. This property is kept in the non paper version, as it keeps the page numbering the same on each version, making it easier to refer to a single page.

Each chapter starts with a small introduction text, describing the contents of the chapter, followed by a table of contents displaying an overview of the sections in the chapter. The chapters will end with a summary of the contents. This summary is shown by a horizontal line separating it from the contents of the chapter it self.

Figures are numbered and captioned with an identifying text. Figures often break the text and is shown in the center of the page, but it may also be shown on the side of the page, with the text flushing around it, if this suits the figure better.

Tables are numbered and captioned with an identifying text. They always break the text and fills the width of the paper.

Code Snippets are numbered and captioned with an identifying text. A box is drawn around these. They always break the text, and fills the width of the paper. Some of the code in the snippet may be omitted in order to cause brevity, this is shown by a horizontal line across the snippet.

References References are made using the IEEE reference style, numbered based on occurrence in the report. The bibliography can be found at the end of the report.

Acronyms

ASN.1 Abstract Syntax Notation 1.

C-V2X Cellular-V2X.

CAM Cooperative Awareness Message.

DENM Decentralized Environmental Notification Message.

DI Dependency Injection.

DSRC Dedicated Short Range Communication.	SPaT Signal Phase and Time.
DSRC-V2X Dedicated Short Range Communication V2X.	TLC Traffic Light Controller.
ETSI European Telecommunication Standardisation Institute.	TLEX Traffic Light Exchange.
GLOSA Green-Light Optimal Speed Advisory.	TTL Time To Live.
I2X Infrastructure to anything.	TTSC Time To Signal Change.
IoT Internet of Things.	V2I Vehicle to Infrastructure.
ITS Intelligent Transportation System.	V2N Vehicle to Network.
IVI Infrastructure to Vehicle Information.	V2X Vehicle to anything.
JWT JSON Web Token.	VM Virtual Machine.
	WS WebSocket.

Glossary

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.

ASN.1 Abstract Syntax Notation 1 (ASN.1) is a standard for defining data structures that can be serialized and deserialized. According to the ETSI standard, applied to the TLEX platform, all ITS messages are serialized in ASN.1 format.

ASP.Net Core is a cross-platform framework for building .NET Core web apps and services in C#.

C-V2X Cellular-Vehicle To Anything (C-V2X) communication differentiates itself from the IEEE 802.11p V2X by enabling communication based on LTE.

CAM Cooperative Awareness Message (CAM) is an ITS message that contains data about a vehicle, such as speed and heading. It is used by the project to send data to the intersections.

DENM Decentralized Environmental Notification Message (DENM) is an ITS message which contains data about road conditions, such as if the road is slippery or congested.

DI Dependency Injection (DI) is a method whereby one object creates the dependencies of another object.

DSRC Dedicated Short Range Communication (DSRC) is a standardized form of wireless communication specifically designed for automotive use.

DSRC-V2X Dedicated Short Range Communication Vehicle To Anything (DSRC-V2X).

Egress Approach The exit road leading away from an intersection.

Elasticsearch is a search engine that functions as a NoSQL data store and supports near real-time search.

ETSI European Telecommunication Standardisation Institute (ETSI) is a standardization organization in the telecommunication industry. Providing various standards within the domain of smart traffic solutions.

GLOSA Green-Light Optimal Speed Advisory (GLOSA) is a way of presenting a speed recommendation to a driver when approaching an intersection.

I2X Infrastructure to Anything (I2X) communication is passing information from infrastructure, like intersections, smart traffic signs etc, to any receiver.

Ingress Approach The entry road leading up to an intersection.

ITS Intelligent Traffic Solution (ITS) is a solution that acts intelligible and dynamically based upon data collected from real-time traffic.

IWI Infrastructure to Vehicle Information (IVI) communication is passing information from infrastructure to vehicles.

JSON JavaScript Object Notation (JSON) is a standard for defining data structures that can be serialized and deserialized. The standard provides human-readable text and can be directly used within a JavaScript environment.

JWT JSON Web Token (JWT) is an open standard for describing an access token and user claims in JSON format.

Kibana is a visualization plugin for Elasticsearch which provides all sort of visualization options for data stores in Elasticsearch. Real-time visualization are supported and automatically updates as new data is appended.

LTE Long Term Evolution (LTE) is a standard for wireless network communication on mobile devices.

Map Map is an ITS message that contains data about intersection topography, lanes, and geographical location. It us used in this project for Selecting the correct lanes to visualize in the GLOSA application..

middleware is a software layer situated between applications and operating systems. Middleware is typically used in distributed systems where it simplifies software development.

MVC Model View Controller (MVC) is a software architecture pattern.

MySQL is a free and open-source relational database management system that is queryable through SQL.

real-time data Data that is not kept or stored, but is passed along to the end user as quickly as it is gathered.

REST Representational State Transfer (REST) is a software architecture often used in stateless HTTP communication between a server and a client.

RPC Remote Procedure Call (RPC) is a communication protocol that allows remote clients, to invoke methods on a server.

Service Fabric is a distributed systems platform for easily packaging, deploying and managing scalable and reliable microservices.

SignalR ASP.NET core SignalR is a library for ASP.NET core developers to add real-time web functionality to their applications..

SPaT Signal Phase and Time (SPaT) is an ITS message that contains data about intersection state, such as if it is green, and when it will change. It us used in this project for calculating GLOSA speed recommendations.

TCP Transmission Control Protocol (TCP) is a low level network protocol.

TLC Traffic Light Controller (TLC) is a term used for describing the software and hardware that controls the traffic lights in an intersection. This term will be used interchangeably with intersection in this report..

TLEX Traffic Light Exchange (TLEX) is a platform for exchanging information between infrastructure and vehicles through a cloud server.

TTL Time To Live (TTL) is a timespan for an ITS message to be valid within. For example the SPaT messages have a TTL of three seconds, and should be considered invalid if older.

TTSC Time To Signal Change (TTSC) is the time until the signal changes from red to green or green to red.

V2I Vehicle To Infrastructure (V2I) communication is passing information from vehicles to infrastructure.

V2N Vehicle To Network (V2N) is a type of C-V2X and is passing information from vehicles to a cellular network.

V2X Vehicle To Anything (V2X) communication is passing information from vehicles to any receiver.

VM Virtual Machine (VM) is an isolated emulation of a computer.

WS WebSocket (WS) is a full duplex communication protocol, which runs on TCP sockets. Full duplex communication means that the traffic can flow both ways, e.g. the server can make calls to the client.

List of Figures

1.1	Visualization of Map message data	4
3.1	A mock-up of the C-Roads application Traffic Pilot. <i>source: [3]</i>	10
3.2	Screenshot of the Swarco Bike Timer prototype. <i>source: [4]</i>	10
3.3	GLOSA Integrated into the speedometer of a Car. <i>source: [5]</i>	10
3.4	A mock-up of GLOSA shown as a HUD in a vehicle. <i>source: [2]</i>	10
3.5	GLOSA embedded in the pavement with LEDs indicating the speed. <i>source: [6]</i>	10
5.1	Domain Model of the System	17
5.2	Sequence diagram of the process of Subscribing to updates	19
5.3	Sequence diagram showing the process of Sending data to Intersections	20
5.4	Sequence diagram showing the process of saving Statistics data	20

6.1	Component diagram of the system	22
6.2	First part of the Traffic Light Exchange (TLEX) Service class diagram	23
6.3	Second part of the TLEX Service class diagram	24
6.4	Traffic Streaming Service class diagram	25
6.5	TLC Registry Service class diagram	25
6.6	Geolocation Service class diagram	26
6.7	TLC Location Updater Service class diagram	27
6.8	Statistics Service class diagram	27
6.9	Authentication Service class diagram	28
6.10	Intersection selection sequence diagram	29
6.11	Demonstration of the shortcomings of this way of the cone approach	29
6.12	Subscribe to updates from TLC sequence diagram	30
6.13	ITS message received sequence diagram	31
6.14	Send CAM data sequence diagram	32
6.15	Save intersection pass data sequence diagram	33
6.16	Concept design for the Green-Light Optimal Speed Advisory (GLOSA) visualization	34
7.1	Component Diagram of the System	36
7.2	Using a reverse proxy to reach a service instance	40
7.3	Shows the Unity visualization inside an Android app while Google Maps runs picture-in-picture in the bottom right corner	45
7.4	Kibana visualization of egress & ingress approaches from intersection passes	47
7.5	Heatmap of intersection passes in Copenhagen, Denmark	48
7.6	Number of intersections passes and number of stops by intersection	48
7.7	The TLC overview dashboard from Kibana	49
7.8	The nodes and artifacts involved in a full system deployment	51
8.1	The three intersections that are hooked up to the TLEX Platform	53
8.2	Graph of system metrics from Distributed GeolocationService instances	54
8.3	Graph of system metrics from local GeolocationService instances	55
8.4	Graph of system metrics from local TrafficStreaming instances	55
8.5	Boxplot showing the distribution of the gathered latency data	56
8.6	Messages sent in TlexService	58
8.7	Messages sent in TrafficStreamingService	58
8.8	Three Node Configuration	59
8.9	Two Node Configuration	59
A.1	The up-to-date Gantt chart of the project milestones.	A
A.2	The project board on GitHub	D
A.3	Continuous integration using Jenkins	E
A.4	Better Code Hub analysis of the server code base	F
A.5	Service Fabric cluster explorer	G
A.6	Kibana front-end that is used to access Elasticsearch	G
E.1	Google Trends comparison of Service Fabric, Kubernetes, and Docker Swarm from the 8th of January 2017 to the 1st of June 2019	O
F.1	File structure of a Service Fabric Solution	R
G.1	High Fidelity Prototype	V

Attachments

This section will describe the contents of the ZIP file attached to this report.

The attached .Zip file contains the following folders:

- Source Code:** This folder contains the source code, as is.
- Demo-Movie:** This folder contains both the full length field test and a narrated demo movie, showing the system in use. *It is recommended to watch the latter before reading the report.*
- Report:** This folder contains a two-page printer friendly .pdf version of the report.
- Images:** This folder contains the images and figures shown throughout the report, named after their figure numbers.

1 : Introduction

This chapter will introduce and describe the basis of the project and the problems it aims to solve, as well as defining the goals of the project. Lastly, it will briefly introduce the rest of the report.

Contents

1.1 Introduction	1
1.2 Background Information	1
1.2.1 Traffic Light Controllers	1
1.2.2 The TLEX platform	2
1.2.3 Vehicle To Anything	2
1.2.4 ITS Message Protocols	3
1.2.5 Green-Light Optimal Speed Advisory	4
1.3 Project Description	5
1.4 Report Outline	5

1.1 Introduction

This project is issued by Swarco Technology A/S (Swarco), which is a company that develops and produces Intelligent Transportation Systems (ITSs) such as traffic lights and Traffic Light Controllers (TLCs) that is used to control one or more intersections [7]. Swarco Technology is part of the Swarco Group, which is one of the largest manufacturers of traffic solutions in the world [8].

The project concerns the development of a system that integrates with the TLEX platform. It should enable drivers to exchange data with an intersection in real-time. Thus, providing the foundation for developing intelligent smart traffic solutions based on real-time data.

1.2 Background Information

This section will present important background information, which is essential in order to understand the domain and the various products in the domain.

1.2.1 Traffic Light Controllers

Modern TLCs can dynamically adjust the phase and timing of the intersection, in order to optimize the traffic flow based on traffic conditions [9]. This is useful for decreasing time spent at red lights in the

intersection [10], or decreasing greenhouse gas emissions, by prioritizing large vehicles such as trucks and busses [10].

The TLCs can adapt to the traffic conditions based on sensor input. These sensors are often copper coils embedded in the road, which acts as an inductive loop. The sensors are activated, whenever a large piece of iron, e.g. a car or a truck, passes over the coils [11]. These coils are good at detecting the presence of larger vehicles, but not so good at detecting smaller vehicles such as motor cycles or cyclists, as they often do not contain enough iron to activate the sensors. These sensors are also not able to track the vehicle through the intersection, or accurately classify what type of vehicle is activating the sensor, making it difficult to grant priority to some vehicle classes.

Another vehicle detection technique is to use cameras. This technique is better at tracking vehicles and classifying vehicle types [12], but it may have problems in low visibility weather conditions such as heavy rain, snow, and fog [12].

IoT in the Traffic

Modern TLCs, produced by Swarco and their competitors, are connected to the internet, and exchanges data with cloud solutions, such as the condition of the equipment, a bad light bulb which needs to be changed etc. [13]. The TLCs can also send the phases of the signal groups in the intersection, which lanes have green and red light, and how long it will take before it will change phase [13].

An issue with these cloud-based solutions, is that they are often proprietary solutions, and incompatible with similar solutions from other manufacturers [14]. This problem has been addressed in the Netherlands, where ITS manufacturers and the government have started a partnership called Talking Traffic, which Swarco is a member of [15]. A product of this partnership, which addresses the problem, is a platform called TLEX [16]. This platform is a cloud platform, which can exchange real-time traffic data between TLCs and road users [17].

1.2.2 The TLEX platform

The TLEX solves the issue of proprietary solutions. The TLEX is a Vehicle to Network (V2N) platform developed and maintained by Monotch [17]. The purpose of the platform is to exchange real-time data about road side equipment and vehicles by using standardized ITS protocols [16, 14]. By using a standardized message set, instead of proprietary message formats, the TLEX makes it possible for road users to communicate with intersections, no matter who created the intersection, or who made their vehicle.

This platform is intended to be used by the car manufacturers, and other information brokers to provide valuable information about the individual intersections [17], as well as sending data to the TLEX, which it will forward to the TLCs.

The TLEX platform provides two different endpoints, the first, a REST endpoint for managing the session with the TLEX, the second, a TCP socket endpoint for exchanging data with the platform [18].

1.2.3 Vehicle To Anything

Vehicle to anything (V2X) is a technology which allows vehicles to transmit data to other nearby vehicles and infrastructure, using the Dedicated Short Range Communication (DSRC) protocol IEEE 802.11p (Wi-Fi p) [19]. Recently though, there has been a push for using 4G or 5G as a communication protocol. This approach is called Cellular-V2X (C-V2X), and the reasons behind this push, is firstly that it allows for economies of scale, as these protocols are already widely supported by telephone carriers; and secondly,

it also allows for V2N, where the system can make use of cloud services, such as the TLEX, to retrieve data about intersection timings, traffic conditions etc.

This data transmitted from vehicles follows the ITS message protocols, and could contain information about the presence and status of the car. This can be useful for detecting vehicles in intersections, as the data can describe the vehicle at a much greater detail than conventional detectors. It can also be used for communicating between vehicles, e.g. sending a warning to nearby cars that the road is slippery [20].

Infrastructure to anything (I2X) is a technology which allows infrastructure, such as TLCs and smart signs, to communicate with vehicles and other nearby infrastructure. This allows TLCs to broadcast messages about the intersection state and layout [21] to vehicles, which can be used to increase both road safety and driver awareness.

One of the great issues of V2X is currently that it still is an emerging technology, and only very few car manufacturers, is already producing cars with V2X support [20]. And as the average car in Europe is more than ten years old [22], it will be some time before this technology is available in all the cars on the road.

1.2.4 ITS Message Protocols

The ITS standard message protocols is a set of message types which is intended to be used for V2X communication. From all the message protocols, the most relevant message protocols for this project is the Signal Phase and Time (SPaT), Map, and Cooperative Awareness Message (CAM) protocols.

The protocol specifications states that these messages must be encoded using the Abstract Syntax Notation 1 (ASN.1) UPER¹ encoding [23, 24]. This encoding helps minimizing the amount of bandwidth used to transmit the data, as ASN.1 encodes the data at bit level [25].

SPaT is a message protocol for transmitting information about the state of intersections [21]. It is used to inform vehicles about when the signal will change e.g. to green. These messages should be sent from the TLCs each time they update their phase, which is ten times each second. The TLCs connected to the TLEX only sends the SPaT once every second.

Map (which is not short for anything) is a message protocol for transmitting geographic information about the intersections [21]. It specifies all the signal groups and lanes of the intersection. It can be used for drawing the intersection on a map, with some conversions. The Map messages are sent whenever the intersection geometry is changed, which they rarely do.

¹Unaligned Packed Encoding Rules

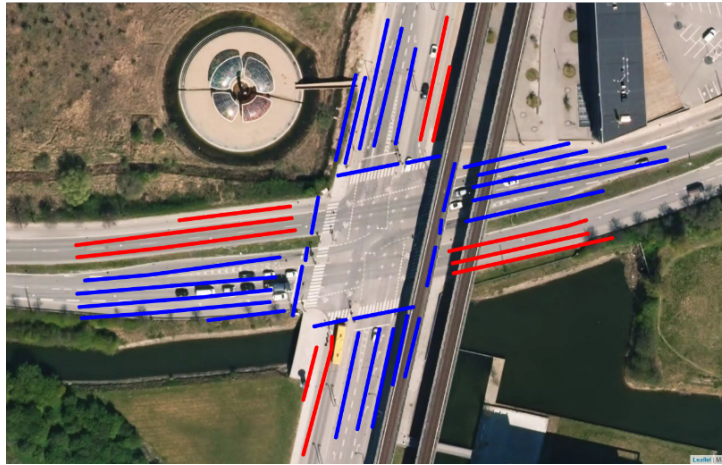


Figure 1.1: Visualization of Map message data

Figure 1.1 shows a visualization of a Map message, it shows all the lanes in the Vejlands Allé / Ørstedes Boulevard intersection on Amager in Copenhagen. The lanes account for both cars, cyclists and pedestrians. The blue lines are the ingress (entrance) lanes and the red lanes are egress (exit) lanes.

CAM is a message protocol for transmitting information about the presence of your vehicle [23]. The CAM protocol message consists of four different containers, two of which is mandatory, and two which are optional. The mandatory containers are the Basic container and the High-Frequency (HF) container [23].

The Basic container contains data about what type of vehicle or station the messages originates from, and the last geographic position of the vehicle or station.

The HF container contains information about the speed, heading and status of various fast changing equipment in the vehicle, e.g. the angle of the steering wheel or if the break or gas pedals is engaged.

The two optional containers are the Low-Frequency (LF) container and the Special Vehicle (SV) container [23].

The LF container specifies what role the vehicle has on the road, e.g. if it is a pedestrian, passenger car or a public transport vehicle. It also contains a status of the exterior lights, which shows if the vehicle has its turn indicators engaged etc. Finally, the LF container contains a path history for the vehicle. This container must be transmitted at least once every second.

The SV container specifies various extra information for special vehicles, such as busses ambulances, or trucks driving with dangerous goods.

The CAM specification specifies two tiers of CAM systems, tier 1, which sends the CAM multiple times each second. And a slower tier 2, which sends a full cam message once per second [23].

This message protocol can be used for safety critical purposes, such as blocking a left turn, if a CAM message is received, from another vehicle, would indicate that they will have a collision [20].

1.2.5 Green-Light Optimal Speed Advisory

One of the applications of V2X is a so called GLOSA system, which can suggest speeds to vehicles, so they can pass the intersection at green, based on the state of the intersection a vehicle is driving

towards. This speed suggestion can be achieved by several methodologies and ways of visualizing the speed advice².

Studies have shown, using traffic simulators, that GLOSA systems has the potential to save drivers upwards of 10% in fuel, by having to accelerate less often in intersections [2, 1]. This also have the added benefit of less air pollution from combustion engines. The studies also indicate that drivers can save upwards of 10% time spent idle in the intersections [2, 1].

1.3 Project Description

The project concerns the development of a system that should present users with data from road infrastructure via the TLEX platform. The system should also pass data from the users to the road infrastructure via the TLEX platform.

A product like this could help with the adoption of V2X , as it could bridge the gap of V2X support by allowing older vehicles to exchange data with other vehicles and infrastructure using the application.

It could also allow for better vehicle detection for the intersections. The data transmitted to intersections is more detailed and describes, among other things, vehicle type which could be used for vehicle prioritization, and various traffic flow optimizations.

It would also be a great tool for analyzing traffic flow patterns in intersections, as the application could be able to log which intersection ingress approach (direction of entry) and egress approach (direction of exit) was used. This could help optimizing the controller algorithms to adapt better to the traffic conditions of the intersection.

1.4 Report Outline

The report will continue with the problem chapter. This chapter will identify the essential problems, that the project will address, and determine the objectives for the project.

The following chapter, Related Work, will analyze and showcase similar solutions already made to this problem.

The Requirements chapter will describe the requirements defined for the system, as well as describing selected user stories, which will be used throughout the report. The report will be technology agnostic until the Implementation chapter.

The Analysis chapter will formalize the requirements into a simple system description. This section will also flesh out the selected user stories.

The Design chapter will describe various design choices made to the system, as well as determining the system architecture.

The Implementation chapter will describe the most relevant technology choices and how the user stories was realised.

The Experimental Validation chapter will aim to validate the non-functional requirements which was specified for the system.

The Discussion chapter will discuss whether the product lived up to the requirements, as well as if it solved the specified problems.

The Conclusion chapter will summarise the report and conclude the project.

²some of these will be shown in chapter 3

The Future Works chapter will lay out future directions for the project.

The end of the report contains an Appendix, which contains a Process report, that describes how the project was conducted, as well as establishing a timeline for the project.

Summary: *This chapter has introduced the relevant background information, such as current ITS technology and TLCs, V2X, the TLEX platform, the ITS message protocols, and GLOSA. It has described the project at a high level, and provided a layout of the following chapters of this report.*

2 : Problem

This chapter will introduce the main problems, which the project aims to address. It will also describe the objectives of the project, as well as describing some high-level features of the product, and determine the scope of the project.

2.1 Problem Description

Many modern intersections are controlled by adaptive traffic controllers, which tries to optimize the traffic flow, based on traffic conditions by using hardware detectors. As mentioned in Introduction, these detectors are good, but not flawless, as they are not able to precisely determine vehicle type and speed. This can negatively affect the flow of traffic and it removes the possibility of optimizing the traffic algorithm to prioritize various vehicle types. Some detectors lose precision in poor weather conditions, which can make the TLCs blind to vehicles all together.

In the ever-expanding world of interconnected devices and systems, a way of connecting vehicles and intersections seems essential. In present day a great number of drivers travel the roads and pass through intersections, having to continuously assess the status of the traffic lights. Enabling V2X communication has the potential of improving the mobility ecosystem such as adaptive traffic control and driver awareness.

PS 1: Is it possible to make a live data exchange between vehicles and roadside infrastructure?

(a) Is there too much delay in the data exchange, for the data to be considered usable?

PS 2: What architectural choices can be made to promote scalability within a server application?

PS 3: Is it possible to give drivers a speed recommendation based on the infrastructure?

(a) What is the most viable way of presenting speed recommendations?

These are the problem statements, which this report strives to answer.

2.2 Project Objectives

We intend to make a server solution which should enable live data exchange between users and road infrastructure via the TLEX platform. Furthermore, we intend to investigate which software architecture best fulfill the requirements of scalability, resilience and responsiveness.

We intend to make a mobile application as a proof of concept. This application should utilize the live data exchange functionality of the server solution. The application should provide GLOSA [26] functionality to the user, which should help vehicles to pass through the intersection at an optimal speed.

Lastly, we intend to make a web application to act as an administrative client on top of the server solution. The web application should display statistics about traffic infrastructure and users.

2.3 Problem Scope

We do not intend to neither configure or implement new features inside the TLCs themselves. Even though we wish to exchange data with TLCs, the TLCs themselves will be out of scope for this project. At the end, this means that we will not change any TLC traffic algorithms.

Summary: *This chapter have described the problems, which this project aims to address. These problems are the problem of vehicle detection for intersections, and the drivers having to constantly orientate them selves about the state of the intersection.*

The chapter have also defined some questions, which the report aims to answer. The first question is regarding the possibility of real-time data exchange with intersections. The second is regarding the architecture of the system. The last is regarding the possibility of speed recommendations.

The chapter declared some objectives for the project. These objectives are to create a server solution, which interfaces with the TLEX platform. And to create a proof of concept application, which, based on data from the server, should provide drivers with speed recommendations to pass the intersection.

Finally, the chapter declared the scope of the problem, to not include changes to the TLCs.

3 : Related Work

This chapter will present state of the art related work, in order to assess how others in the domain are approaching this problem.

3.1 NordicWay2

NordicWay2 is a V2N test project, that aims to provide non-safety critical information to vehicles, such as road conditions and intersection timings [27]. It is the product of Nordic collaboration between public and private sectors in Denmark, Sweden, Norway and Finland [28]. NordicWay2 has a broad scope, and focuses on multiple technologies for enhancing traffic safety, fluency, and decrease vehicular pollution [29]. The project succeeds the first NordicWay project, which focused on sharing information about road and traffic conditions across the Nordic borders [30].

3.2 GLOSA Visualizations

For a GLOSA system to be effective, it is important that its speed recommendation is effectively communicated to the driver. The following sections will describe different ways of visually communicating the GLOSA suggestions.

3.2.1 C-Roads

The C-Roads Platform is a joint initiative of European Member States [31] who has proposed a GLOSA application [26]. This application has been implemented by GEVAS software GmbH under the name Traffic Pilot [3]. Their application, shown in Figure 3.1, provides a so called ‘*forecast*’, which is a visualization of the signal phases and the location of the user.

The forecast will visualize the green and red times of any signal group that an incoming intersection approach consists of. The application will also display a countdown from red to green when holding still at an intersection.

3.2.2 Bike Timer

Swarco Technology has developed a prototype application referred to as ‘Bike Timer’, which is shown in Figure 3.2 [4]. The application will visualize a speed recommendation by displaying a speedometer with a green curve surrounding the speedometer. The key concept being that the vehicle should keep a speed that is within the green curve to ensure they will pass an intersection when the signal is green.



Figure 3.1: A mock-up of the C-Roads application Traffic Pilot. *source: [3]*



Figure 3.2: Screenshot of the Swarco Bike Timer prototype. *source: [4]*



Figure 3.3: GLOSA Integrated into the speedometer of a Car. *source: [5]*



Figure 3.4: A mock-up of GLOSA shown as a HUD in a vehicle. *source: [2]*



Figure 3.5: GLOSA embedded in the pavement with LEDs indicating the speed. *source: [6]*

3.2.3 Vehicle Dashboard Integrated

While it is certainly possible to build a GLOSA application for a smart phone, and position the smart phone cleverly inside your car. It is also possible to integrate a GLOSA system directly in the car. Volvo has been cooperating with Swarco in a project, where they show the advised speed as an interval on the speedometer as shown in Figure 3.3 [5].

3.2.4 Heads-up display

Figure 3.4 shows another approach to displaying GLOSA information is to use a heads-up-display (HUD) to project the information into the windshield of the car, making it appear to the driver as an overlay on the road [2].

3.2.5 Roadside Equipment

Figure 3.5 shows a proprietary solution, developed by Swarco, where LED lights is embedded along the cycle path towards an intersection. These LED lights indicate the green wave of the upcoming intersection. In order to pass the intersection for green, the cyclists must ride along the green-lit LEDs [6].

Summary: This section described the NordicWay2 C-ITS test projects. And multiple ways of displaying GLOSA speed recommendations to drivers was introduced.

4 : Requirements

This chapter describes the actors of the system. It will describe some of the key user stories elicited from the functional requirements. The chapter will also describe the non-functional requirements, and various system constraints and concerns influenced by the domain. Lastly, it describes the boundaries of the system, where it interfaces with other systems.

Contents

4.1	Actors	11
4.2	Central User Stories	12
4.2.1	User Story 1: Subscribe to Updates	12
4.2.2	User Story 2: Send Data to Intersection	12
4.2.3	User Story 3: Statistics	12
4.3	Nonfunctional Requirements	12
4.4	Software Quality Attributes	14
4.5	Boundaries	16

4.1 Actors

The first actor is the “*Administrative User*” (admin). This user is a Swarco employee, and is working on the development and maintenance of the system. This admin will have access to system logs and configurations, the admin also has access to runtime metrics.

The second actor is the “*Driver*”. This user uses any kind of automotive vehicle for transportation. The driver uses the system to gain speed recommendations, when driving towards an intersection, in order to conserve fuel. The driver also provides the TLC in the intersection with data about his presence in the intersection, which the TLC potentially can use for regulating the green-light timings.

The third actor is the “*Cyclist*”. This user uses any kind of bicycles or a moped for transportation. The cyclist has access to the same features that the driver has, except he is only shown recommendations for roads which he can drive on, which is bicycle paths and car lanes on which he must drive.

The fourth user is the “*Customer*”. This user is a customer of Swarco, often municipalities, or traffic regulators. This user has access to various statistics about intersection performance and utilization.

The fifth user is the “*Special Vehicle Driver*”, this user drives special vehicles, e.g. a bus, an ambulance, or a truck loaded with dangerous goods. This driver has access to the same features as the driver, but can specify more data, which are transmitted to the TLCs about their vehicle. This may allow the TLC to give them better prioritization.

Drivers, cyclists, and special vehicle driver will also be referred to as the “*app user*”.

4.2 Central User Stories

This section presents three central user stories selected from the complete list of user stories. These user stories cover the most central parts of the system. More details about the development process and the collaboration with Swarco can be read throughout the process report in appendix A.

All the user stories are elicited continuously throughout the project based on collaboration with Swarco. The complete list of user stories can be seen in the appendix B.

4.2.1 User Story 1: Subscribe to Updates

"As a driver, I want to subscribe to updates from relevant intersections so that I can get speed recommendations and countdowns to signal change on the intersection I'm passing through"

When an app user drives towards an intersection, it may be relevant to know when the user will get a green light, or for how long the current light will last. This information should be conveyed to the user through the system and displayed visually for the user. The visual representation of the information should be easily understood at a glance – that is; not a lot of text and numbers that needs to be read while driving. If the user is met with a red light and must stop then a visually represented countdown should be presented to the user.

4.2.2 User Story 2: Send Data to Intersection

"As a driver, I want to send data to the intersection I am driving towards in order to ensure that the intersection knows about my presence and can adapt its state accordingly"

Most intersections in Denmark use detectors to adjust the signal based on current traffic conditions [32]. However, as traffic detectors are physically placed in the world, they are subject to wear and tear and as Steen Merlach Lauritzen, from the highway agency (Vejdirektoratet) in Denmark, says every fifth intersection is expected to have malfunctioning traffic detectors [32]. This leads to an increased number of vehicle stops and accelerations as well as potential traffic incidents.

By giving intersections new digital ways of detecting traffic, without physical detectors, the intersections will be able to better adjust their state based on current traffic conditions. This leads to less vehicle stops and less accelerations, which is a financial benefit to all parties, because of fuel consumption and less frustration for drivers.

4.2.3 User Story 3: Statistics

"As a customer, I want to view statistics about traffic flow so that I will know if an intersection must be optimized for better traffic flow"

The traffic at intersections will never be a constant, but will instead change because of many factors like road work, holidays, new buildings etc. Therefore, being able to monitor the traffic at every intersection becomes relevant for optimizing the intersections. It should be possible for customers to see traffic information regarding traffic volumes, stops, and stop duration.

4.3 Nonfunctional Requirements

The nonfunctional requirements of a system specify various requirements, to which the system must comply. This section is divided into three groups of nonfunctional requirements. Firstly, product re-

quirements, which are requirements to the system. Secondly, Organizational requirements, which are requirements to the development process. Finally the External requirements, which are requirements imposed by external sources, such as legislation.

These requirements are elicited in collaboration with Swarco and in accordance with the TLEX documentation specified by Monotch [18].

Product Requirements

ID	Description
P01	The system should discard older ITS messages, according to the TLEX specification [18]
P02	The system should exchange ITS messages at nearly the same rate as the expected throughput of the TLEX service [18]
P03	The system should be designed with an architecture that allows for scalability and resilience.
P04	The system should be able to communicate using full duplex communication
P05	The system should make use of SSL certificates for encryption of data
P06	The system should make use of external authentication providers (OAUTH)
P07	The system should include an Android client and a browser client
P08	The system should store data in a data store, which makes it queryable.
P09	The application should convey information at a glance
P10	The system must be able to automatically recover from faults within a reasonable amount of time

Table 4.1: Product Requirements

The requirements above state that the system must not introduce bottlenecks to the real time data, the specific requirements can be seen in appendix C. They state that the system must be implemented using an architecture, which promote scalability and resilience. They also present some security requirements of data encryption and user authentication. It is specified that the system must provide two user interfaces to the system. The system should store data in a way that allows it to be queried. Finally, the application should be able to show information to the driver at a glance.

Organizational Requirements

ID	Description
O01	The development process should be executed using an agile methodology.
O02	The produced codebase should be written in C# (C-sharp) and Java

Table 4.2: Organizational Requirements

The requirements above specify that the system must be implemented using an agile approach. They also state that the codebase must be written in C# and Java.

External Requirements

ID	Description
E01	The system must comply to the EU GDPR legislation [33].
E02	It is illegal to interact with handheld phone equipment while driving in Denmark [34].
E03	It is illegal to drive faster than the speed limits in Denmark [34].

Table 4.3: External Requirements

The external requirements state that the system must comply with the EU-GDPR legislation. It is also stated that it is illegal in Denmark to drive while interacting with a mobile phone, meaning that the interaction with the application must be kept at a minimal. Lastly, it states that it is illegal to drive faster than the speed limit, which means that, ethically at least, the application should not show speed recommendations above the speed limit.

4.4 Software Quality Attributes

This section will describe the quality attributes, which must be provided by the system. These quality attributes are nonfunctional requirements used to evaluate the performance of the system.

Fault Tolerance

In a real-life scenario with a live high demand system running 24/7, users are likely to see downtime if fault tolerance is not considered for the system. At some point a system may run into a fatal error and crash, the hardware that hosts a server shuts down, for whatever reason, or maybe the network connection to a server gets cut off. For either reason all the user will see is that the system is faulty and unresponsive.

In order to ensure that the system is more fault tolerant, the system must be hosted on multiple hosting environments. There are different aspects to this, but the main concern is that if one network switch or one server cluster is down, then the entire system should not go down with it.

Scalability

As the demands for a system increases the ability of a system to increase its resources becomes important. We have all experienced the frustration of wanting to watch the latest episode of your favorite TV-show but being unable to because the demands for a system becomes too much for the system to handle.

Being able to scale the system will save the users from experiencing this frustration. The system must support scaling both horizontally, adding more machines, and vertically, adding more computing power, to handle increased loads [35]. Most importantly the system must support horizontal scaling for the simple reason that vertical scaling is limited by hardware [35].

Availability

For a system to have high availability a system must have a high fault tolerance, but other service-affecting operations will also affect the availability of a system.

To achieve as high availability as possible the system must be able to receive software and configuration upgrades without causing any downtime.

Reliability

As described in the requirements section 4.2 the system must enable users to subscribe to updates from intersections. This means that the system must keep a state describing which user is subscribing to which intersection. To ensure as high reliability as possible this state must be replicated into backup service instances, hosted on separate hosting environments other than the primary service. These backup services must be activated in case of any downtime to the primary service instance.

Performance

The latency and throughput of the service are very important, as it deals with real-time data, meaning data which is time sensitive. Telling an app user that the traffic light will be green for another three seconds will not help the app user, if the message is delayed by e.g. five seconds. Therefore it is important to keep the added delay at a minimum, meaning that the time it takes from the TLC sending the message until the app user receives it must be minimal.

Safety

Traffic Safety: The safety of users is paramount, meaning the safety of the users when they drive using the application. It is important that the application is “Traffic safe”, meaning that it does not cause any unnecessary distractions to the driver, and that the driver can gain the information needed at a glance.

Data Safety: It is also important that any personal data that pass through the system is dealt with safely and transparently, as these should comply with the European Union General Data Protection Regulations (EU-GDPR).

Distribution

The system architecture needs to be distributed, as there must be a server solution and a client system (the GLOSA application), in order to support multiple app users at once with the same data connection to the TLEX platform. The server solution in itself must also be distributed in order to fulfill the other quality attributes; allows for scaling horizontally, being fault tolerant, being highly available etc.

Interoperability

The system must be able to communicate with the TLEX cloud service using standardized ITS message protocols.

Persistency

The system must store data about the drivers passing through intersections. This is necessary, as it can be used for statistical analysis of intersection performance, or potentially for predicting traffic patterns.

Maintainability

The system should be easily maintainable, which means that the codebase should be written using common coding practices, as this could make it easier for new developers to understand the code.

The codebase should also strive to be as loosely coupled as possible, as this will make changes to the codebase easier, without having to deal with ripple effects.

Extensibility

The system should strive to be extendable, in the sense that it should be possible to deploy new services which can provide new features and endpoints.

Testability

The system should strive to be testable. This will make it easier to write automated tests, which can be used for regression testing; this ensures that breaking changes to the codebase are discovered and handled.

This will also make it possible to setup a continuous deployment pipeline, which will decrease the time it takes for new features to make it to the production environment.

4.5 Boundaries

This section will describe the system boundaries, which are the borders of the system, where it interacts with the actors and other systems.

The system should interact with the TLEX platform to gain data from the intersections. The data from the TLEX uses the ETSI ITS message protocol standards SPaT and Map. This data is encoded using the UPER ASN.1 encoding protocol. This means that the system must be able to decode these messages in order to do anything useful with the data.

The app users should interact with the system through the mobile application. The application should periodically fetch information about nearby intersections, such as their location and ID. Whenever the application user heads towards a TLEX enabled intersection, the application must establish a connection with the server solution to exchange real-time data about the intersection status.

Summary: *This section has introduced the actors of the system. It has introduced a selection of the functional requirements in the form of user stories. The nonfunctional requirements and quality attributes of the system. Finally, it has introduced the most relevant system boundaries.*

5 : Analysis

This chapter will present a domain model, which maps real world objects, and gives an overview of the relationships between them. The chapter also elaborates on the central three user stories by visualizing them with sequence diagrams, showing their overall goals.

Contents

5.1 Domain Model	17
5.2 Sequence Diagrams	19
5.2.1 User Story 1: Subscribe to Updates	19
5.2.2 User Story 2: Send Data to Intersection	19
5.2.3 User Story 3: Statistics	19

5.1 Domain Model

A domain model describes the real-world concepts which are discovered when investigating the domain and when talking to clients. This also means that the domain model should be understood both by software developers as well as domain experts. Therefore, it's important to use terms from the domain. The following diagram visualizes the domain as discovered for the project.

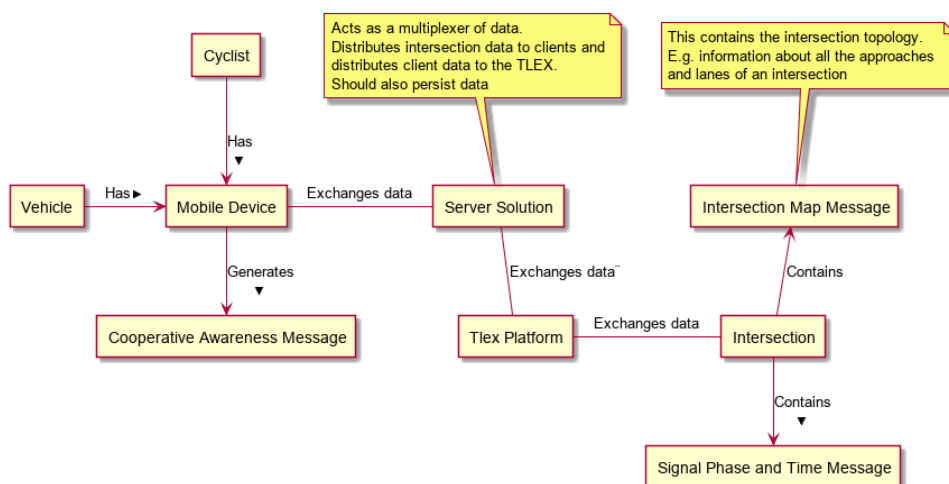


Figure 5.1: Domain Model of the System

In figure 5.1 the most relevant parts of the domain are highlighted. The server solution is particularly noteworthy in this case, as it is responsible for exchanging data between mobile devices and the TLEX. The TLEX platform will be responsible for gluing any other services to the intersections which interfaces with the platform. This openness to other services is what this project will be based on. Without this platform to connect to intersections the interaction with intersections would not be possible.

Intersections

The system must be able to communicate relevant intersection data to mobile devices. The system must also be able to receive and forward data to the intersection sent by mobile devices. In order to accomplish this a standardized data format is required. The data format requirements are set by the TLEX platform, as they must interpret and multiplex the data. According to their documentation [18] the message payloads must be UPER ASN.1 encoded [36]. Three message payloads were discovered to be particularly relevant to the project; the SPaT, Map, and CAM. Further information about the ITS Message protocols can be seen in subsection 1.2.4

Signal Phase and Time Message

As an intersection changes its state and time passes by SPaT messages are generated to represent the intersection's current state. This state contains all the information regarding all its signals, that is, which signal are currently red, and which are green. The state also contains information about when every single signal is set to change its state, red to green etc. This information is very relevant for giving speed recommendations as described in the user stories in chapter 4.

Map Message

The Map message contains information about an intersection's topology. The geographical location of the intersection as well as the geographical location of lanes are contained in this message. The lanes of the intersection are grouped into approaches and split into vehicle lanes and bicycle lanes. This message is useful for understanding the intersection and necessary for knowing which SPaT message might be relevant to the individual app user.

Mobile Devices

It is expected that mobile devices will communicate with the server solution. The way the app user interacts with mobile devices changes whether you are in a vehicle or on a bicycle. For example when you are riding on a bicycle the system will display bicycle lanes. Therefore, the differentiation is important for this domain. Certain safety critical requirements arise, both legally and ethically, when building an application for end-users driving through traffic¹.

Cooperative Awareness Message

Many intersections are traffic dependent and will dynamically change their state depending on current traffic conditions. This is done using physical detectors which are subject to wear and tear, which often results in faulty detectors [32]. This can be frustrating for driver stopped at a red light, as the only vehicle in the intersection. To mitigate this problem the mobile device must be able to send CAM to the server solution. This information will then be distributed to relevant intersections which ultimately will be able to use the CAM as a means for vehicle detection.

¹see the external requirements in chapter 4

5.2 Sequence Diagrams

This section will present high level sequence diagrams covering the central user stories. The diagrams are not final and will be elaborated in the following *Design* chapter.

5.2.1 User Story 1: Subscribe to Updates

This user story can be divided into two steps, fetching data about nearby intersections, and subscribing to updates from an intersection.

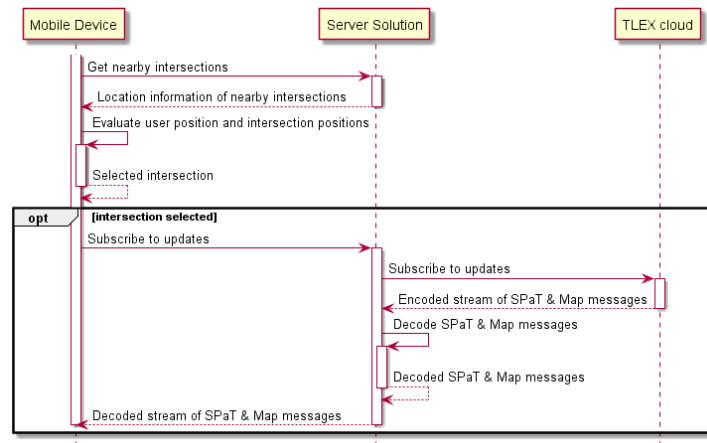


Figure 5.2: Sequence diagram of the process of Subscribing to updates

Figure 5.2 shows the two steps of subscribing to updates. It starts by sending a request to the server solution for getting nearby locations. The mobile device then evaluates if it is heading towards an intersection, using its current location and heading. If a valid intersection is found, the mobile device will send a request to the server solution, asking to subscribe to real-time data from the given intersection. The server solution will get this data, encoded using ASN.1, from the TLEX platform. The server solution will decode the ASN.1 encoded data, and pass it back to the mobile device as it is received from the TLEX.

5.2.2 User Story 2: Send Data to Intersection

This user story is driven by location updates in the application.

Figure 5.3 shows the sequence started by a location update received. This triggers the mobile device to generate a new CAM message, which is sent to the server solution where it is encoded to an ASN.1 format. The encoded message is then dispatched to the TLEX platform, where it should be dispatched further to the TLC.

5.2.3 User Story 3: Statistics

This user story can be divided into two steps, a data logging step and a dispatch step.

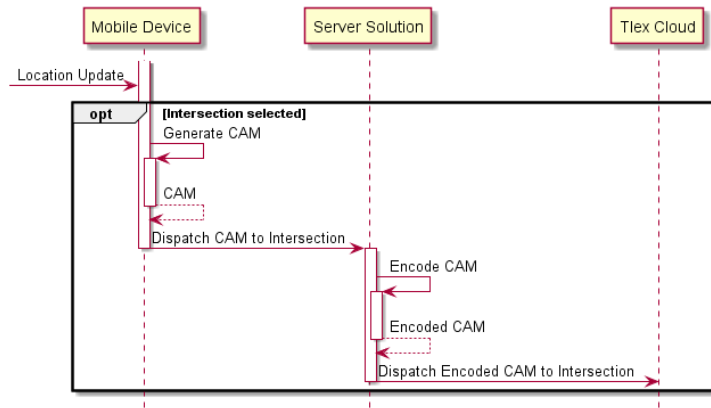


Figure 5.3: Sequence diagram showing the process of Sending data to Intersections

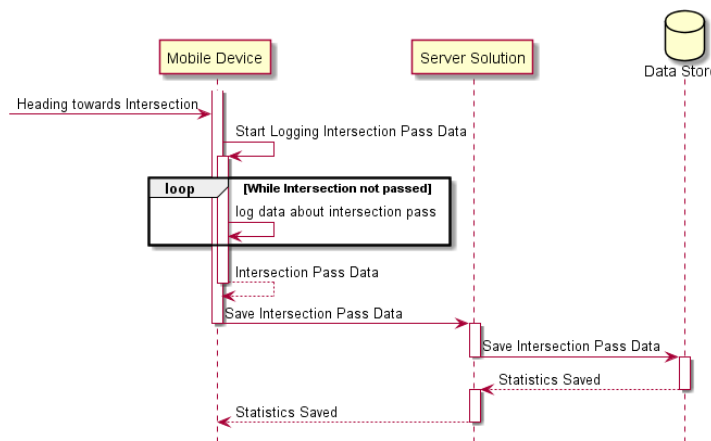


Figure 5.4: Sequence diagram showing the process of saving Statistics data

Figure 5.4 shows the sequence triggered by an event, which indicates that the application user is heading towards an intersection. The mobile device will start logging data, such as speed and stops, until the intersection has been passed. The mobile device will then send a request to the server solution, asking it to save the intersection pass data. The server solution will then save it to its data store.

Summary: This chapter has described various elements of the domain, the SPaT, Map, and CAM, as well as the server solution, TLEX, and Mobile device. This chapter has also described the central user stories at a high level and defined their key interactions with the server solution.

6 : Design

This chapter will determine the choice of software architecture of the server solution. The composition of the system will be presented at a high level in the form of a component diagram. The chapter will then take a deeper dive into various parts of the system and describe the role and inner composition of these. The chapter will describe the transformations that the design choices has on the three user stories. Finally, the chapter will describe the GLOSA visualization.

Contents

6.1 System Architecture	21
6.1.1 Service Overview	21
6.1.2 Service Description	22
6.2 Sequence Diagrams	28
6.2.1 User Story 1: Subscribe to Updates	29
6.2.2 User Story 2: Send Data to Intersection	31
6.2.3 User Story 3: Statistics	32
6.3 GLOSA Application	33
6.3.1 Selecting the Correct Lanes	33
6.3.2 Visualization	34

6.1 System Architecture

The architecture of the system is the high-level structure of the system. Appendix D describes three different approaches to software architecture, which are considered for the server solution. The architectures are the client server, component based, and the microservice architecture. Based on the properties of these architectures, the microservice architecture is chosen for the server solution. This architecture is chosen as it suits the requirements regarding scalability, fault tolerance, and resilience the best [37, 38]. This means that the system must be decomposed into small units with a limited area of responsibility.

6.1.1 Service Overview

This section will introduce and describe the various microservices, which makes up the server solution.

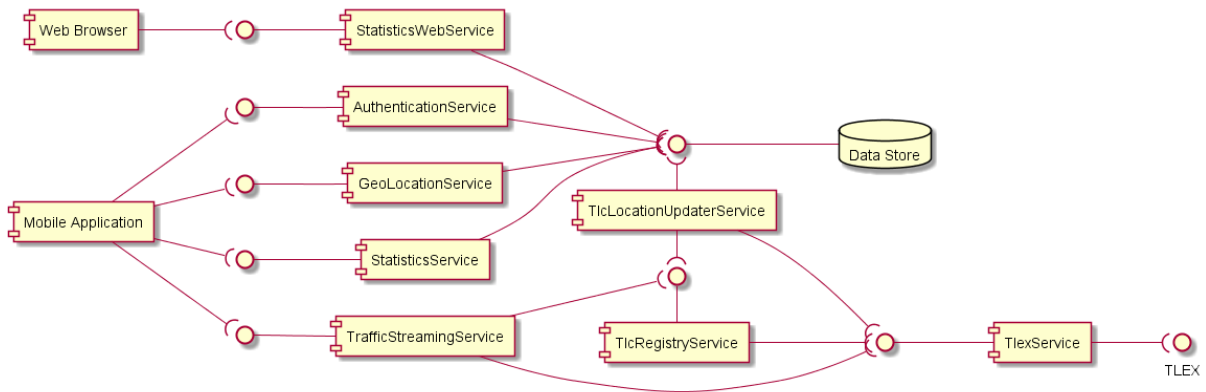


Figure 6.1: Component diagram of the system

Figure 6.1 shows an overview of the system decomposed into components. It shows the mobile application on the left side, interfacing with the four services **AuthenticationService**, **StatisticsService**, **GeoLocationService**, and **TrafficStreamingService**. The first three of those interfaces with the data store, from which they get the data they provide the mobile application. The **TrafficStreamingService** and **TlcLocationUpdaterService** interface with the **TlcRegistryService**, where they get the correct endpoint to interface with the **TlexService**. The **TlcRegistryService** also interfaces with the **TlexService** to register the endpoints. Finally, the **TlexService** interfaces with the external TLEX platform.

The figure also shows the **StatisticsWebService**, which provides a web interface for data visualization, which can be accessed through a web browser.

6.1.2 Service Description

This section will dive deeper into each service, and explain their purpose and role in the system, as well as the inner composition of the services.

TlexService

This service is the interface to the TLEX platform. The purpose of this service is to send messages to and receive messages from the TLEX platform. It has the responsibility of creating and managing the session with the TLEX platform. This service should provide an interface, which will allow other services to subscribe to updates from and send data to the TLEX platform.

This service is probably the most complex of them all, as it has such a wide area of responsibility. The reason it has a wide area of responsibility is that it all relates to the TLEX system, and in the event that Swarco decide to change the underlying platform to something other than the TLEX, it would essentially be possible to reuse all the other services in the system.

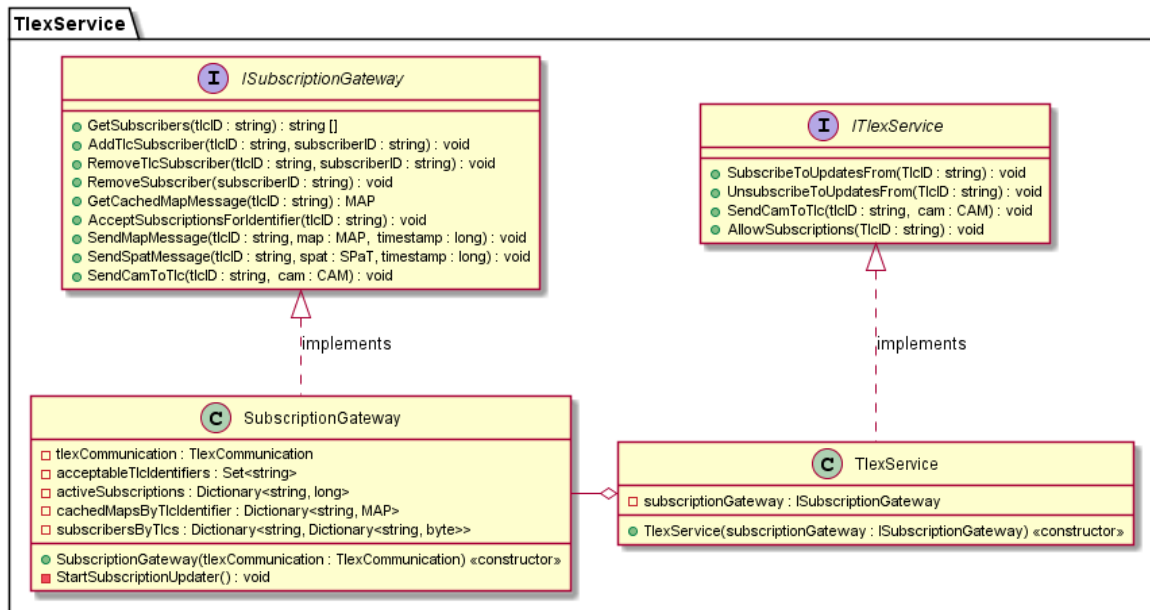


Figure 6.2: First part of the TLEX Service class diagram

Figure 6.2 shows a class diagram covering the first half of the `TlexService`. The `TlexService` in the lower right corner implements the interface `ITlexService`. An instance of the `ISubscriptionGateway` is injected into the `TlexService` class through the constructor. The `ISubscriptionGateway` is implemented by `SubscriptionGateway`, and its purpose is to handle the subscriptions, and make sure that the TLEX session is updated as new subscriptions come in.

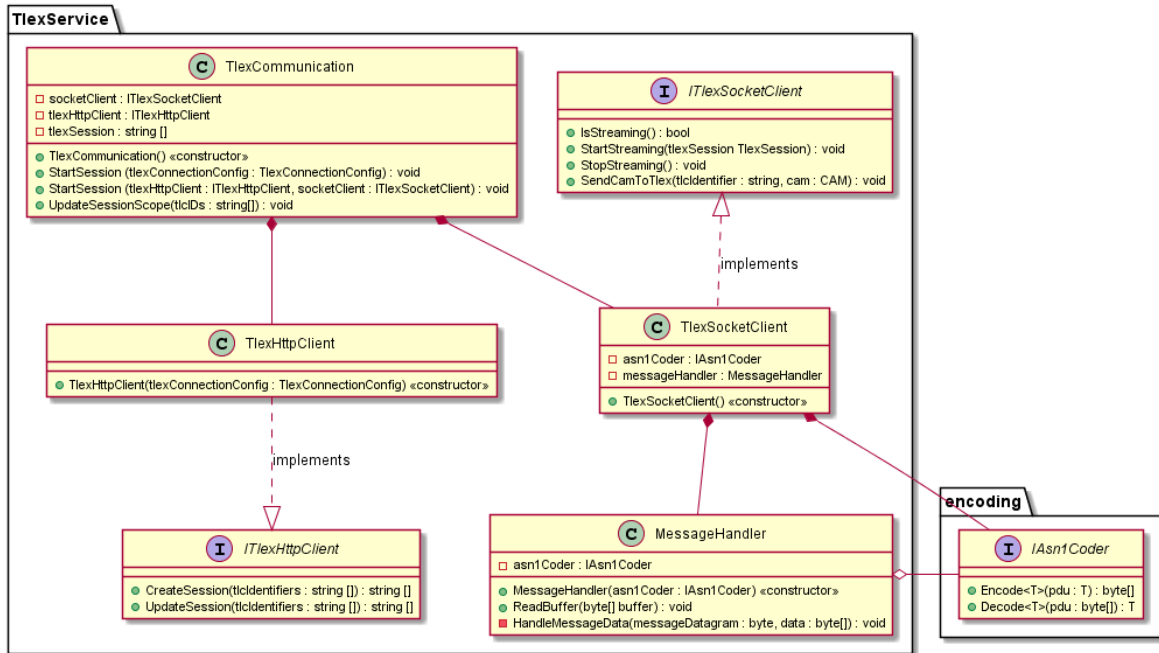


Figure 6.3: Second part of the TLEX Service class diagram

Figure 6.3 shows a class diagram of the second half of the `TlexService`. The `TlexCommunication` class is used by the `SubscriptionGateway` to handle the TLEX session. It utilizes the `TlexHttpClient` for starting and updating the session using an HTTP API provided by the TLEX platform. The `TlexCommunication` passes the session details on to the `TlexSocketClient`.

The `TlexSocketClient` is then responsible for opening a TCP socket stream to the TLEX platform. This will start an exchange of ITS messages. The messages are decoded using the `MessageHandler` class, which defines utility methods for reading and decoding ITS messages. Finally, the `TlexSocketClient` passes the decoded messages on to the `SubscriptionGateway`, where it is distributed to the subscribers.

TrafficStreamingService

This service is responsible for exchanging real-time data with the mobile application. The reason for having the mobile application connect to this service, and not directly to the TLEX service, is that it makes it possible to multiplex the data received from the TLEX platform, and not make redundant sessions with the TLEX platform.

Redundancy is not necessarily a bad thing in the perspective of microservices, but this constraint has been introduced as the TLEX platform does not allow intersections to be part of multiple sessions with the same TLEX user.

It also makes it possible to scale the service based on user demands, without having a major impact on the rest of the system.

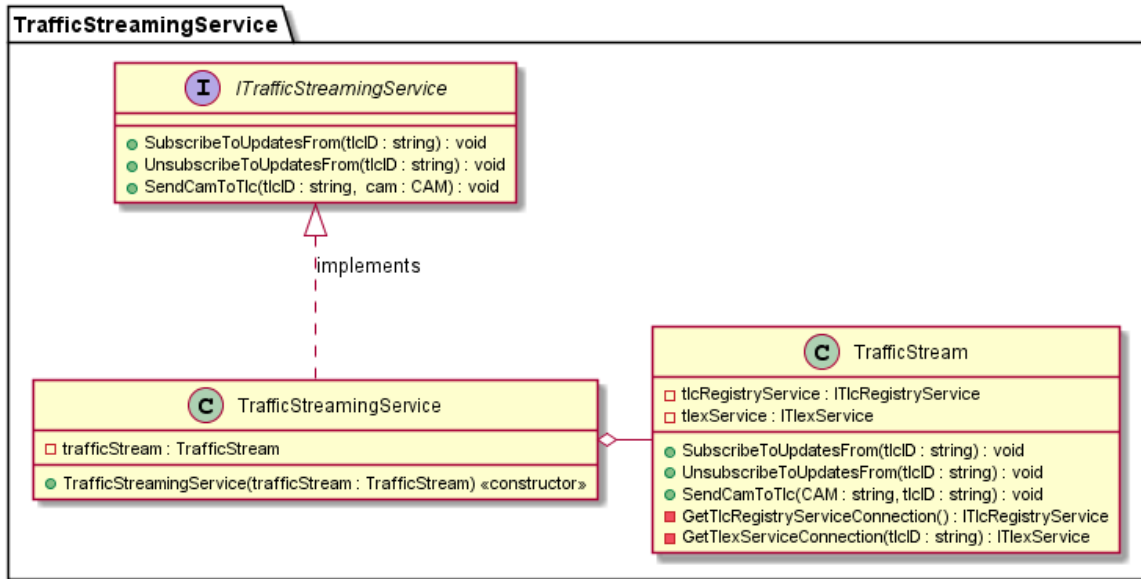


Figure 6.4: Traffic Streaming Service class diagram

Figure 6.4 shows a class diagram of the TrafficStreamingService. Starting with the TrafficStreamingService, in the lower left corner, it implements the ITrafficStreamingService interface, which defines functions for subscribing and unsubscribing to updates from a TLC, as well as sending CAM messages to a TLC. The TrafficStreamingService gets the TrafficStream class injected, through its constructor, and uses it for connecting to the TlcRegistryService, where it finds the correct TlexService connection for a given TLC.

TlcRegistryService

This service is responsible for connecting other services with the correct TLEX Service instance, based on the TLC the other service needs data from. This service is also responsible for telling the TLEX service which intersections it can receive subscribe requests from, as a safeguard against redundant sessions.

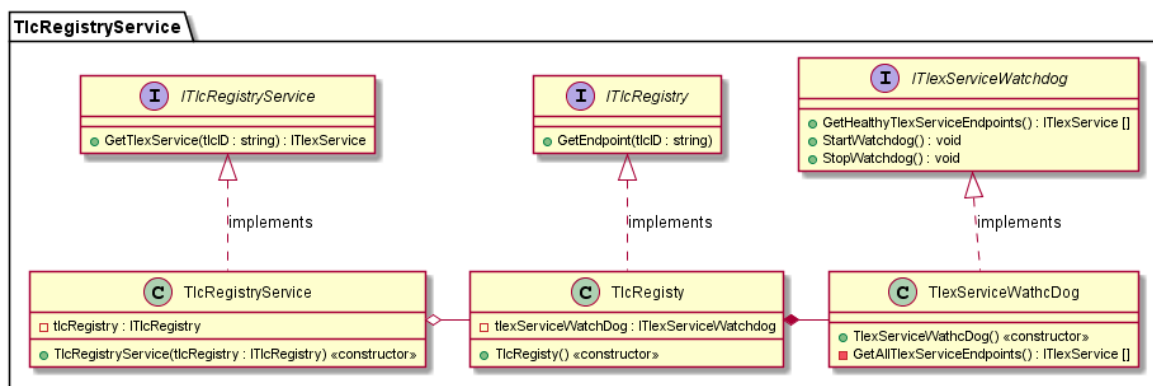


Figure 6.5: TLC Registry Service class diagram

Figure 6.5 shows a class diagram of the `TlcRegistryService`. Starting from the `TlcRegistryService`, in the lower left corner, it provides functionality to get a the correct `TlexService` endpoint for a given TLC. It does this using the `TlcRegistry` class, which keeps track of all `TlexService` instances using the `TlexServiceWatchdog`. The “Watchdog” periodically scans for changes in the available `TlexService` instances and keeps the `TlcRegistry` updated.

GeolocationService

This service is responsible for providing the mobile application with information about nearby intersections, which it fetches from a central data store. This information includes the location of the intersection as well as an ID which is needed for subscribing to data from the intersection.

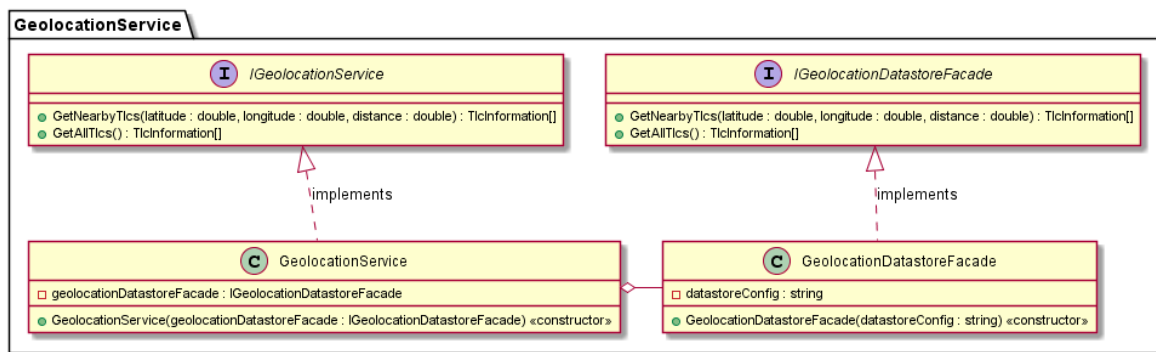


Figure 6.6: Geolocation Service class diagram

Figure 6.6 shows a class diagram of the `GeolocationService`. Starting with the `GeolocationService`, in the lower left corner, this class is the interface for the mobile application to get the information about the intersections. The information comes from a data store, which is accessed using the `GeolocationDatastoreFacade`, which defines functionality for fetching data from the data store.

TlcLocationUpdaterService

This service is responsible for updating the data about intersections for the Geolocation to provide. The data is updated over time and is supposed to catch updates such as intersections being added to or removed from the TLEX platform, or changes made to the intersection such as an update to the Map data.

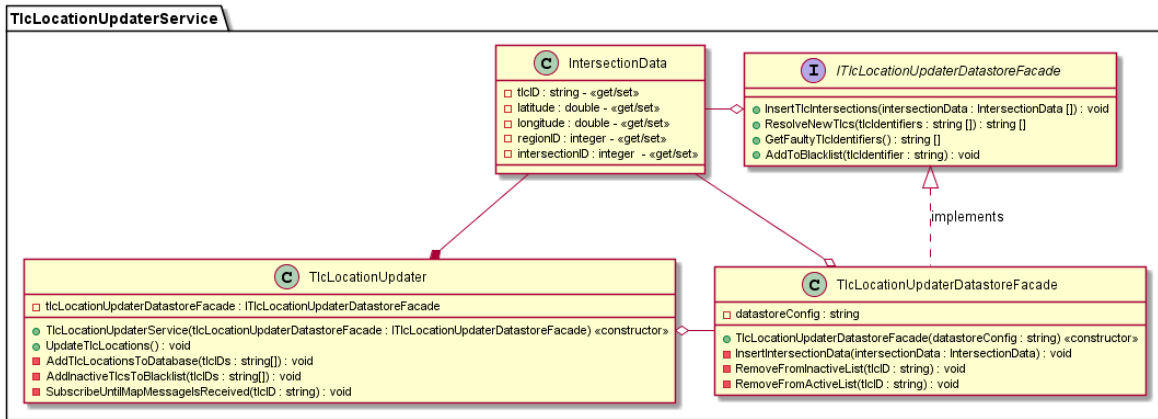


Figure 6.7: TLC Location Updater Service class diagram

Figure 6.7 shows a class diagram of the `TlcLocationUpdaterService`. This service is different compared to the other services, as it does not provide any interfaces to other services. The class `TlcLocationUpdater` will periodically subscribe for updates from all available intersections, and insert the data received into the data store using the `TlcLocationUpdaterDatastoreFacade`.

StatisticsService

This service is responsible for collecting data from the mobile application and storing it in the data store.

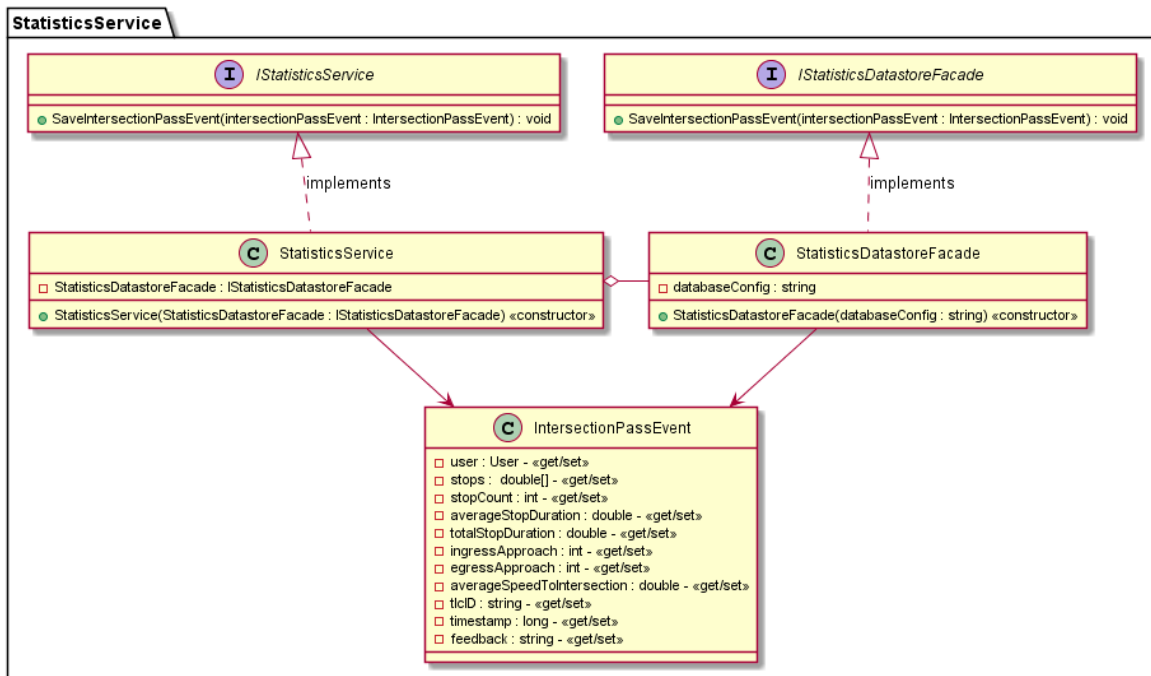


Figure 6.8: Statistics Service class diagram

Figure 6.8 shows a class diagram of the `StatisticsService`, it shows the class `StatisticsService`, which is where the mobile application sends the intersection pass data. The `StatisticsService` uses the `StatisticsDatastoreFacade` to store the data in the data store.

The intersection pass data is also shown in Figure 6.8. Some of the data will be optional while other data will be mandatory. The user identifier will be optional for anonymity, while the user client connection identifier will be mandatory. This means that intersection passes will be relationally connected by a connection identifier. This can become useful when investigating driving patterns. Interestingly as well is that statistics regarding number of stops and the total duration of stops. This can be used to evaluate traffic conditions for drivers. Many stops mean many accelerations, which is economically bad for drivers and bad for the environment.

AuthenticationService

This service has the purpose of providing a means of authentication. This is used by both the mobile client, where users can create an account, but it is also used by the services, which are authenticated, as a way of guarding against illegitimate services accessing the system.

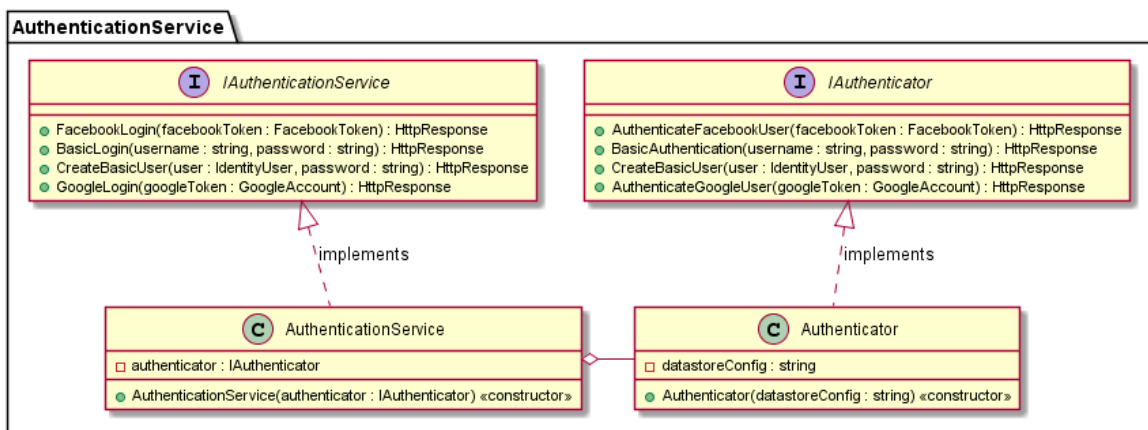


Figure 6.9: Authentication Service class diagram

Figure 6.9 shows a class diagram of the `AuthenticationService`, where the `AuthenticationService` provides functionality for authentication. It needs to check the authentication requests against a data store, which is the purpose of the `AuthenticationDatastoreFacade`.

6.2 Sequence Diagrams

This section will describe the transformations that the design choices has on the three user stories, as well as describing some of the key design decisions made to accommodate the architecture.

Generally, the changes made to these sequence diagrams, compared to the ones shown in chapter 5, is that they now interact with the various services, instead of just the server solution.

6.2.1 User Story 1: Subscribe to Updates

This user story is presented by use of three sequence diagrams.

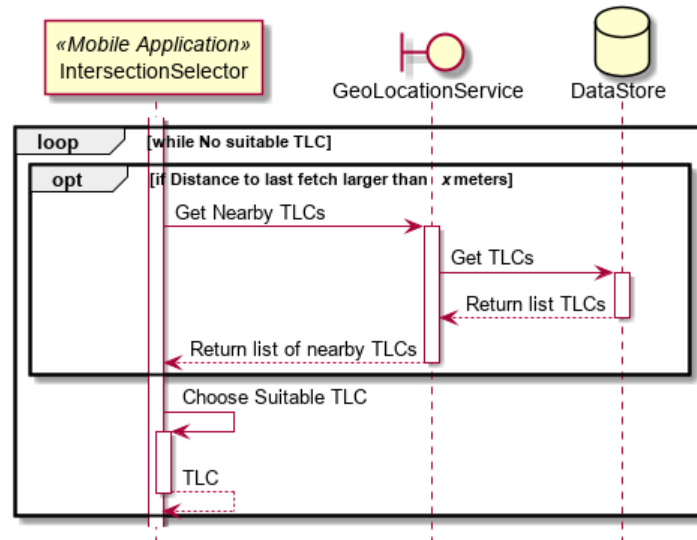


Figure 6.10: Intersection selection sequence diagram

Figure 6.10 shows the process of fetching and selecting an intersection to subscribe to updates from.

Looking at the optional block, if the mobile device moves more than x meters, it is time to update the list of nearby intersections. The `IntersectionSelector` will then send a request to the `GeoLocationService`, to get the nearby TLCs. This service will get the TLCs from the data store and return them.

As the location and status of the mobile user changes, the `IntersectionSelector` will continuously check the set of nearby TLCs for subscription candidates.

The TLC candidates are chosen based on the distance and if it is within 20 degrees angle in front of the mobile device. This makes a cone shaped field, from which the `IntersectionSelector` picks the nearest TLC as a candidate for subscription. This is not a perfect way of selecting TLCs as there is theoretically a risk of subscribing on the wrong TLCs, or unsubscribing by mistake, as shown in Figure 6.11, where the vehicle fails to see the intersection, as it is not inside the field.

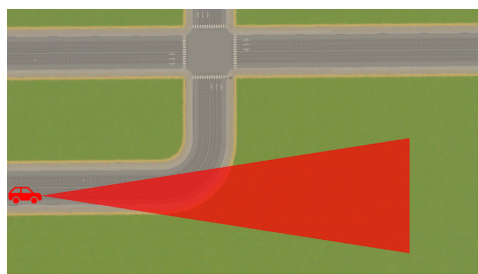


Figure 6.11: Demonstration of the shortcomings of this way of the cone approach

This selection style has been chosen, as it is the simplest and least expensive way of finding the nearest intersection. Other approaches could make use of a route finder API, such as Google Maps, but these are not free. Because of this it was decided to just use the simple and cheap proximity solution.

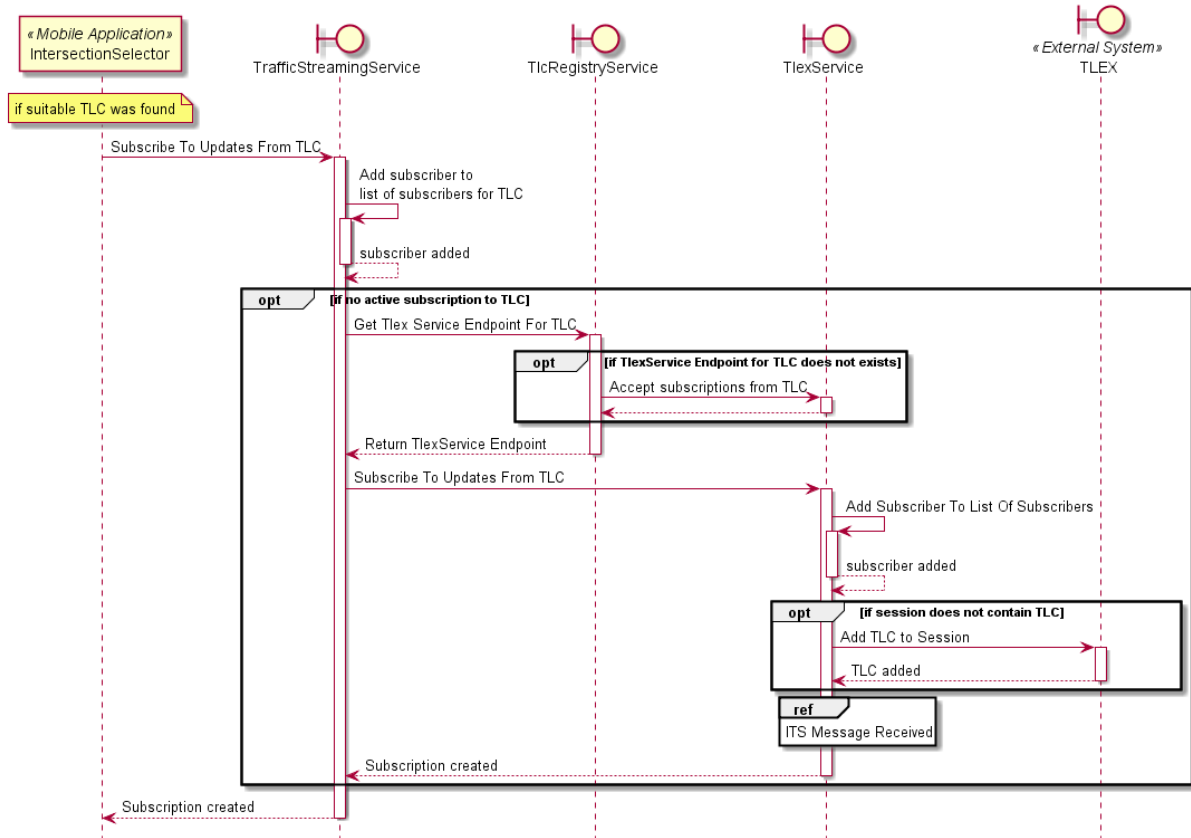


Figure 6.12: Subscribe to updates from TLC sequence diagram

Figure 6.12 shows the second part of the sequence. This diagram shows the process of subscribing to updates to an intersection.

A prerequisite for this sequence is that the `IntersectionSelector` has found a suitable TLC to subscribe to.

When the `IntersectionSelector` has found a suitable intersection to subscribe to, it requests the `TrafficStreamingService` to subscribe to updates for that given TLC. The `TrafficStreamingService` will add the `IntersectionSelector` to a list of subscribers for that TLC.

If the `TrafficStreamingService` instance does not currently have an active subscription for that TLC, it will start one.

The subscription starts by retrieving a `TlexService` Endpoint for the TLC from the `TlcRegistryService`. This step is needed as there is a possibility that another instance of the `TrafficStreamingService` already has an active subscription for the TLC, which means that there should be an instance of the `TlexService` providing data from the TLC.

In case there is no registered `TlexService` endpoints for the TLC, the `TlcRegistryService` will choose one of the `TlexService` instances and register it to the TLC. It then asks that `TlexService` instance to

accept subscriptions from that particular TLC. Finally, the `TlcRegistryService` will return an endpoint of the `TlexService` to the `TrafficStreamingService`.

Then the `TrafficStreamingService` will send a request to subscribe for updates from the TLC to the provided `TlexService` endpoint.

The `TlexService` will add the `TrafficStreamingService` to a list of subscribers. If the `TlexService` does not have the TLC in its session with the TLEX platform, it will try to add it to the session, using a REST API provided by the TLEX platform.

The `IntersectionSelector` is now subscribed to updates from the TLC.

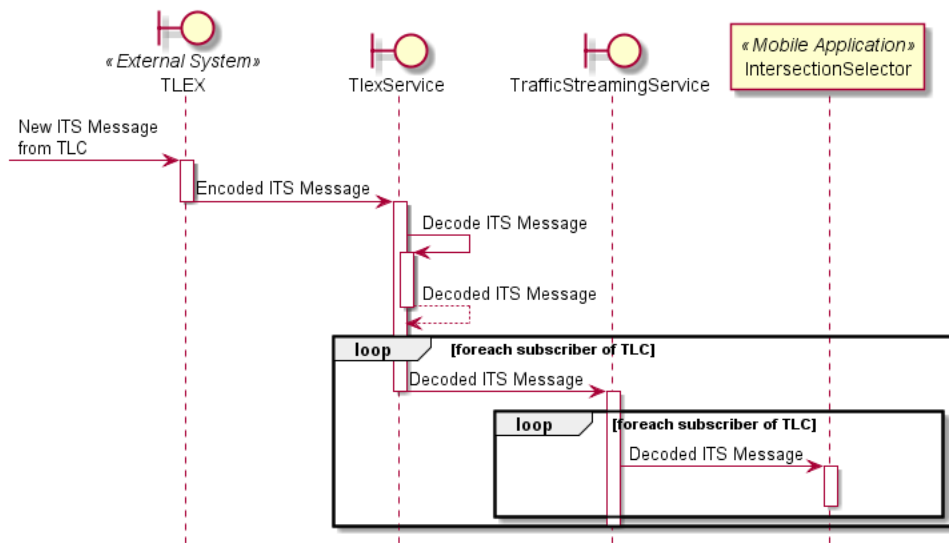


Figure 6.13: ITS message received sequence diagram

Figure 6.13 shows the third part of the sequence. This diagram shows how the ITS messages from the TLC are sent to the mobile application. It starts by the external TLEX platform receiving a message from the TLC, it then sends it to the `TlexService` through a TCP socket connection.

The `TlexService` then decodes the message and sends it to all the `TrafficStreamingService` instances which are subscribing to updates from the TLC. The `TrafficStreamingService` then sends it to all its subscribers on the TLC.

6.2.2 User Story 2: Send Data to Intersection

Figure 6.14 shows the sequence diagram for the process of sending CAM data to the TLC. The prerequisite for this process is that the mobile application is subscribed to and receives data from a TLC.

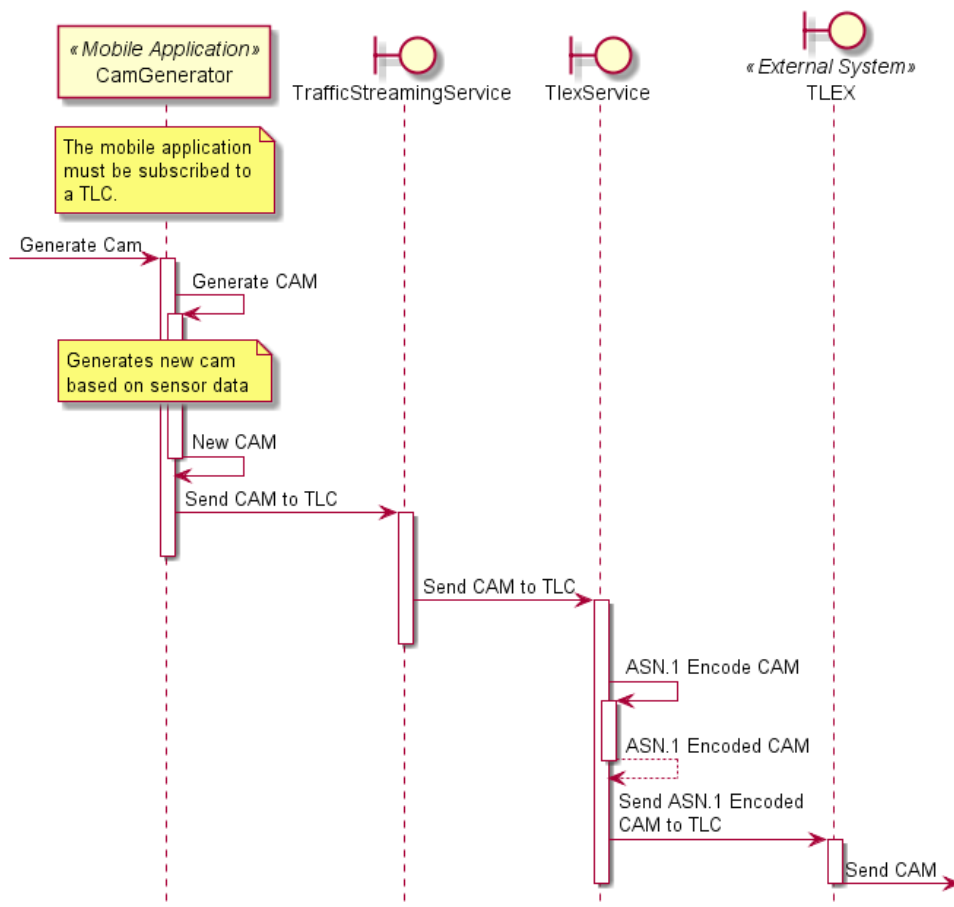


Figure 6.14: Send CAM data sequence diagram

Figure 6.14 is initiated by the **CamGenerator** being told to generate a new CAM message. This request is triggered by a change in location of the mobile device. The **CamGenerator** then generates a new CAM message based on the current location and sensor data available to the mobile device.

The mobile application then sends a request to the **TrafficStreamingService**, and asks it to send it to the TLC, via the **TlexService**, where it is encoded to ASN.1 and passed on to the TLEX platform.

As the CAM is generated from a mobile device, and not directly from the car, the CAM data will be limited to the data which is available to the phone, such as the location and heading. Luckily it is not all the fields of the CAM protocol which are mandatory for a CAM to be valid.

In order to be more power efficient, it has also been decided that the mobile device should act as a tier 2 CAM system¹.

6.2.3 User Story 3: Statistics

Figure 6.15 shows the sequence diagram for the process of saving intersection pass data to the data store.

¹The tier 2 CAM system is described in paragraph 1.2.4

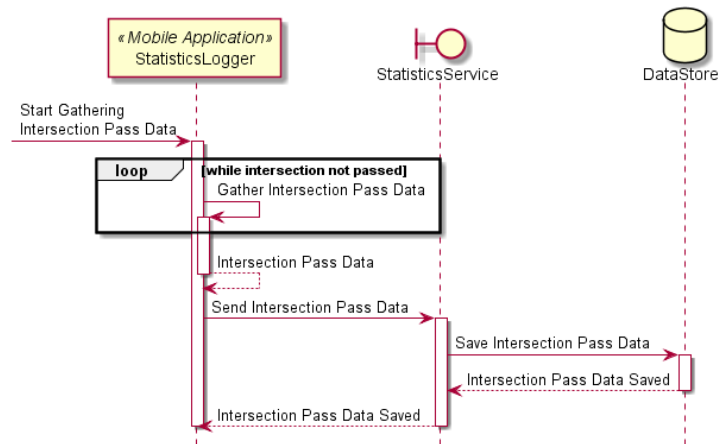


Figure 6.15: Save intersection pass data sequence diagram

In Figure 6.15 it is seen that the `StatisticsLogger` is told to start logging data about the intersection pass. This is triggered by a suitable TLC being selected, which is shown previously in Figure 6.10.

The `StatisticsLogger` then logs data, such as the amount of stops and the duration of these, as long as the distance to the intersection is decreasing. When the distance starts increasing, the intersection is passed, and the `StatisticsLogger` will send the collected intersection pass data to the `StatisticsService`, which will store it in the `DataStore`.

6.3 GLOSA Application

This section will describe the design choices taken with regards to the GLOSA application, and the visualization of the GLOSA speed recommendations.

6.3.1 Selecting the Correct Lanes

In order to give a good speed recommendation, the application must know how long time there is until the intersection changes state. This data comes from the SPaT messages. These messages contains information for all signals in the intersection, also the ones which are irrelevant to the driver. Therefore, it is necessary to find the signals which are relevant and filter out the ones which are not. This is done by grouping the lanes by their approach to the intersection, meaning that the lanes coming from the same direction are grouped. After this the mobile application will choose one of the approaches based on proximity.

An Alternative Lane Selection

This way of selecting lanes could be optimized by looking at other parameters than just proximity. Humans are creatures of habit, and usually, we take the same route every day on our commutes. This means that it could be possible to personalize the lane selection based on historical data for individual users and suggest the same lanes to a driver every day. In order to not suggest the wrong lanes every day, there would need to be implemented validation mechanisms.

6.3.2 Visualization

The visualization style of the GLOSA system is very important, and it must provide information at a glance, in order to be safe to use in the traffic. It was decided that all numbers must be removed from the GLOSA user interface, as there are better ways of communicating the speed recommendations. The choice of visualization is inspired by some of the examples in the related works section. It works by zoning the lanes into planes which are either red or green, indicating that the driver will not be able to pass (red) or that they will be able to pass (green) the intersection in time. These planes shrink in size as the Time To Signal Change (TTSC) decreases.

The planes are added as an overlay to a road, on which the app user drives, and he or she can easily see which plane they are inside. This will grant the app user the option to proactively slow down if they are inside a red plane, to get into a green plane, and pass the intersection.

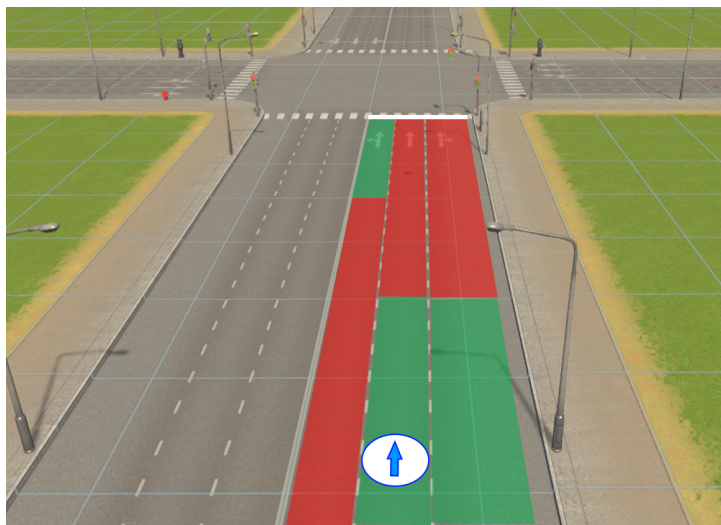


Figure 6.16: Concept design for the GLOSA visualization

Figure 6.16 shows a conceptual drawing of the visualization. It shows the driver driving in a green plane. If the driver were to change lane to the left turn, he would need to slow down, in order to try and reach a green plane further behind.

Calculating Speed Advice Areas

The length of the planes indicates a window of opportunity for crossing the intersection. This window is ever shrinking, as the TTSC is decreasing. The length is not only dictated by the TTSC, but also the speed of the vehicle, as it is possible to drive further in the same time period at higher speeds. Therefore, the length of the planes is simply calculated by multiplying the speed of the vehicle with the TTSC.

Summary: *This chapter has defined the system architecture of the server solution as a microservice architecture. It has introduced the microservices which make up the server solution. It has also shown what impact this architecture choice has had on the central user stories, and which services are relevant for these. Finally, it has presented the chosen approach for the GLOSA application, both for lane selection and for visualization of the speed recommendations.*

7 : Implementation

This chapter will discover the various chosen technologies and describe how these have been utilized in order to realise the system. This chapter will then expand on the previous component diagram, by adding the chosen technologies where they have been utilized. Finally, the chapter will give an overview of the system in its deployed state.

Contents

7.1 Technology choices	35
7.1.1 Distributed Systems Platform	35
7.1.2 Visualization Framework for GLOSA	36
7.1.3 Data stores	36
7.2 Component Diagram	36
7.3 Implementation of User Stories	37
7.3.1 User Story 1: Subscribe to Updates	37
7.3.2 User Story 2: Send Data to Intersection	45
7.3.3 User Story 3: Statistics	47
7.4 Authentication and Authorization	49
7.5 Deployment diagram	50

7.1 Technology choices

This section will describe the technology choices made for the implementation of the system.

7.1.1 Distributed Systems Platform

As it was decided to create the server solution with a microservice architecture, it will be beneficial to find a good container orchestrator. A container orchestrator provides functionality to manage deployment, and the life cycle of the services.

For this orchestrator it was decided to to use Service Fabric [39], as it provides all the desired features, as well as having good support for C#, which was one of the specified requirements. Service Fabric also provides life-cycle methods specifically designed towards the microservice architectures. Appendix E describes other containers orchestrators, which have been considered for the project.

7.1.2 Visualization Framework for GLOSA

The visualization of the GLOSA needs to update rapidly as the driver gets nearer to the intersection and the timing changes. It would be possible to implement the visualization logic for this in android, but it would be far easier to use a game engine for this purpose. Game engines are optimized for drawing graphics, and making multiple updates every second [40].

The choice of visualization framework is Unity [41], as it provides a great set of tools for creating visualizations in 3d. Unity is also very portable, as it is possible to compile to Android, iOS, web and others [42]. Another contributing factor to this decision is that the group has prior experience with Unity, giving it a less steep learning curve.

7.1.3 Data stores

Two technologies are chosen to support the development of the system. Namely MySQL [43] and Elasticsearch [44]. These two technologies fit well into the requirements of the system. MySQL is a good data store for relational data, such as user information. While Elasticsearch is optimized for searching large amounts of information, such as large number of statistics events and log messages.

7.2 Component Diagram

The relationship between the components somewhat differentiates from that of the design section. The main relationships remain the same, however, some relationships are expanded and further defined into specific interfacing technologies.

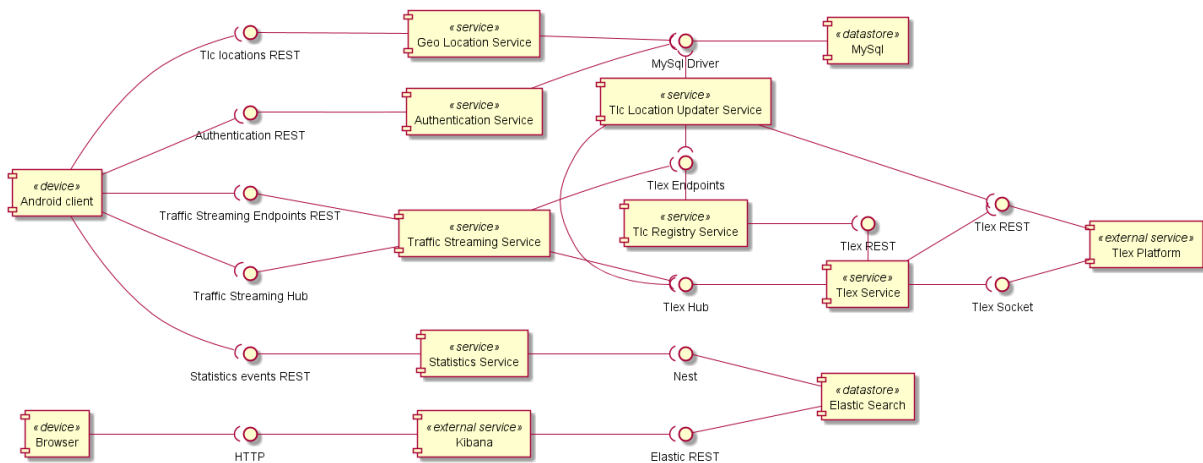


Figure 7.1: Component Diagram of the System
See larger version in appendix H

Figure 7.1 shows the relationship between the components as they are implemented. All the interfaces named *REST* will provide a REST API that can be interacted with using HTTP. Whereas all the interfaces named *Hub* will provide a SignalR hub, which can be interacted with through a WebSocket (WS) connection. The other interfacing technologies will be described later in this section.

The design component diagram suggested having one data store for persisting all the data in the system. The implementation differs from that, by having two completely separate data store solutions. The

first one being a MySQL database, and the second being Elasticsearch. The MySQL database is a good solution for persisting relational data such as which user is authorized for certain roles etc. While the Elasticsearch data store is optimized for searching through huge amounts of information, which is good for statistics and searching through logs. Both of the data stores can be used through third-party libraries. MySQL is queried through a MySQL driver that processes SQL queries [45]. Elasticsearch can be accessed through a Nest package which is able to automatically translate C# model classes into its required format [46].

All the services will send their logs to Elasticsearch through a third-party logging NuGet¹ called Serilog [47]. This relationship is not shown on the component diagram as it would clutter the diagram.

In order to access and process the statistics data and logs stored in Elasticsearch, the design component diagram introduced the **StatisticsWebService**. Instead of creating this service, a decision has been made to use *Kibana* [48].

Kibana is an off the shelf open-source product made by the Elastic team, which means that it is deeply integrated with Elasticsearch. Kibana is made for data visualization and provides a wide range of options to visualize data, making it an attractive solution compared to implementing all kinds of visualization methods from the bottom up.

The session between the **TlexService** and the **TLEX Platform** is created through a REST endpoint, where it gets the connection details. These details are then used to establish a TCP socket connection, for exchanging ITS messages. This session is updated over time through the REST endpoint as different data needs arise, specifically based on the intersection data subscriptions from the app users.

7.3 Implementation of User Stories

This section will go into greater detail of the implementation of Service Fabric services, and describe the implementation logic required to support the three central user stories.

Appendix F contains a short description of Service Fabric concepts. These concepts are useful in the following section. Even more information about Service Fabric can be found on their website² [39].

7.3.1 User Story 1: Subscribe to Updates

The logical process of the user story has been described in the design subsection 6.2.1. This section will describe how this process is implemented using Service Fabric.

The first of the services that is called upon from the client application is the stateless service **GeoLocationService**.

To better explain the implementation details of stateless services, all the Service Fabric terms and methodologies will be explained in regards to the **GeoLocationService**. The following service explanations will not go into the same depth as the **GeoLocationService** description, as they are nearly identical, apart from the **TlcRegistryService** as it is a stateful service.

The Stateless **GeoLocationService**

The **GeoLocationService** will access a *MySQL* database, which contains location detail for all of the TLCs that are available through the TLEX. This means that the **GeoLocationService** does not need to keep any state, and it can be implemented as a stateless service.

¹NuGet is the package manager for .NET

²Service Fabric website: <https://docs.microsoft.com/en-us/azure/service-fabric/>

Stateful services are harder to manage, as the state of the service instance needs to be replicated across multiple replicas, and it is more complicated to automatically scale.

The starting point of a Service Fabric service is the `Main` method in the `Program.cs` file in the service project. This `Main` method registers the service to the Service Runtime in the Service Fabric cluster, by providing a service type name and an instance of a service class that extends either `StatelessService` or `StatefulService`.

By extending either one of these specific types the implementing class will inherit several life-cycle methods which will be called by Service Fabric. These methods are listed below:

- `CreateServiceInstanceListeners()` : `IEnumerable<ServiceInstanceListener>`
- `OnOpenAsync(CancellationToken cancellationToken)` : `Task`
- `RunAsync(CancellationToken cancellationToken)` : `Task`
- `OnCloseAsync(CancellationToken cancellationToken)` : `Task`

The two most important life-cycle methods are the `CreateServiceInstanceListeners` and `RunAsync` methods. However, the `OnOpenAsync` and `OnCloseAsync` can be relevant as well for opening and closing resources or doing other work that is required before and after the `RunAsync` method is executed.

The `CreateServiceInstanceListeners` method is used to setup the communication endpoints of the service. The `RunAsync` method is used as the main method of the service, and this is where the service should begin its continuous work if this apply, which it does not for the `GeoLocationService`.

The Communication Listener is created by the `CreateServiceInstanceListeners` method. Custom communication listeners can be implemented by implementing the `ICommunicationListener` interface, this makes it possible to create any type of networked listener through *TCP* and *UDP*. However, all the services in the server solution make use of the `KestrelCommunicationListener`, which is the default communication listener for Service Fabric services. This communication listener is built on the ASP.Net Core Kestrel web server implementation and has support for *HTTP*, *HTTPS* and *WS* communication, which are the only communication protocols needed for the services in the server solution.


```

1  protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
2  {
3      return new[]
4      {
5          new ServiceInstanceListener(serviceContext =>
6              new KestrelCommunicationListener(serviceContext, (url, listener) =>
7                  {
8                      Log(LogEventLevel.Information, "Starting Kestrel on {0}", url);
9
10                     return new WebHostBuilder()
11                         .UseKestrel()
12                         .UseServiceFabricIntegration(listener,
13                             ServiceFabricIntegrationOptions.UseReverseProxyIntegration |
14                             ServiceFabricIntegrationOptions.UseUniqueServiceUrl)
15                         .UseUrls(url)
16                         .ConfigureServices(
17                             services => services
18                                 .AddSingleton(_logger)
19                                 .AddSingleton<ITlcManager>(_tlcManager)
20                                 .AddSingleton<ServiceContext>(serviceContext)
21                                 .AddSingleton<IServicePartition>(Partition))
22                         .UseStartup<Startup>()
23                         .Build();
24                  })
25      };
26 }

```

Code Snippet 7.1: Creating a Kestrel communication listener

Code Snippet 7.1 shows the `CreateServiceInstanceListeners` method implementation for the `GeoLocationService`. An instance of `KestrelCommunicationListener` is created, on line 6, and a `WebHost` is given to the communication listener. The `WebHostBuilder` is used to configure the communication listener. Line 11, `UseKestrel()`, tells the communication listener to use the kestrel middleware³.

Lines 12 and 15, `UseServiceFabricIntegration(...)` and `UseUrls(...)`, prepares the listener for Service Fabric. These will assign a unique path to the service. This preparation is important, as Service Fabric is able to host multiple services, and even multiple instances of the same service, on a single machine. This means that each service instance will, if not prepared, attempt to listen on the same port. The Service Fabric middleware will grant the service listeners unique endpoints in the following format: `protocol://ip:port/partition-uuid/instance-id`, resulting in an URL like: <http://185.92.0.0:49974/b7a1a02f-0ce7-42f6-84f4-1f45f6dc36e1/132023547891537536>

In this case we want Service Fabric to dynamically assign an available unique port to the service, however, it is also possible to declare a predefined value for the port. This is particularly useful for internet-facing services, which are services where the typical network traffic comes from end-users. The `GeoLocationService` is in fact an internet-facing service that should be accessible by app users. Therefore a predefined port would be a good choice, however, the service takes advantage of the reverse proxy provided by Service Fabric, which redirects the user from a predefined service endpoint to the endpoint of a service instance, this will be explained in more detail later in this section.

The last setup of the `WebHostBuilder` in Code Snippet 7.1 is the `ConfigureServices(...)` on lines 16-21 and `UseStartup()` on line 22. The `ConfigureServices(...)` method allows the service to inject different objects into a Dependency Injection (DI) framework. In this case instances of the logger, `ITlcManager`, `ServiceContext` and `IServicePartition` are injected into the framework. This can, however, also be done inside the `Startup` class declared by the `UseStartup()` method. Which has the single purpose of setting up additional middleware and adding services to the DI framework. More details on the `Startup` class will follow.

³As a web server as opposed to *HTTP.sys*, which is an older web server implementation.

The Service Fabric Reverse Proxy solves the problem of hosting multiple services and service instances on the same machine. It acts as a middleman and connects the unique endpoints of the service instances with a service specific endpoint, like <http://185.92.0.0:19081/SwarcoTrafficTimer/GeoLocationService/api/route>⁴. The reverse proxy will redirect the request and to one of the service instance endpoints described earlier.

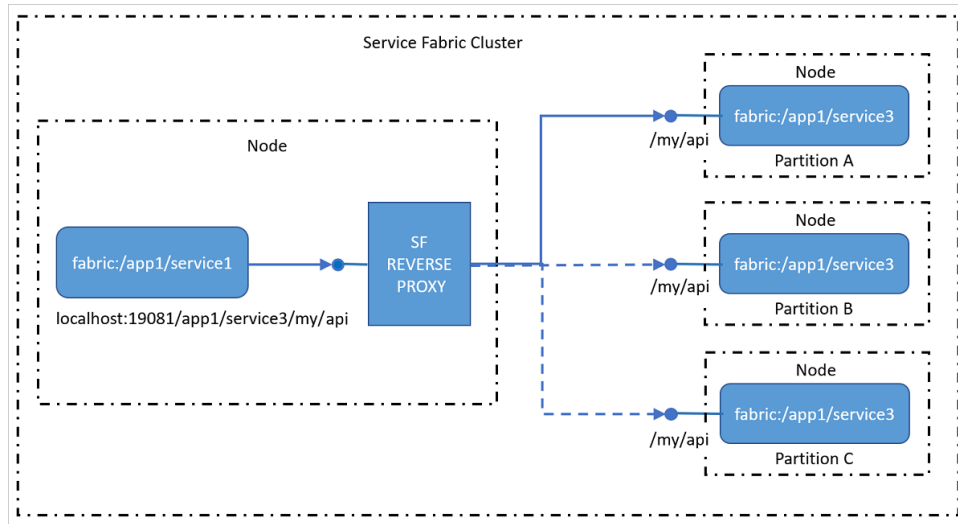


Figure 7.2: Using a reverse proxy to reach a service instance
source: [49]

Figure 7.2 shows an example scenario using a reverse proxy. It is seen how left-hand node tries to reach an endpoint with an application and service name `/app1/service3` using an API `/my/api`. Instead of trying to reach the endpoint directly, the left-hand node queries the reverse proxy at port 19081. By the given endpoint information, the reverse proxy will be able to resolve a service instance and redirect the request from the left-hand node to one of the three right-hand nodes.

The service instance is selected using a round-robin approach. This means that all the service instances will get an equal number of requests. This is both good and bad as the nodes, in which the services are hosted, may have varying hardware specifications. This could be solved by using a load balancer, but this has not been implemented in this project.

Not only will the reverse proxy select a service instance and redirect the request. The reverse proxy also implements retry logic in the case of connection failure. This means that the clients will always be directed to healthy service instances, which answer the request.

The Startup Class, which is declared in the `WebHostBuilder`, allows developers to easily add dependencies and middleware as described by the OWIN (Open Web Interface for .NET) specification [50]. The core concept of which is to decouple server and application. This is a good place to setup the server part of Service Fabric services.

To provide a REST interface the `GeoLocationService` uses the Asp.net Core Web API which is part of the MVC framework which is built into .Net Core [51]. This framework needs to be initialized and configured inside the Startup class.

⁴explanation: `<protocol>://<ip>:<reverse proxy port>/<applicationtype name>/<servicetype name>/<api route>`

```
1 public class Startup
2 {
3     // Use this method to add services to the container.
4     public void ConfigureServices(IServiceCollection services)
5     {
6         services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
7     }
8
9     // Use this method to configure the HTTP request pipeline.
10    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
11    {
12        app.UseMvc();
13    }
14 }
```

Code Snippet 7.2: Adds the MVC middleware to the service

Code Snippet 7.2 shows the two methods that will be called by the framework in the Startup class: `ConfigureServices(...)` and `Configure(...)`. The configuration of services is done in the `ConfigureServices(...)` method on line 4. Any service that is added to the `IServiceCollection` will be available to inject into the application code⁵. The `Configure(...)` method, on line 10, is used to configure the HTTP pipeline of the web server. This is where the service can configure routing⁶.

With the configuration of the Startup class, the reverse proxy, the communication listener and the necessary configuration to the Service Fabric cluster the Service Fabric service is now ready to be deployed.

Asp.Net Core is a tool for developing web servers, APIs and websites. The `GeoLocationService` takes advantage of the web API functionality of the Asp.Net Core package. To show how REST APIs are implemented throughout the server solution the API controller of the `GeoLocationService` is highlighted.

⁵An example of this will be shown in the REST controller of the `GeoLocationService`

⁶This will be explained later with an example of the `TrafficStreamingService` on page 43

```

1 [Route("api/[controller]")]
2 [ApiController]
3 public class TlcController : Controller
4 {
5     private readonly ITlcManager _tlcManager;
6
7     public TlcController(ITlcManager tlcManager)
8     {
9         _tlcManager = tlcManager;
10    }
11
12    [HttpGet("near")]
13    [AllowAnonymous]
14    public async Task<ActionResult<List<Map>>> GetNearbyTlcs(CancellationTok
    ↪ cancellationToken, double latitude,
15    double longitude, double distance = TlcManager.DefaultNearbyTlcDistanceInMeters)
16    {
17        try
18        {
19            var tlcs = await _tlcManager.GetNearbyTlcLocations(cancellationToken, latitude,
    ↪ longitude, distance);
20            return Ok(tlcs);
21        }
22        catch (Exception e)
23        {
24            return BadRequest(e);
25        }
26    }
27 }

```

Code Snippet 7.3: Creates a REST API endpoint

Code Snippet 7.3 shows how an endpoint is created using the Asp.Net Core web API. This code defines a REST endpoint with the following route: `.../api/Tlc/near?latitude=0&longitude=0&distance=0`.

On line 1, at the top of the `TlcController` class, the attribute `Route(...)` declares the route of the endpoint. The "[controller]" string will be replaced by the actual controller name prefix, `Tlc` in this case. The `ApiController` attribute, on line 2, registers the class as an API controller in the Asp.Net framework, and makes sure that the class will automatically be loaded by the MVC middleware, that was configured in the Startup class.

The constructor, on line 7, takes an instance of `ITlcManager` in its parameters. This is automatically passed to the controller, through the built in DI framework in Asp.Net Core. The instance of the `ITlcManager` was passed to the DI framework, when the service communication listener was created, as seen in Code Snippet 7.1.

Two attributes are declared above the API method on line 14, `HttpGet(...)` and `AllowAnonymous`. The first attribute is used to declare the method as a REST get method. The second attribute is used to disable the default authentication behavior. Without this attribute the user must include an authentication header in their HTTP request. More information about how the authentication of the system works can be read in section 7.4.

The logic of the web API controller is simple, shown on line 19. All it does is invoke the `GetNearbyTlcLocations` of the `ITlcManager`. This method will query a MySQL database for intersections within the given distance of the given location. If it is successful, it will return a HTTP `Ok` response to the client.

The result of the `GeoLocationService` is that the client will have a list of nearby intersections. This list of intersections must then be evaluated by the client in order to tell exactly which intersection the

client is driving towards. This leverages some processing power from the mobile devices, in order to keep processing power demand for the service at a minimum.

Real-time Data Exchange

When the client has selected which intersection it is headed towards, it will subscribe to updates from the intersection. This is done by providing the TLC identifier for the intersection, to the `TrafficStreamingService`. This service will then make sure that the client receives data from the intersection, as well as forwarding any CAM that the client may send to the intersection.

Given that there must be a two-way real-time exchange of information between the client and server, it makes sense to use a communication protocol that support full duplex. It also makes sense to use a communication protocol that is optimized for sending frequent and small messages. So based on the findings from the networking prototypes made (seen in appendix G) it was decided to use the WS protocol for communication between the services. As a choice of technology SignalR [49] was chosen. SignalR has great support for .Net Core and other Microsoft frameworks, such as Service Fabric and Identity Core⁷.

The following subsections will go into detail about how a connection is made from an Android client to a `TrafficStreamingService` instance. The internal technology required to accomplish the connection to the TLEX platform and exchange live traffic data will also be explained. Finally, the visualization of a speed recommendation will be presented.

When Using SignalR in Asp.Net Core the first thing that must be done is configuring the Startup class to include SignalR. The required setup resembles that seen in code snippet Code Snippet 7.2. In order to add the SignalR service, simply add `services.AddSignalR()` to the `ConfigureServices(...)` method, and configure the HTTP pipeline in the `Configure(...)` method. This looks a little different than the mentioned code snippet. Instead of simply saying `app.UseSignalR()` it is also required to define a route to the SignalR hub. A SignalR hub declare the methods which a client will be able to invoke.

All a class needs to do to become a SignalR hub is to extend the `Hub` class. By doing so all the public methods within the class will be invocable through client connections. However, authentication and authorization rules do apply to these methods, so it is possible to restrict public methods to only be available for certain users and user-roles.

The `TrafficStreamingService` has a method called `SubscribeToUpdatesFrom(TLC)`. This method should be invoked by a client when the client wishes to start a live exchange of data with an intersection. The stream of data will be open for as long as the connection is open or until the client invokes the `UnsubscribeToUpdatesFrom` method on the hub. When either method is invoked the `TrafficStreamingService` must internally ask the `TlcRegistryService` to provide a `TlexService` endpoint for the given TLC identifier. Reason being that the `TlexService` is responsible for establishing the connection between the cluster and the TLEX platform.

Connecting to a TlexService by use of the TlcRegistryService

Figure 6.12 (in the Design chapter on page 30) shows the expected behavior when subscribing for intersection data as a sequence diagram. This section will describe how services within a cluster communicate with each other, to realise this behaviour.

The `TlcRegistryService` is a stateful service, as there it is needed that the state, which contains the TLC registry, must be persisted.

⁷Which is used for authentication as seen in section 7.4

For the `TrafficStreamingService` to query the `TlcRegistryService` an instance endpoint must be resolved. To accomplish this the `ServicePartitionResolver` helper class in the Service Fabric package is used.

```

1 private readonly ServicePartitionResolver _resolver = ServicePartitionResolver.GetDefault();
2
3 private async Task<Uri> GetTlcRegistryService()
4 {
5     // The Stateless Service Context is provided by the class constructor
6     var tlexUri = new Uri(Context.CodePackageActivationContext.ApplicationName + "/" +
7     ↪ TlcRegistryServiceName);
8     var partition = await _resolver.ResolveAsync(tlexUri, new ServicePartitionKey(0),
9     ↪ CancellationToken);
10
11     var primaryEndpoint = partition.Endpoints.FirstOrDefault(p => p.Role ==
12     ↪ ServiceEndpointRole.StatefulPrimary);
13     if (primaryEndpoint == null)
14     {
15         throw new Exception("Unable to determine endpoint for Tlc Registry Service");
16     }
17
18     var jobject = JObject.Parse(primaryEndpoint.Address);
19     var first = jobject?.First.First;
20     var endpointString = first?.Value<string>("");
21     return new Uri(endpointString);
22 }

```

Code Snippet 7.4: Resolving a service instance endpoint to the `TlcRegistryService`

Code Snippet 7.4 shows how the service partition resolver, declared in line 1, is used to resolve an endpoint to a `TlcRegistryService` instance. In order to query the Service Fabric cluster, the application name and service name must be provided. The application name is fetched from the service context at line 6. This assumes that the service we want to target is hosted on the same application as the caller, which is true for the case of `TlcRegistryService` and `TrafficStreamingService`. The service name is stored as a constant variable, as it does not change. Now with an application name and service name the `ServicePartitionResolver` can get the endpoint of the `TlcRegistryService`, as shown in line 7. The resolver also need a service partition key to resolve the endpoint, which is needed to access the correct partition. Even though the `TlcRegistryService` is in a ranged partition, it only ever has a single partition by design. Hence, the key passed is `0`.

Lines 9-13 shows how the response from the resolver is filtered from a list of the discovered endpoints. We are only interested in the primary endpoint, of which there will only ever be 1 or 0. If no primary endpoint was found it throws an exception which is propagated through to clients.

Lines 15-18 shows how an endpoint URI is extracted from the endpoint address. Which the Service Fabric team saves as a JSON string instead of parsing it to a model object.

Given the endpoint URI to a `TlcRegistryService` it is now possible to connect to the service instance and query its REST API for the `TlexService` endpoint for the given TLC identifier.

After querying the `TlcRegistryService` we will now have the endpoint URI to a `TlexService` instance. This endpoint URI is used to establish a SignalR WS connection to a `TlexService` instance from the `TrafficStreamingService`. The `TlexService` has a hub with the behavior almost identical to that of the `TrafficStreamingService`. The implementation reason for this behavior duplication stems from the limitations from the TLEX platform. Only one connection per intersection can be established with the same TLEX user, which leaves no room for session redundancy.

Exchanging data with an intersection through the TLEX platform

When a client subscribes to data from an intersection, in the `TrafficStreamingService`, internally the `TlexService` will be prompted to start a session with the TLEX platform. The TLEX platform is a multiplexer of intersection data. All intersections connected to the platform will continuously send Map and SPaT data to the TLEX platform. This data will then be forwarded, by the TLEX platform, to the `TlexService`. Details of this process is described in subsection 6.1.2.

Visualizing the speed recommendation

The visualization of the speed recommendation is done through an Android client with a Unity application. This client will query the `GeoLocationService` and the `TrafficStreamingService` in order to get the intersection data that is required for providing the speed recommendation. The above sections have covered the internal working on the server for exactly that.

The Android application consists of two parts, an Android part, and a Unity part. The Android part is responsible for communicating with the server, getting location updates, finding intersections, and sending CAM to the server. It is also responsible for sending data to the Unity part, which is used for calculating the GLOSA speed advice, and visualizing it.

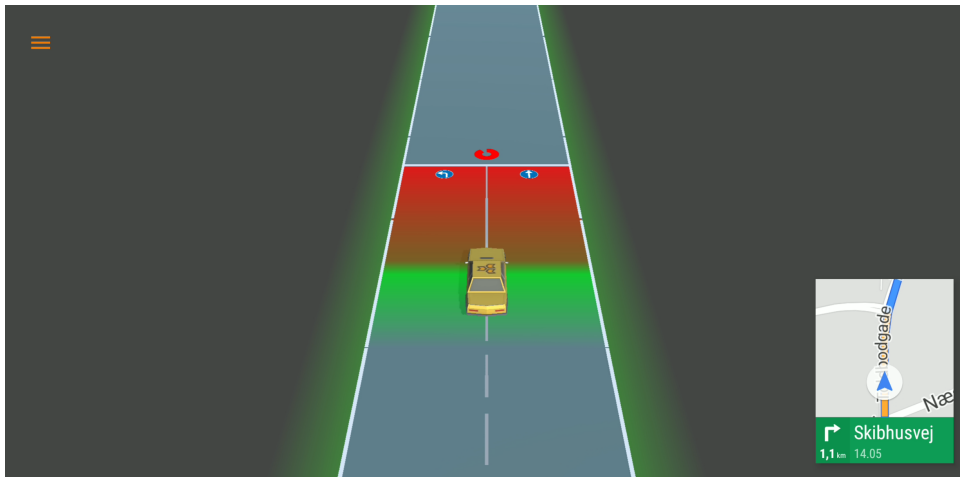


Figure 7.3: Shows the Unity visualization inside an Android app while Google Maps runs picture-in-picture in the bottom right corner

In Figure 7.3 the implementation of the visualization, that is described in the design chapter, is shown. The graphical interface shows a car on the road that symbolizes the position of the user. The car will remain stationary as the horizontal line, representing an intersection, move downwards in a speed set by the GPS location updates received on the Android device.

The implementation of the visualization differentiates from the design description by also including a countdown progress bar. This circular progress bar appears above the intersection line as the signal groups, ahead of the driver, changes from green to red.

7.3.2 User Story 2: Send Data to Intersection

When the driver is heading towards an intersection, the Android client will send CAM data to the intersection. This data is sent each time the client gets a location update.

```

1 private void constructCam(Location location) {
2     lastCamGenerationTime = Calendar.getInstance();
3
4     CoopAwareness ca = new CoopAwareness();
5
6     int generationDeltaTime = (int) ((location.getTime() - Time2004) %
7     ↪     GENERATIONAL_TIME_WRAP);
8     ca.setGenerationDeltaTime(generationDeltaTime);
9
10    CAMParameters camParams = new CAMParameters();
11
12    // Basic Container
13    BasicContainer bc = new BasicContainer();
14    bc.setStationType(mVehicleType);
15
16    ReferencePositionWithConfidence rp = new ReferencePositionWithConfidence();
17    rp.setDeltaAltitude((long) (location.getAltitude() * 100));
18    rp.setDeltaLatitude((long) (location.getLatitude() * 10000000d));
19    rp.setDeltaLongitude((long) (location.getLongitude() * 10000000d));
20    rp.setDeltaAltitudeConfidence(location.hasVerticalAccuracy() ?
21    ↪    CamUtilities.getAltitudeConfidence(location.getVerticalAccuracyMeters()) :
22    ↪    AltitudeConfidence.unavailable);
23
24    PosConfidenceEllipse pce = new PosConfidenceEllipse();
25    pce.setSemiMajorConfidence((int) (location.getAccuracy() * 100));
26    pce.setSemiMinorConfidence((int) (location.getAccuracy() * 100));
27    pce.setSemiMajorOrientation(0); // As we are not constructing an ellipse, but a circle,
28    ↪    the orientation does not matter.
29    rp.setPositionConfidenceEllipse(pce);
30
31    bc.setReferencePosition(rp);
32
33    camParams.setBasicContainer(bc);
34
35    ca.setCamParameters(camParams);
36
37    sendCam(ca);
38 }

```

Code Snippet 7.5: Generate CAM

Code Snippet 7.5 shows the process of generating the CAM data in the Android client. It starts, on line 2, by updating a timestamp which makes sure that the data is sent no more than once each second. It then creates a `CoopAwareness` instance, on line 4, which is the CAM object. Then it calculates the Generational Delta Time, on line 6, which according to the CAM specification is the *milliseconds since Jan. 1st 2004 modulo 65536* [23]. This odd timestamp is probably used for determining the order and delta time of received messages, and not the timestamp it self.

On line 9 the snippet goes on to create the `CamParameters`, which contains the four containers in the CAM data. Line 12 shows how the basic container is instantiated. Line 15-21 shows how the delta position data is drawn from the Android `Location` object. The altitude, latitude and longitude are then converted to the correct formats, and the altitude confidence is set. Lines 23-26 shows somewhat the same thing, but for the confidence ellipse, which are given a circular shape, instead of an ellipse, as the Android `Location` object only provides one axis of confidence. This also means that it is irrelevant to set an angle, because circles are the same no matter the angle.

The rest of the containers are omitted in order to make the snippet shorter. The process of inserting the data is the same throughout the method, *get* → *convert* → *set*.

The snippet ends by setting the `CamParameters` on the `CoopAwareness` object on line 32, before sending it to the server solution on line 34.

The CAM data passes through the `TrafficStreamingService`, to the `TlexService`, where it is encoded to ASN.1 format and packaged into a message frame according to the TLEX specifications, before it is sent to the TLEX.

7.3.3 User Story 3: Statistics

As the app user drives through an intersection, a background android service will log statistics about their pass through the intersection. These statistics will be sent to the `StatisticsService` which then stores it in *Elasticsearch*.

Data Visualizations in Kibana

The following visualizations shown in this section, are based on simulated intersection pass data, in order to give a better overview of how these can be useful to customers.

Several visualizations of the data can be presented to administrative users and customers. One of which is the visualization of which ingress drivers entered, and which egress they exited an intersection.

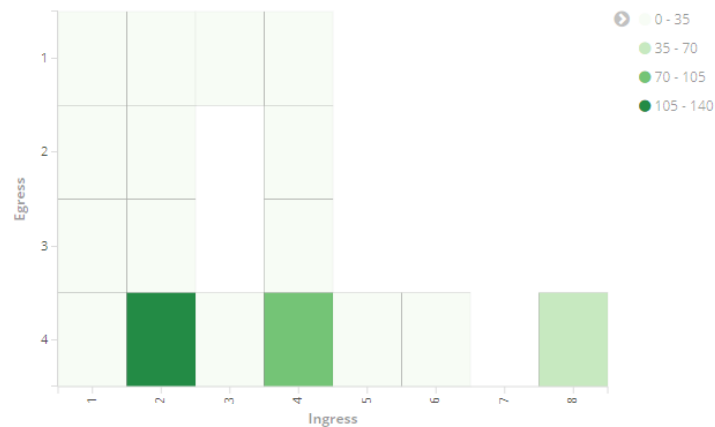


Figure 7.4: Kibana visualization of egress & ingress approaches from intersection passes

Figure 7.4 shows all the intersection passes made through one intersection. Each logged intersection pass contain information about what ingress the driver entered the intersection through and what egress the driver exited the intersection through. The figure is useful for discovering which way the average driver is headed. In this sample we see that many drivers enter the intersection through approach numbers two, four, and eight. All the three ingress approaches mostly exit the intersection through egress approach number four. This would let customers know that most drivers, passing through the intersection, are headed towards egress number four. The numbers are represented in the Map data and the customers would have access to information about which number points to which physical ingress and egress approach.

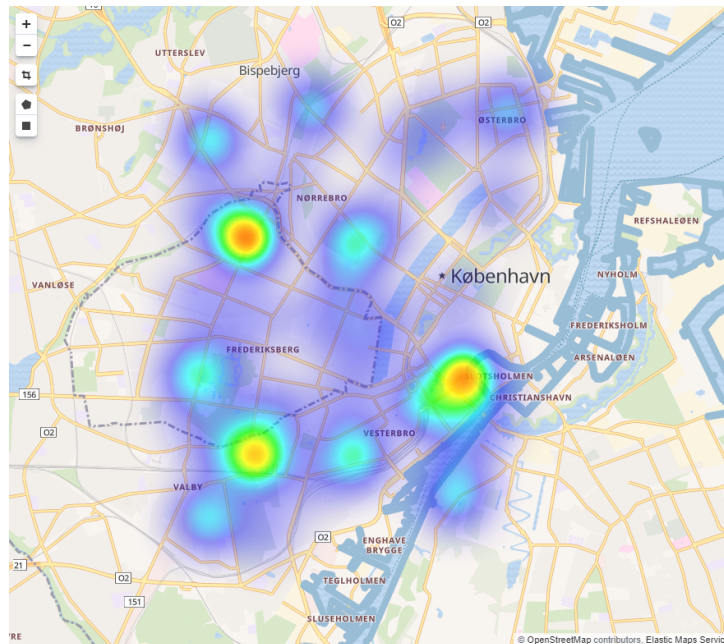


Figure 7.5: Heatmap of intersection passes in Copenhagen, Denmark

A way of visualizing which intersections are more utilized can be done by use of a heatmap. The heatmap in Figure 7.5 shows all the traffic within a certain geographic area and time period. The warmer color the area has, the more traffic there is in that area.

This can be used by customers to discover which intersections are under heavier load, and together with the previous visualization, it can be used to discover where the drivers are coming from, and going to.

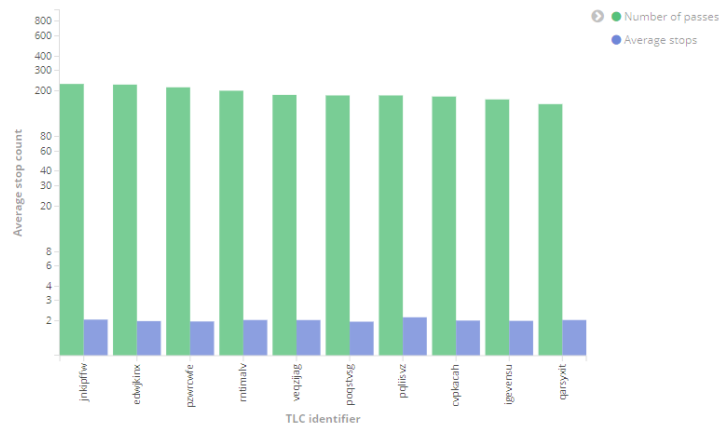


Figure 7.6: Number of intersections passes and number of stops by intersection

Figure 7.6 captures the number of intersection passes and stops. The data that is visualized is based on randomly generated data, and is thus uniformly distributed, meaning that all intersections have almost the same amount of intersection passes and stops. However, given real data this visualization would be able to highlight which intersection has more intersection passes and which intersection has more

stops. This can be useful for discovering the most utilized intersections, as well as poorly performing intersections.

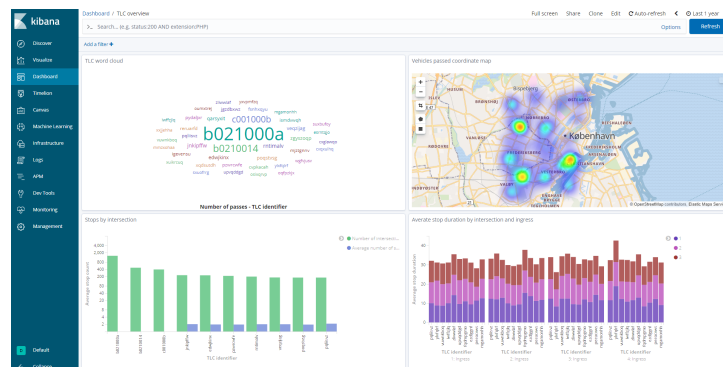


Figure 7.7: The TLC overview dashboard from Kibana

It is possible to create different dashboards in Kibana, which will provide an overview of many visualizations at once, with different perspectives. Figure 7.7 shows the ‘TLC Overview’ dashboard, which provides all kinds of information on the most frequently used intersections in real time.

Several other visualizations can be provided to customers and administrative users through Kibana. Thus, giving the customers more insight into the traffic conditions in the real world. Capturing this sort of information in a digital format makes it less time consuming and less costly, than capturing the information through physical means. It is not unheard of to station a person physically in the world to get this type of information.

7.4 Authentication and Authorization

Authentication and authorization are vital concepts for an application that should handle user information in a secure manner. The system solves this problem using the *.Net Core Identity* membership system.

This membership system supports *basic authentication* as well as *OAUTH 2.0 authentication* using bearer tokens. In order to accomplish the implementation of a membership system, the `AuthenticationService` was created. More importantly the service was created as an ASP.NET Core MVC service with `Individual authentication` checked.

By having this authentication method checked the creation of the service would include a series of template files used for *.Net Core Identity*. These template files must be used with the `dotnet database migration tooling`. The result of which is to initialize a database scheme to support the authentication and authorization. The target database had to be a relational database and it was decided to use MySQL as it was free to use and the development team knew how to navigate it.

After performing the initialization from the template files a series of tables were created in MySQL. The most important of which are the `aspnetusers` and `aspnetuserlogins` tables, as these two tables will store the user information as well as their logins. However, other tables do exist for roles, tokens, and for making relationships between tables. The `aspnetusers` table will by default contain usernames, emails, phone numbers etc., but more fields can be added as required. The default fields were sufficient for the project.

With the database in place the implementation of the actual authentication could take place. The goal in which is to enable authentication and authorization to internet-facing services and for inter-service

communication. It should be achieved using bearer tokens [52] wrapped in a JSON Web Token (JWT) [53]. Whenever a user logs into the server a JWT is sent in response that the user must use for future requests. The JWT describes all the claims the user has and is signed using a secret available only to the `AuthenticationService` [52, 53].

The `AuthenticationService` has support for two external OAUTH 2.0 providers, which are Facebook and Google, as well as basic login by username and password. When a user wants to login using an external OAUTH provider they must forward an external access token, from a successful login at an external OAUTH provider, to the system. Upon receiving an external access token at the server, it is validated by making an HTTP request to the OAUTH provider in question. If the external access token is valid then a login is done and a new access token is generated. This access token authorizes the user to interact with the system.

ASP.Net MVC Service Authentication

All the services that requires authentication must include some configuration in their Startup class. Particularly the services must add the authentication dependencies in the `ConfigureServices` as well as change their `services.AddMvc()` method to include an authorization filter. The filter will ensure that, by default, any endpoint requires authentication. The individual endpoints must then explicitly allow anonymous users if they wish.

The authentication dependencies must be configured to authenticate users using the JWT bearer scheme. In order to validate a JWT, the secret that was used to sign the JWT must be used to decrypt the JWT. An additional hook must also be configured to support SignalR WebSocket authentication headers.

The `AuthenticationService` required a bit of additional setup in the Startup class as this service will need access to the database in order to authenticate users. Specifically a `DbContext` object must be created using a connection string to the database as described in the beginning of this section [54].

Internal Inter-Service Communication

Certain endpoints should only be available internally. To support this requirement an extension of the stateless and stateful service classes have been derived into an "authenticated" version. Normally a Service Fabric service must either derive from `StatelessService` or `StatefulService`, however, to be authenticated instead they must derive from the custom `AuthenticatedStatelessService` or `AuthenticatedStatefulService`. The implementation of the two are the same, though it was necessary to make two classes as `StatelessService` and `StatefulService` does not share any interface or base class.

By deriving from an authenticated service class a new method will be added to the life-cycle. The method extends the `RunAsync` method with an `ITokenProvider`. This method is called after the regular `RunAsync` method completes. The regular `RunAsync` method requests an access token from the `AuthenticationService` using a basic login. The result is that the authenticated service will receive a callback when it is authenticated, and thus will be able to invoke internal endpoint.

7.5 Deployment diagram

With the implementation details in place the system must now be deployed to a production environment. This includes the setup of three Service Fabric nodes and a machine for hosting Elasticsearch, Kibana, and MySQL server.

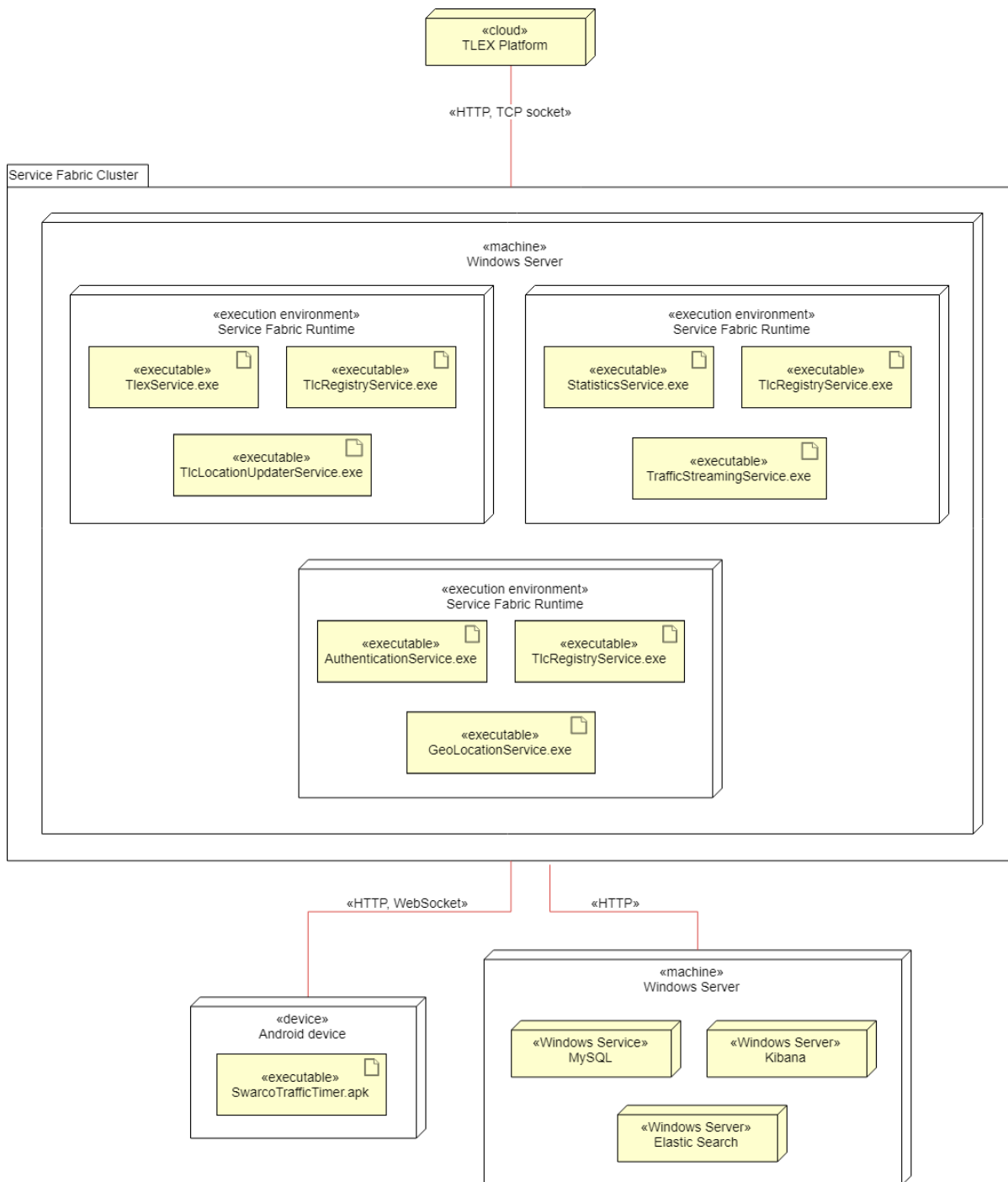


Figure 7.8: The nodes and artifacts involved in a full system deployment

Figure 7.8 shows the full system deployment, including the external TLEX platform. The Service Fabric cluster contains at a minimum three Service Fabric nodes setup on Windows Server machine. The cluster nodes are contacted through HTTP and through WS. The Cluster nodes connects to the external TLEX platform through HTTP and TCP sockets. The cluster nodes also must connect to the Windows server hosting MySQL, Kibana, and Elasticsearch.

The client application, installed on an Android device, contact the server through the reverse proxy of the internet-facing services.

The cluster nodes are hosted within a virtual network, preferably within a Windows Domain network. This allows the nodes to securely communicate with each other. However, the internet-facing services will still need to be reachable through their reverse proxies. The reverse proxy port will be defined in the cluster manifest when setting up the cluster nodes.

The project contains four different publish profiles. Three for development publishing and one for publishing to the production environment. Two of the development profiles are for publishing to a locally hosted development cluster, while the third is for publishing to a remote development server. The publishing profile to the production environment is only recommended to use in release mode, as the compiler will cut out some debugging tools, making the system a bit quicker. The debug mode is acceptable for the development server.

When a new version of a service is ready to deployed, it is possible to do a rolling upgrade. A rolling upgrade will deploy the new version of a service alongside the old service, ensuring that the system have minimal downtime [55]. As an upgrade is in progress, the cluster can be accessed to keep track of upgrade progress and cluster health.

Summary: *This chapter has described the relevant technology choices for the implementation and described the impact of these on the server solution with a component diagram. The user story implementations have been described as well as the authentication and authorization implementation of the system. Finally, the deployment diagram of the system was presented.*

8 : Experimental Validation

This chapter will describe the experiments conducted in order to validate the non-functional requirements of the system. The experiments includes validation of: scalability, latency, performance, fault tolerance, and usability. The chapter will also describe how a field test was performed.

Contents

8.1	Test Setup	53
8.2	Scalability Experiment	54
8.3	Latency Experiment	56
8.4	Performance Validation	57
8.4.1	Sub-experiment 1: From TLEX to Clients	57
8.4.2	Sub-experiment 2: From Clients to TLEX	58
8.5	Chaos Experiment	59
8.6	Field Test	61
8.7	Usability Validation	61

8.1 Test Setup

In order to validate the system, there is a need to setup a test environment. This environment consists of three TLCs, and three Virtual Machines (VMs) hosted on Azure. The VMs each have one virtual Intel Xeon CPU core, 3.5 GB of memory, and uses SSDs for storage.

Furthermore, some tests are conducted locally on a desktop PC, which has an Intel I7-7700k CPU and 32 GB of memory.



Figure 8.1: The three intersections that are hooked up to the TLEX Platform

Figure 8.1 shows the three intersections hooked up to the TLEX platform. It shows that the intersections are on the same road, *Toldbodgade*, at Odense Harbour.

8.2 Scalability Experiment

Description: This experiment tries to emulate what will happen if the system is brought under a heavy load.

Objective: The goal of the experiment is to see if the system is able to scale out, by adding new service instances to the system at high demands; if the system is able to scale in, by removing service instances at low demands; and to see if the system is able to scale across a distributed set of nodes.

Methodology: This test will be conducted by logging the metrics of the service instances, while running a stress test, which is designed to emulate a large user demand.

Results The outcome of the experiment can be seen in the figures 8.2, 8.3 and 8.4, which show the metrics of the service instances in the y-axis over time in the x-axis.

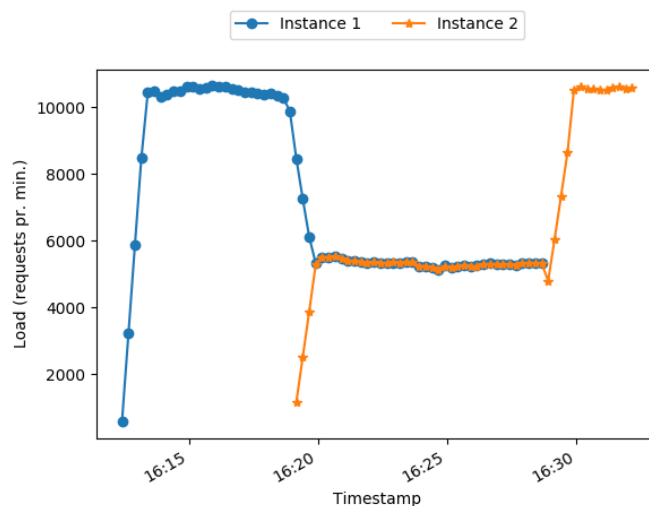


Figure 8.2: Graph of system metrics from Distributed GeolocationService instances

Figure 8.2 validates the systems ability to both scale out and in. This experiment was run on the GeolocationService, which has the metric *requests pr minute*. The service the been configured to scale out when the average metric across all instances exceeds 10000, and to scale in when it falls below 6000.

The graph in Figure 8.2 clearly shows the service exceeding the upper threshold, hereafter it scales out, by adding another instance. The system then falls below the lower threshold and scales in, by removing one of the instances.

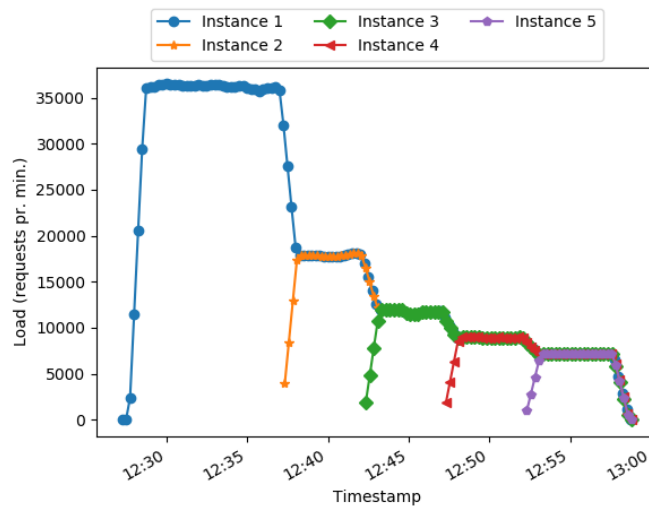


Figure 8.3: Graph of system metrics from local GeolocationService instances

Figure 8.3 shows the diminishing gain from scaling a service out. The impact from adding a new service is decreased each time a new service is added. The first time it has a huge impact on the load, but the impact keeps growing smaller as more instances are added.

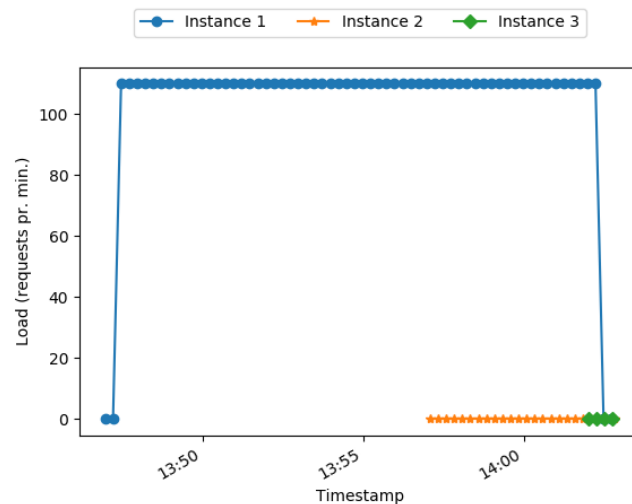


Figure 8.4: Graph of system metrics from local TrafficStreaming instances

Figure 8.4 shows the metrics for the TrafficStreamingService. It shows that the service has some balancing issues, as the load is constant, even though new service instances are added to the system. This issue is due to the statefulness of the WS connection in the service, as it does not automatically balance it self, as opposed to the REST endpoint of the GeoLocationService.

8.3 Latency Experiment

Description: This experiment should validate that the system is able to deliver the SPaT messages to the mobile application within the specified Time To Live (TTL) for that message. As seen in appendix C SPaT messages are invalid after a period of three seconds.

Objective: The goal of this experiment is to see if the system is able to deliver SPaT messages to a client before the TTL time has passed.

Methodology: A client will establish a connection to the TrafficStreamingService and subscribe to an intersection.

The client will then log the *TLC time delta* from the origin time¹, which is the time the SPaT was generated, until received by the client. This time delta will show how much time it takes for a message to be generated by a TLC till it is received by a client.

The client will also log the *System time delta* since the message was originally received by the TlexService² until received by the client. This time delta will show how much of the TLC time delta is time spent in the server solution.

This experiment will then run for a long period of time, in order to ensure that the data is more accurate.

Results: The outcome of this experiment can be seen in Figure 8.5, which shows a boxplot of the two time deltas.

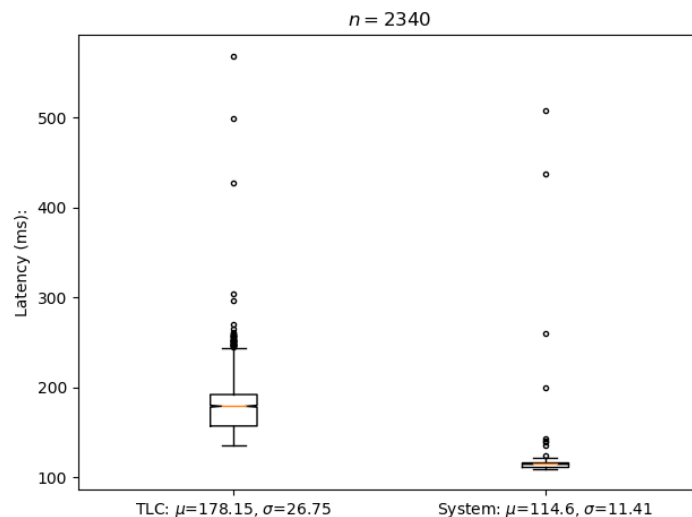


Figure 8.5: Boxplot showing the distribution of the gathered latency data

Figure 8.5 shows that the messages all of the spat messages are well below the TTL time. It also shows that the System time deltas are fairly consistent with a standard deviation of only $11.41ms$, where as the TLC time deltas varies more, with a standard deviation of $26.75ms$.

¹this is part of the SPaT message

²this information is passed on along with the SPaT

The average time of System time deltas seems to be a large part of the TLC time delta, but it is still a long way from the three seconds (3000 ms) defined in the TLEX TTL requirements.

8.4 Performance Validation

This experiment aims to validate the server solution's ability to handle messages. This ability is compared to the TLEX performance requirements. As seen in appendix C, the server solution should be able to handle approximately 132 thousand messages per second. In this situation, the server solution is connected to 1300 TLCs, all running at maximum capacity.

The experiment is divided into two sub-experiments. The first sub-experiment validates the ability to receive SPaT messages from the TLEX and send it to the clients. The second sub-experiment validates the ability to receive CAM from clients and send it to the TLEX.

8.4.1 Sub-experiment 1: From TLEX to Clients

Description: This experiment tries to validate the server solution's ability to forward messages to the client, from the TLEX.

This experiment is conducted locally, on a single machine, which runs both the client and server solution on five nodes. The server solution will not be connected to the TLEX platform, but the messages will be simulated instead.

Objective: The objective of the experiment is to verify that the server solution is able to handle SPaT load as required by the TLEX documentation, which is 1300 SPaT pr. second.

Methodology: A simulation task will be run on the `TlexService` that will continuously send mock SPaT messages to any client that is subscribing for data. Meanwhile a stress testing application will subscribe to data from 20 clients. The experiment will be conducted over five minutes.

The server solution is, in this experiment, configured to log the amount of messages that are sent from both the `TlexService`, and the `TrafficStreamingService`

Results: The result of the experiment is shown in the Figure 8.6 and Figure 8.7.

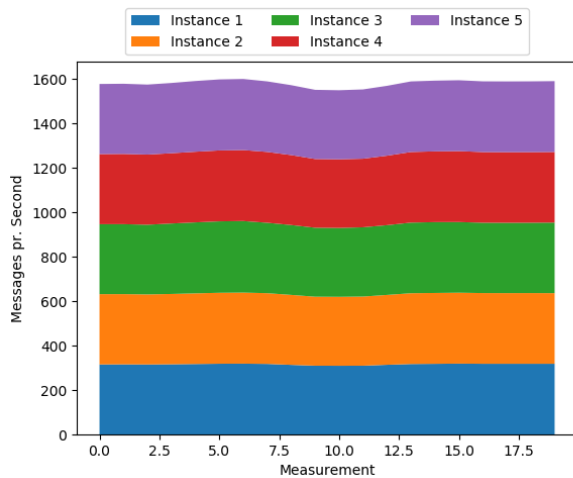


Figure 8.6: Messages sent in TlexService

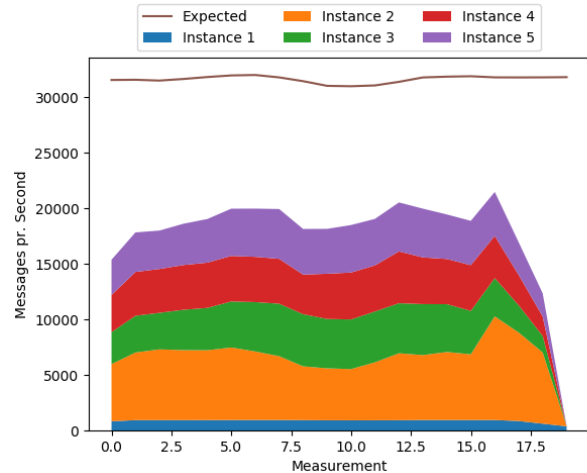


Figure 8.7: Messages sent in TrafficStreamingService

Figure 8.8 shows the amount of messages sent from the `TlexService`. The total number is somewhat stable and averages at 1611 messages pr. second.

Figure 8.7 shows the amount of messages sent from the `TrafficStreamingService` and a calculated expected value. The expected value is calculating the number of messages sent by the `TlexService` multiplied by the number of clients.

The results show that the `TrafficStreamingService` are sending fewer messages than expected.

During the experiment the `TrafficStreamingService` kept consuming an increasing amount of memory. It was discovered that it was the `SignalR` buffer that kept growing, meaning that it was not able to keep up with the high demand.

8.4.2 Sub-experiment 2: From Clients to TLEX

Description: This experiment is conducted on a local network with a setup of multiple machines.

The server is hosted on one physical machine which is the one described at the top of the chapter. This machine hosts four VM running Windows Server 2019 Standard. One of which is used as a Domain Controller using Active Directory to host a domain network. This network is used to connect the three other VMs, which each will host a Service Fabric Node. In order to evenly split the CPU processing power amongst the nodes, each VM was limited to using only one CPU core using Hyper-V.

The VMs will be reached by two different laptops acting as clients. The laptops are connected through WLAN while the server will be hosted on LAN.

Objective: The objective of the experiment is to verify that the server solution is able to handle CAM load as required by the TLEX documentation, which is 130 thousand CAM pr. second.

Methodology: The server solution is setup in a simulation mode that will ensure no connection with the TLEX platform will be made. Instead the number of messages received will be logged as load metrics, and sent to Kibana for further investigation.

All clients must establish 50 connections to the server and send as many messages as the CPU or network allows.

To test the scalability of the performance the experiment will initially be run on three nodes. After running the first test one of the nodes will be taken offline and the experiment will be run again.

Results: The results show that when running the server solution with three nodes the server is able to receive approximately 95 thousand messages per second. However, when running the server on two nodes the server is able to receive almost 110 thousand message per second. These results are depicted in Figure 8.8 and Figure 8.9.

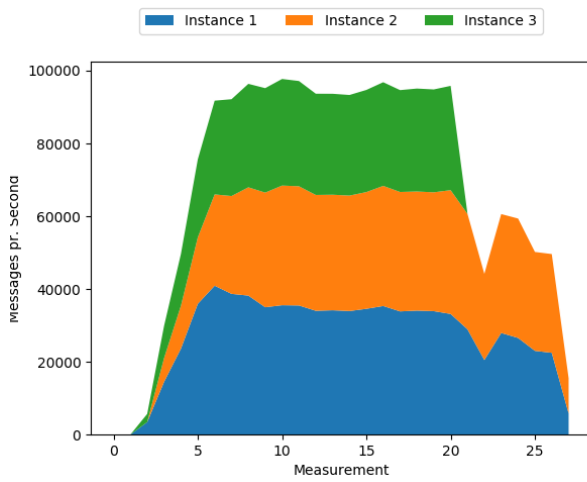


Figure 8.8: Three Node Configuration

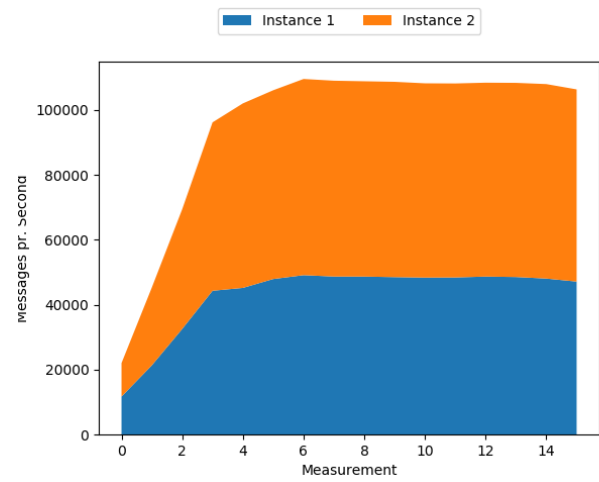


Figure 8.9: Two Node Configuration

The results show that the load is split almost evenly amongst all nodes. The results show that that fewer nodes provides better performance, which seems counter intuitive.

The drop in messages for instance 3 in Figure 8.8, is caused by a deliberate shutdown of that node's host VM. This was done as the experiment initially was meant to run over a period of time, while the number of nodes decreased. But the developed experiment tools, was not able to handle the decreasing number of endpoints, and the test had to be restarted, with only two nodes. It was not possible to run the experiment in this configuration with only one node.

8.5 Chaos Experiment

Description: Naturally in a system's time of life the system will experience faults. This is an unavoidable fact whether it be networking issues, power outages or other issues [56]. As a way of testing these sorts of faults the chaos testing approach is useful. The idea is to induce graceful and ungraceful faults while the system is running.

For this experiment the system will be deployed on a local development environment. The development environment is configured with 5 nodes. MySQL, Elastic Search and Kibana is hosted on this local environment as well. Chaos events will not occur on these three external services.

The nodes and the services deployed to the nodes can be interacted with through an HTTP client. Thus, enabling the functionality of gracefully and ungracefully shutting down services and emulating a node crash - *inducing chaos*.

Objective: The main purpose of the experiment is to see how well the system performs while a series of chaos events are induced on the system.

Methodology: Twenty chaos events will be induced one at a time. The health of the cluster will be validated after each chaos event which lasts up to four minutes. The chaos event will be chosen at random from a series of chaos events. While a chaos event is induced one workload will be performed at the same time.

A number of different graceful and ungraceful chaos events can occur.

- **Restart node:** Ungraceful shutdown and restart of a random node.
- **Remove replica:** Graceful removal of a random service instance.
- **Restart replica:** Graceful restart of a random service instance.
- **Move primary:** Graceful move of a random stateful service instance to another random node.

A number of workloads can be run while a chaos event occurs

- **Traffic streaming:** Opens a WebSocket connection and subscribes for data updates.
- **Several streaming clients:** Continuously keep 10 WebSockets connections each living a random duration between three and five seconds.
- **HTTP Geolocation:** Queries the 'nearby' endpoint on the Geolocation service.
- **HTTP Authentication basic login:** Queries the 'basic authentication' endpoint on the Authentication service.
- **HTTP Statistics intersection pass:** Queries the 'intersection pass' endpoint on the Statistics service.
- **HTTP Traffic Streaming endpoint:** Queries the 'endpoint' endpoint on the Traffic Streaming service.

All HTTP workloads will query their endpoint two times per seconds with required parameters. The Geolocation and Traffic Streaming services will receive GET requests while the Authentication and Statistics services will receive POST requests.

Results: No cluster health validation error were discovered. This means that cluster was able to recover from all 20 chaos event within four minutes.

A few bugs were discovered through the chaos events.

- A Dictionary in the `TlcLocationUpdaterService` received a 'collection modified while iterating' exception. Recommended measure would be to convert the dictionary to a `ConcurrentDictionary` as is it used concurrently.
- The `TlcLocationUpdaterService` and `TrafficStreamingServices` resolves an endpoint URI to the `TlcRegistryService` and caches the URI. However, as the `TlcRegistryService` primary replica moves to another node or is restarted the URI will change. This means the cached URI will be invalid. Recommended measure would be to invalidate the cached URI on connection errors.
- Moving the primary replica of the `TlcRegistryService` results in unexpected behavior and needs to be further investigated.
- An issue was discovered regarding subscription to data. Even though a client would be connected to a `TrafficStreamingService` and have an active subscription to a specific TLC identifier the client would not receive any data. The cause of this error needs to be investigated

further. However, it is suspected that the error occur when the `TlexService` instance, providing the data to the `TrafficStreamingService` instance, gets restarted. This should cause the `TrafficStreamingService` to reconnect and resubscribe on behalf of a client, but it apparently this does not behave as expected.

8.6 Field Test

Description: When developing a piece of software, it is often developed in an optimal environment. Taking the software out into the real-world brings new parameters into the equation. This means that the software should be validated by using it in the real-world.

The field test will validate the system's ability to correctly give speed recommendations and show the intersection state of incoming intersection as a driver drives towards them. It is important that the correct intersection approaches are visualized in the Android application and that they appear in a timely manner.

This field test also provides an opportunity to validate that the data is usable for GLOSA recommendations.

Objective: The goal is to validate the system in a real-life scenario and see if the system is fully functional under real-life conditions. The goal is further to validate that the data received is usable, for GLOSA recommendations.

Methodology: The system is tested by driving through the three intersections on Odense Harbour. The Android application must be installed on an Android device, placed in a convenient position, so that the device can be seen while driving. The application must be hooked up to a networked cluster using the Android device's cellular network. In this case the Android device is hooked up to the cluster as described in the test setup.

Results: This experiment was recorded and can be seen on the video on the attached Zip Archive³. The experiment showed that the system was able to determine the correct intersection as a driver drove towards the intersections.

A few observations were made during the experiment. One of which included the fact that one approach to an intersection was undetermined as the driver drove towards it. Another was the fact that the visualized areas were improperly scaled, making it difficult to adjust the speed of the vehicle to the signal area on the application. Finally, the video shows that the signal changes almost exactly the same time on the mobile device, as it does in the actual intersection signal.

8.7 Usability Validation

Description: This is a qualitative experiment, where the GLOSA application is presented to potential end users. The potential users are interviewed afterwards, using an unstructured interview approach.

Note: The experiment was not executed in a live environment with the system actually running. As the TLEX platform was inconveniently unavailable at the time of testing.

³Alternatively available at: <https://drive.google.com/open?id=1ibVeLRgI9lpVByG085DpANWF-eokhK15>

Objective: The goal of this experiment is to validate whether the GLOSA visualization makes sense to a potential end user.

Methodology: The potential users are presented with a video-demo of the GLOSA application. The potential users are asked questions, which will lead to answers regarding the visualization, if it makes sense, if it communicates information at a glance, or if they would use it themselves, etc.

The answers are then generalized, in order to evaluate the given feedback.

Results: Four people, all relatives to the project group, was interviewed for this experiment. All of the respondents are potential users, as they all are eligible and experienced drivers.

The general impression of the GLOSA visualization was good, and they all understood how to use it, and thought it would be possible to understand the speed recommendation at a glance.

One of the respondents answered that the visualization could be even more simple, suggesting to simplify or remove the vehicle from the center of the screen.

When shown a real life demo video of the application, some of the respondents raised concerns about the changing size of the visualized areas. It was not obvious that the areas changed size based on the vehicle speed.

Some of the respondents answered that they would use the system, if they lived in a place where they would pass multiple intersections daily. Otherwise, they would probably not use it that much.

Three of the respondents raised concerns, that the system would lead to more people speeding. This would be the effect of a green area just ahead of the driver, which would be possible to catch, if the driver accelerates above the speed limit.

Summary: *This chapter describes all the conducted experiments. The methodologies of the experiments are explained and the results are presented. The results act as a validation of the non-functional requirements regarding the system's scalability, latency, performance, fault tolerance, and usability.*

9 : Discussion

This chapter will discuss the experiments and results obtained from these. The chapter will try to discuss how well the experiments are executed, as well as how applicable the results are for making any real world conclusions.

Contents

9.1	The Produced Software Artifacts	63
9.2	How well does the solution solve the problem	63
9.2.1	Does the system introduce too much delay	64
9.2.2	How well does the solution scale	64
9.2.3	How well does the system recover from simulated chaos	64
9.2.4	How well does the solution perform	64
9.2.5	How well does the GLOSA application work 'at a glance'	65
9.2.6	How well does the solution provide statistics	65

9.1 The Produced Software Artifacts

The problem chapter declares three central objectives, which the system specification and implementation has been developed to achieve.

A Microservices solution was created using Service Fabric to support the requirements of scalability, resilience and responsiveness. This solution was able to successfully connect clients and TLCs by enabling real-time data exchange between the two. The client consists of a Unity application embedded in an Android application which is able to visualize a GLOSA in a real life scenario. As clients drive through intersection statistics are sent to an Elasticsearch database that is used by a Kibana front-end. This front-end can be access by administrative users and customers to visualize the statistics about intersection.

9.2 How well does the solution solve the problem

The Problem chapter defined three problem statements, which have been used to define requirements to the system. These problem statements, have been validated by conducting various experiments. This section will discuss the results and validity of these.

9.2.1 Does the system introduce too much delay

The produced solution was able to successfully utilize real-time data exchange. Its ability to provide current information was evaluated through a latency experiment. This revealed that the messages have a delay that is well below the requirements as specified by the TTL requirements from the TLEX specification appendix C. The field test also validated that the delay was unnoticeable.

9.2.2 How well does the solution scale

As the solution must be able to scale, its ability to do so has been validated experimentally. The experiments showed that the REST interfaces, behind a reverse proxy, was able to properly scale in and out. They were able to evenly distribute the load amongst all service instances. However, the SignalR WS interfaces did not fare as well. While they are properly scaled in and out based on load metrics they are unable to evenly distribute the load amongst all service instance. The reason that this happens is that the WS connections will remain even after a service has scaled out. This causes an imbalance between the newly instantiated service - that has no WS connections yet - and the initial service instance. The round-robin approach of the reverse proxy will continue to distribute requests evenly amongst the SignalR hubs potentially overloading the initial service instance while the newly created instance has plenty of available resources. The issue could be dealt with by introducing a software load balancing mechanism, that will distribute client requests based on available service instance capacity.

9.2.3 How well does the system recover from simulated chaos

The requirements defined that the system had to be fault tolerant and resilient. These attributes were validated by inducing chaos to the system. This experiment evaluated how the system would react to various chaos events, e.g. a service being taken down, or moved to another node. The experiment introduced 20 chaos events, from which the system recovered successfully from all 20, within four minutes. This experiment uncovered a few previously undiscovered bugs in the system, which was caused by certain edge conditions.

The workloads configured for this experiment, does not necessarily cover all the possible states of the system. Therefore it can not be determined with certainty that all faults can be discovered by the chaos experiments.

9.2.4 How well does the solution perform

The product requirement P02 specifies that the solution should exchange messages at nearly the same rate as described in the TLEX specification in appendix C. The solution's ability to do so was validated through two performance experiments. Difficulties in acquisition of necessary hardware meant that the performance experiment could only be performed on a single server machine. This meant that the performance experiment does not evaluate the solution's performance when scaling across multiple machines. The fact that the server solution was hosted on only one machine, means that any data obtained from these experiments are not very representative for a real world scenario. This will make it difficult to draw any valid conclusions as the solution is meant to be hosted on multiple machines at once.

The experiment was conducted in a set of two sub-experiments. The first experiment showed that the solution was able to send 25 thousand message per second through the wire to the client. This experiment cannot be translated into a real-world settings as the experiment was done using only twenty client connections which all received more than 1 thousand messages per second. In a real-world setting a client would exchange 2-3 messages per second. Thus, approximately 1000 clients would be needed

to reach the 25 thousand messages per second. However, the experiment does not show whether or not many clients would affect the system differently than many messages per second.

The second sub-experiment was performed under different circumstances than the first sub-experiment. This experiment showed that two nodes performed better than three nodes - which is contradictory to the idea of scalable microservices. However, the experiment was performed on a single server machine, hosting multiple virtual machines, which means that unexpected parameters could have affected the experiment. The results show in both the two and three node configuration almost the same number of messages was sent through the wire. This could indicate that the network was a bottleneck for the experiment. In order to get more meaningful results, a proper setup of at least five dedicated server machines, on a solid network connection, would be required.

9.2.5 How well does the GLOSA application work 'at a glance'

The GLOSA application needs to be placed somewhere visible for the driver, as to not cause the driver to deviate his eyes too much from the road. In the experiment, all of the users answered that they thought that it would be possible to understand the speed recommendations at a glance.

In order to ensure safety on the road the GLOSA application must not be too distracting to the driver. This have been a deciding factor in the visual design of the application UI. The design must be easily understandable, and clearly communicate the speed recommendation without the usage of numbers.

The usability validation revealed that resizing the signal areas, on the visualization, based on the speed of the vehicle lead to areas changing size erratically, which caused confusion to the respondents. It should be investigated whether this could be solved by sizing the signal areas using a fixed speed e.g. the speed limit.

The respondents for the usability validation was conveniently sampled among relatives of the project group. This brings in a major selection bias, as they could have potentially had a more positive answer than they otherwise would have had. The sample of respondents also only consists of four people, which is far from a representative number.

These two factors makes it hard to justify any real world conclusions made, based on the feedback from the respondents.

9.2.6 How well does the solution provide statistics

The solution provides various visualizations for investigating traffic conditions. These visualization can be presented based on live data or historical data. However, even with the number of different ways of visualizing the data, it is still difficult to investigate the relationship between multiple intersections. Say an administrative user wanted to view the flow of traffic at peak hours. This would require non-obvious methods of visualizing the data to determine the flow.

An evaluation of exactly how well these statistics perform has not been performed. It would provide better insight into the feature by doing so, however, no domain expert was consulted to make this evaluation.

10 : Conclusion

This report documents the development of a system for integrating vehicles with smart traffic solutions. The system consists of a microservice server implemented as a Service Fabric application. The server connects the vehicles to road infrastructure, through the TLEX platform. The server provides real-time data to a proof of concept GLOSA Android application, which helps drivers find the optimal speed for crossing intersections. As drivers pass through intersections, the GLOSA application sends CAM data to the intersection. The application also collects data about intersection passes, which is stored for further statistical analysis.

The microservice architecture has proven to be a very scalable solution for processing large amounts of information. The system is able to provide the necessary information well below the TTL time of the ITS messages.

The scalability experiments have shown that the Service Fabric services are able to automatically scale based on load. The distribution of the server, that comes with the ability to scale in and out, results in a server that is more resilient and available than a basic monolithic server. To accomplish this, Service Fabric will automatically heal services as faults occur. This was validated through chaos experiments that induced 20 faults on the server from which the system completely recovered within four minutes.

The performance experiments yielded inconclusive results. The results show that two nodes performs better than three nodes. This is counter intuitive and further experimentation would be required, in order to properly assess the system's potential performance.

The GLOSA application was presented to some potential users, who all answered that it was easy to understand how the GLOSA visualization worked. The validation also indicated that the GLOSA speed recommendation is understandable at a glance.

The system makes it possible for any road user to communicate with road infrastructure, no matter what type of vehicle they are using. This in turn, gives an incentive to further develop road infrastructure to become smart infrastructure, as the technology becomes more generally available for road users.

11 : Future Work

This chapter will direct the future of the system, by introducing any ideas and technologies, which might be interesting and provide value to the system and its users. The topics in this chapter will only be described briefly, and not delve into great details or consider feasibility.

11.1 Find Host Application

From the perspective of Swarco, the features, such as sending CAM data to intersections is useful, but it would be even more useful if the majority of drivers were sending this data to the intersections.

Therefore it would possibly be a great idea, to seek out a partnership with an already established actor in the field of driving applications. These actors could be anyone from *Google Maps* to the Danish *Fartkontrol.nu*. And offer them the possibility of integrating with our server system, which could in turn provide real-time intersection data for their products.

11.2 Autonomous Cars

Autonomous cars could also be interesting to work with, making autonomous vehicles integrate with the server system. This will allow them access to data containing the specific time the intersection will change its state, and could help them implement a more fuel efficient way of driving using GLOSA.

11.3 Wider support of ITS messages

There are more ITS message types than just SPaT, MAP, and CAM. These could be interesting to work with, as it would possibly allow the system to cater to a wider range of users. One example of a ITS message, which was not implemented in the system, is the Decentralized Environmental Notification Message (DENM), which contains data about road conditions, such as if there has been a car accident, or if the road is slippery, etc. This could be interesting to support as well.

11.4 Digitization of Roadside Infrastructure

Another ITS message type, that the system could support is the Infrastructure to Vehicle Information (IVI) message type, which contains data about road signs, such as warning signs or speed limit signs. These could potentially be registered digitally in the system, in stead of the signs needing to be upgraded to an Internet of Things (IoT) sign, that sends data through the TLEX or similar systems.

These signs could be visually shown in the application, so the driver can see the speed limit, even though the sign is out of sight.

11.5 CAM Bicycle Warnings

An idea emerged during the development process, which could help increase the traffic safety for cyclists. Simply put, we would want to set up a service which looks at CAM messages from drivers in the same intersection, and warn car drivers about the presence of bicycles in the intersection. This could maybe help reduce the risk of *right turn accidents*, where large vehicles are unaware of bicycles.

11.6 Better integration with vehicles

At the moment, the CAM data from the application, is restricted to contain what is possible to sense with a standard mobile phone. If we were to interface with the vehicle, which could provide more specific data, such as the turn indicators or the throttle or brake pedal status, the data could be more valuable.

It might be possible to make an integration with the CAN bus of the vehicle, which could read diagnostic data from the vehicle. The CAN bus is an industry standard connector for vehicle diagnostic tools.

Another way of better integrating with vehicles, would be to support smart car systems such as Android Auto or Apple CarPlay. These solutions would help physically integrate the application into the vehicle, by utilizing the displays of the vehicle.

11.7 Better Intersection and Approach Detection

At the moment the intersection and approach of the intersection is based only on proximity, making way for some bad edge cases, where the application potentially will subscribe to the wrong intersection, or show data for the wrong approach. It would definitely be beneficial to investigate better ways of intersection and approach detection.

Bibliography

- [1] D. Eckhoff, B. Halmos, and R. German, "Potentials and limitations of Green Light Optimal Speed Advisory systems," in *2013 IEEE Vehicular Networking Conference*, Dec. 2013, pp. 103–110.
- [2] H. Suzuki and Y. Marumo, "A New Approach to Green Light Optimal Speed Advisory (GLOSA) Systems for High-Density Traffic Flow," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, Nov. 2018, pp. 362–367.
- [3] Gevas Software, "trafficpilot - RIDE THE GREEN WAVE," 2018. [Online]. Available: <https://trafficpilot.eu/>
- [4] S. Larsen, "Bike timer – Simon Hjortshøj Larsen," Mar. 2018. [Online]. Available: <http://simonhl.dk/bike-timer/>
- [5] M. Roland, "Cooperative Systems Partnerships | SWARCO," Mar. 2018. [Online]. Available: <https://www.swarco.com/stories/cooperative-systems-partnerships>
- [6] J. Mikkelsen, "Greenwave for Copenhagen / Denmark | SWARCO," Oct. 2015. [Online]. Available: <https://www.swarco.com/stories/greenwave-copenhagen-denmark>
- [7] Swarco, "SWARCO TECHNOLOGY APS | SWARCO," Feb. 2019. [Online]. Available: <https://www.swarco.com/companies/swarco-technology-aps>
- [8] —, "About Us | SWARCO," Feb. 2019. [Online]. Available: <https://www.swarco.com/about-us>
- [9] L. Li, W. Huang, and H. K. Lo, "Adaptive coordinated traffic control for stochastic demand," *Transportation Research Part C: Emerging Technologies*, vol. 88, pp. 31–51, Mar. 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0968090X1830007X>
- [10] X.-F. Xie, S. F. Smith, T.-W. Chen, and G. J. Barlow, "Real-time traffic control for sustainable urban living," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. Qingdao, China: IEEE, Oct. 2014, pp. 1863–1868. [Online]. Available: <http://ieeexplore.ieee.org/document/6957964/>
- [11] curiousread.com, "It's go time. How do traffic light sensors work?" 2008. [Online]. Available: <http://www.curiousread.com/2008/01/its-go-time-how-do-traffic-light.html>
- [12] S. Messelodi, C. M. Modena, and M. Zanin, "A computer vision system for the detection and classification of vehicles at urban road intersections," *Pattern Analysis and Applications*, vol. 8, no. 1, pp. 17–31, Sep. 2005. [Online]. Available: <https://doi.org/10.1007/s10044-004-0239-9>
- [13] Swarco, "Swarco Cloud," 2017. [Online]. Available: <http://www.swarco.dk/content/download/20451/467490/file/Produktblad-Swarco%20Cloud-DAN.pdf>
- [14] P. Potters and F. van der Valk, "TLEX (Traffic Light EXchange), making intelligent traffic light information relevant," Sep. 2018. [Online]. Available: https://cdn.instantmagazine.com/upload/16332/paper_monotch_tlex_its_copenhagen.b0aff8702e81.pdf
- [15] Talking Traffic, "Het partnership," 2018. [Online]. Available: <https://www.talking-traffic.com/en/talking-traffic/het-partnership>
- [16] —, "TLEX," 2018. [Online]. Available: <https://www.talking-traffic.com/en/talking-traffic/tlex>
- [17] Monotch, "Traffic Live Exchange (TLEX)," 2018. [Online]. Available: <https://monotch.com/en/platforms/tlex>
- [18] —, "iTS Transfer point (TLEX) specification v2.2," Sep. 2017.
- [19] IEEE, "IEEE 802.11p-2010 - IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments," Jun. 2010. [Online]. Available: https://standards.ieee.org/standard/802_11p-2010.html
- [20] GSMA, "Cellular Vehicle-to-Everything - Enabling Intelligent Transport," Jan. 2018. [Online]. Available: <https://www.gsma.com/iot/cellular-vehicle-everything-enabling-intelligent-transport/>
- [21] SAE, "Dedicated Short Range Communications (DSRC) Message Set Dictionary™," SAE International, Tech. Rep., Mar. 2019. [Online]. Available: https://www.sae.org/content/j2735_201603
- [22] ACEA, "Average Vehicle Age | ACEA - European Automobile Manufacturers' Association," 2018. [Online]. Available: <https://www.acea.be/statistics/tag/category/average-vehicle-age>
- [23] ETSI, "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service," Sep. 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_EN/302600_302699/30263702/01.03.01_30/en.30263702v010301v.pdf
- [24] —, "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service," Sep. 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.02.01_30/en.30263703v010201v.pdf
- [25] ITU, "Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)," Aug. 2015, iTU = International Telecommunication Union. [Online]. Available: <https://www.itu.int/rec/T-REC-X.691-201508-I/en>
- [26] C-ROADS, "GLOSA," 2018. [Online]. Available: <https://www.c-roads-germany.de/english/c-its-services/glosa/>

- [27] NordicWay2, “Nordicway2 - Signalized Intersections,” 2018. [Online]. Available: <https://www.nordicway.net/services/signalized-intersections>
- [28] Vejdirektoratet, “NordicWay2 - Nordicway - Vejdirektoratet.dk,” Jan. 2019. [Online]. Available: <http://vejdirektoratet.dk/EN/roadsector/Nordicway/Pages/Default.aspx>
- [29] NordicWay2, “Nordicway2,” 2018. [Online]. Available: <https://www.nordicway.net/>
- [30] —, “Previous projects,” 2018. [Online]. Available: <https://www.nordicway.net/previousprojects>
- [31] C-Roads, “Platform: C-Roads.” [Online]. Available: <https://www.c-roads.eu/platform.html>
- [32] N. Østergaard, “Hvert femte trafikstyrede lyssignal har fejl - bilister spilder tiden,” May 2012. [Online]. Available: <https://ing.dk/artikel/hvert-femte-trafikstyrede-lyssignal-har-fejl-bilister-spilder-tiden-129496>
- [33] Trunomi, “Key Changes with the General Data Protection Regulation – EUGDPR,” 2018. [Online]. Available: <https://eugdpr.org/the-regulation/>
- [34] Transport-, Bygnings- og Boligministeriet, “Bekendtgørelse af færdselsloven,” Nov. 2018. [Online]. Available: <https://www.retsinformation.dk/Forms/R0710.aspx?id=204976#id7dacb0fb-6364-4cc2-9cdc-80fd28edbf80>
- [35] V. Khan, “Difference between scaling horizontally and vertically,” May 2019, original-date: 2016-12-07T12:29:20Z. [Online]. Available: <https://github.com/vaquarkhan/vaquarkhan/wiki/Difference-between-scaling-horizontally-and-vertically>
- [36] asn, “Abstract Syntax Notation One,” Feb. 2019, page Version ID: 883042297. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Abstract_Syntax_Notation_One&oldid=883042297
- [37] M. Fowler and J. Lewis, “Microservices,” Mar. 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [38] C. Richardson, “Microservices Pattern: Microservice Architecture pattern,” 2018. [Online]. Available: <http://microservices.io/patterns/microservices.html>
- [39] “Azure Service Fabric Documentation - Tutorials, API Reference,” 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/>
- [40] Unity Technologies, “Graphics Rendering,” 2018. [Online]. Available: <https://unity3d.com/unity/features/graphics-rendering>
- [41] —, “Unity - Multiplatform,” 2019. [Online]. Available: <https://unity3d.com/unity/features/multiplatform>
- [42] —, “Unity,” 2019. [Online]. Available: <https://unity.com/frontpage>
- [43] MySQL, “MySQL,” 2018. [Online]. Available: <https://www.mysql.com/>
- [44] Elastic, “Open Source Search & Analytics · Elasticsearch | Elastic,” 2019. [Online]. Available: <https://www.elastic.co/>
- [45] Oracle, “MySql.Data 8.0.16,” Apr. 2019. [Online]. Available: <https://www.nuget.org/packages/MySql.Data/>
- [46] Elastic, “NEST 7.0.0-alpha2,” May 2019. [Online]. Available: <https://www.nuget.org/packages/NEST/>
- [47] M. v. Oudheusden, M. Laarman, and M. H. Grabe, “Serilog.Sinks.ElasticSearch 7.2.0-alpha0005,” Apr. 2019. [Online]. Available: <https://www.nuget.org/packages/Serilog.Sinks.ElasticSearch/>
- [48] Elastic, “Kibana: Explore, Visualize, Discover Data | Elastic,” 2018. [Online]. Available: <https://www.elastic.co/products/kibana>
- [49] “Azure Service Fabric reverse proxy,” Nov. 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reverseproxy>
- [50] R. Riley, S. Koon, and M. Rendle, “OWIN — Open Web Interface for .NET,” 2014. [Online]. Available: <http://owin.org/>
- [51] “Overview of ASP.NET Core MVC,” Jan. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview>
- [52] D. Hardt and M. Jones, “The OAuth 2.0 Authorization Framework: Bearer Token Usage,” Oct. 2012. [Online]. Available: <https://tools.ietf.org/html/rfc6750>
- [53] M. Jones, J. Bradley, and N. Sakimura, “RFC 7519 - JSON Web Token (JWT),” May 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7519>
- [54] “Introduction to Identity on ASP.NET Core,” Mar. 2019. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity>
- [55] “Service Fabric application upgrade | Microsoft Docs,” Feb. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-application-upgrade>
- [56] M. Korolov, “Lessons From This Year’s Data Center Outages: Focus On the Fundamentals,” Nov. 2018. [Online]. Available: <https://www.datacenterknowledge.com/uptime/lessons-years-data-center-outages-focus-fundamentals>
- [57] O. Vogel, Ed., *Software-Architektur: Grundlagen - Konzepte - Praxis*, 2nd ed. Heidelberg: Spektrum Akad. Verl, 2009, oCLC: 255278881.
- [58] P. Manickam, S. Sangeetha, and S. V. Subrahmanya, *Component-Oriented Development and Assembly: Paradigm, Principles, and Practice using Java*, 0th ed. Auerbach Publications, Dec. 2013. [Online]. Available: <https://www.taylorfrancis.com/books/9781466581005>
- [59] D. Swersky, “What is Docker, and why is it so popular?” Oct. 2018. [Online]. Available: <https://raygun.com/blog/what-is-docker/>
- [60] I. Eldridge and pwpadmin, “What Is Container Orchestration?” Jul. 2018. [Online]. Available: <https://blog.newrelic.com/engineering/container-orchestration-explained/>
- [61] HashiCorp, “Introduction,” 2019. [Online]. Available: <https://www.nomadproject.io/intro/index.html>
- [62] —, “Nomad,” Jun. 2019, original-date: 2015-06-01T10:21:00Z. [Online]. Available: <https://github.com/hashicorp/nomad>
- [63] Spotify, “Helios,” Jun. 2019, original-date: 2014-05-26T02:29:06Z. [Online]. Available: <https://github.com/spotify/helios>
- [64] “Overview of the Service Fabric Reliable Service programming model,” Mar. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-introduction>
- [65] “Introduction to Reliable Collections in Azure Service Fabric stateful services,” Jan. 2019. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections>
- [66] “Partitioning Service Fabric services,” Jun. 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-partitioning>

-
- [67] "Health monitoring in Service Fabric," Feb. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-health-introduction>
- [68] "Manage Azure Service Fabric app load using metrics," Aug. 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-metrics>
- [69] "Azure Service Fabric Auto Scaling Services and Containers," Apr. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-autoscaling>
- [70] arungupta, "REST vs WebSocket Comparison and Benchmarks," Feb. 2014. [Online]. Available: <http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/>

Appendices

Appendix A

Process Report

To support the development of the product as described in the product report a series of tools and a development process were taken into use. The particulars of these subject will be explained in this process report. All the development iterations will also be described.

A.1 The Scrum development process

The development of the product should be accomplished by use of the Scrum development process. This allows the product to be developed in a set of iterations, called sprints, where every sprint will yield a complete part of the software, one or more complete features, or some other tangible value. The backlog and project task board was kept in a project board on GitHub, which is available here: <https://github.com/sweat-tek/SwarcoTrafficTimer/projects/1>.

Several sprints were completed in the process of developing the project. Each of which ended in a sprint review meeting in order to evaluate the results and to organize the following sprint. Initial to the project, however, a couple of meetings were held in order to understand the scope of the problem domain and to formulate a problem statement. All of these sprint meetings and the initial meetings are documented in the following sections.

The Initial Meetings

Before the development of the product could begin it was necessary to figure out exactly how to begin the development. To do this a couple of meetings were held to discuss the master thesis proposal with a supervisor at SDU and the project manager and the director at Swarco Technology.

The first meeting, at May 15th 2018, concerned what the project was and what milestones we should plan for. After this meeting a basic idea of what the product should be was created. A few requirements were discussed and the 2nd initial meeting was arranged. This second meeting, at May 23. 2018, concerned the product in more detail. More specific requirements were discovered and defined. By the end of this meeting a project proposal for the master thesis was formulated and approved for further research and development.

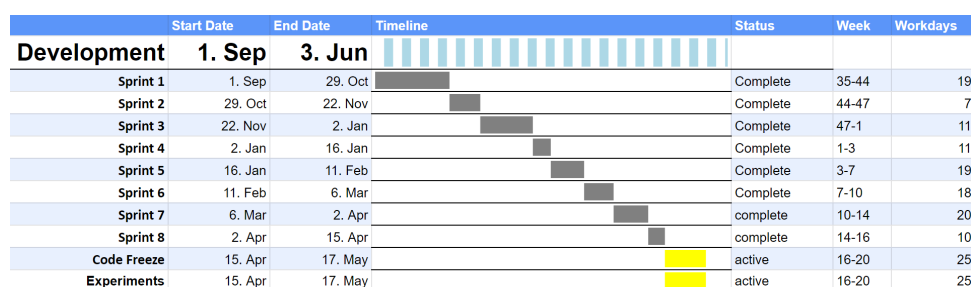


Figure A.1: The up-to-date Gantt chart of the project milestones.

The Figure A.1 shows the updated timetable for the project. However, this Gantt chart represents the organization of the work that was planned. This represents that sprints should be accomplished in 2-4 weeks as per the project agreement with the supervisor at SDU. The first sprint is longer because it represents the research and prototyping that was required to help the project kick-off.

Sprint 1

As this is the first sprint of the development a lot of work went into prototyping and determining the required architecture to support the requirements in the problem domain. Two meetings were held in this - quite long - iteration. This first of which was a supervisor meeting which was the first meeting after the summer break. It was concerning which technology should be used and also how Jenkins should be used alongside GitHub issues.

It was discussed whether or not an Android application or Xamarin application should be used to support the system. It was decided to go with an Android application as building a Xamarin application proved to be too complicated. And complexity in the front-end application is not what was deemed important for the project.

Instead more focus was put on the complexity in the server solution. This meeting discussed the possibility of using the Microservice architecture as a solution. An architectural prototype was discussed and the different hosting options were discussed. The resulting conclusion was to develop the solution using the Microservice architecture as it was deemed most fit to support the requirements in the problem domain.

By the end of the sprint - 9 weeks later - a sprint review meeting was held. At this meeting the requirements for using the OSS ASN.1 coder was established and a request for a student license was sent. No particular product artifact was created in this sprint - however, plans for the next sprint was set. By this point the framework choice was made and the Service Fabric solution by Microsoft was to be taken into use.

Sprint 2

At this time most initial requirements were in place and the development of the first real software artefacts was planned to be developed. The main goal was to make a connection to the TLEX platform from a server solution and provide the data from the platform to an Android device. The purpose of which was to establish the grounding foundation for the system.

A sprint meeting was held 3 weeks later with a demo of the results. The results being that the sprint was not accomplished as hoped. A couple of road blocks with the TLEX platform and not receiving data from the platform proved to be a small setback. The unfinished works was brought into the next sprint alongside a couple of new user stories.

At this point the OSS student license is still in the works. The missing license is an inconvenience but it was no setback, as a temporary DLL file allowed the ASN.1 messages from the TLEX platform to be decoded, as long as the date set on the machine running the program, was within the trial period.

Sprint 3 and 4

This sprint was concurrent with a large workload from other courses at the university. This meant that there was not as much time to work on this project as it was hoped. Some work was done in sprint 3, but only one sprint meeting was held. Thus it makes sense to describe these two sprints as one.

The output of sprint 3 and 4 was the translation of the ASN.1 generated model classes into the projects own model classes. The OSS student license is finally in place and it was possible to generate a set of model classes to support the SPAT and MAP messages from the TLEX platform. This made it possible to decode the messages, however, the generated model classes were unbelievably complicated and with an abundance of unnecessary code duplication. This led to the decision to translate the model classes into the projects own model classes.

With the model classes in place the system would now be able to subscribe for intersection data from the TLEX platform on behalf of clients. The client actually subscribing to the data would use it to visualize the intersection state on an actual map. At first this was a Google Maps map, but was later changed into an Open Streets Maps map.

By the end of the sprint a working demonstration of an Android application was shown to the supervisor at SDU. This alongside a competitor - Traffic Pilot - which would happen to bring new ideas to the visualization of the application. This meant that the current prototype on Android should be revisited.

Sprint 5

This sprint included the setup of Kibana as a logging solutions and to enable to future works on statistics and data gathering.

It also included more work on the server solution. It is wished that the Microservices architecture should be scalable and resilient. To accomplish the scalability of the system auto scaling and loads metric was investigated and implemented. This means that the system should be able to scale by itself dependant on load. To support the functionality of scalability a console application was made to stress test the solution.

Sprint 3 and 4 lead up to the fact that the Android client visualization should be revisited. To accomplish this a 3D engine was required and Unity3D was chosen.

Sprint 6

This sprint was very focused on the new goal of visualizing the speed recommendations. The GLOSA term was discovered and the development of a Unity3D application - embedded inside an Android application required a lot of work.

Besides working on the new visualizations a few bugs and refactoring were made. Most noteworthy of which was a bug with the traffic streaming session between clients and the server. It was seen the connection would be slow, if not faulty, when the WebSocket connection was established through the reverse proxy of the Service Fabric framework. Apparently the reverse proxy only recently started to support WebSocket - however, not very well. To fix this problem a REST API endpoint was created to help clients get direct access to a SignalR server and open a WebSocket connection without the reverse proxy.

By the end of sprint 6 a meeting was held. The status showed that the visualization still needed some work and work should be continued into sprint 7.

Sprint 7

Further work on the client application was done. Making is possible to specify certain settings, which can be used to generate a more meaningful CAM message. The CAM message generation was also accomplished in this sprint. The CAM messages are sent from the client to the server then sent to the TLEX platform which finally sent the message to the intersections.

Work on a statistics logging mechanisms was developed. This makes it possible to see statistics about intersection passes.

Finally a few refactorings on the Android client was made as well as a TLEX platform connection bug.

Sprint 8

This sprint concerned the development of an authentication service. This service should enable the authentication and authorization of all endpoints on all services. Even service to service connections should be authenticated.

Client visualization was improved with intersection warning signs. Signs like "no u-turn" and a bunch of other signs are supported. As long as it is passed through the MAP message the intersections push through to the client.

The client now also support the feature of sending a user report. At least the Android interface part has been implemented - server functionality is yet to be created to better support the feature.

Sprint 9 - Experiments

This sprint concerns the validation of non-functional requirements. Thus 4 full-time weeks was set aside to focus on this as well the product report.

A series of test cases must be developed in order to prove that the Microservices solutions fulfils the non-functional requirements set by Swarco Technology and Monotch (Monotch provided the TLEX platform documentation).

By the end of this sprint the project must come to a conclusion as the deadline for the project is the third of June 2019.

A.2 Process- and Development Tools

GitHub project board: To support the work towards building the project product the agile Scrum process was taken into use. To use this process a backlog is required to fill in all the tasks and user stories related to the project. These user stories and tasks are added continuously and are completed in sprints of 2-4 weeks of work. After every sprint a meeting is held to go through the completed work and plan what the next sprint should accomplish. GitHub project boards has been used to manage this.

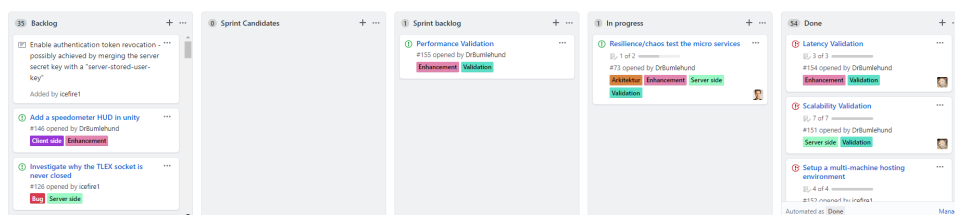


Figure A.2: The project board on GitHub

Figure A.2 shows the project board on GitHub. Five main columns are used to manage all the work. The first being the backlog which contains all the work that it wished to be completed. The second column is the sprint candidates, this column is used by the Scrum master and product owner to prepare candidates for the next sprint. This does not mean that developers should start working on the tasks instead it is only a way to highlight tasks for the next sprint meeting. At the sprint meeting work is chosen from the backlog and the sprint candidates and moved to the third column; the sprint backlog. As items enter this column developers must start working toward completing all the tasks. The Scrum master is responsible for assigning the work an ensuring the sprint successfully comes to a finish. As work begins on a user story or a task it is moved to the forth column; in progress. The item remains

here until it is completed and gone through a code review - if relevant. Lastly the item is moved into the fifth column; done. This means the item is completed and ready for the next sprint meeting. The item should be tagged as "sprint review" until the item has been evaluated and accepted by the customers.

Continuous integration with Jenkins: As tasks are completed pull request are made on GitHub. When a pull request is made at least one developer should review the changes suggested in the pull request. If a reviewer accepts the changes then the pull requests relies on GitHub checks in order to be ready for a merge. Two checks are done for every pull request one is a code quality check and the other is a build of the source code and a run of all unit tests. Jenkins is used to achieve the continuous integration in the second check.

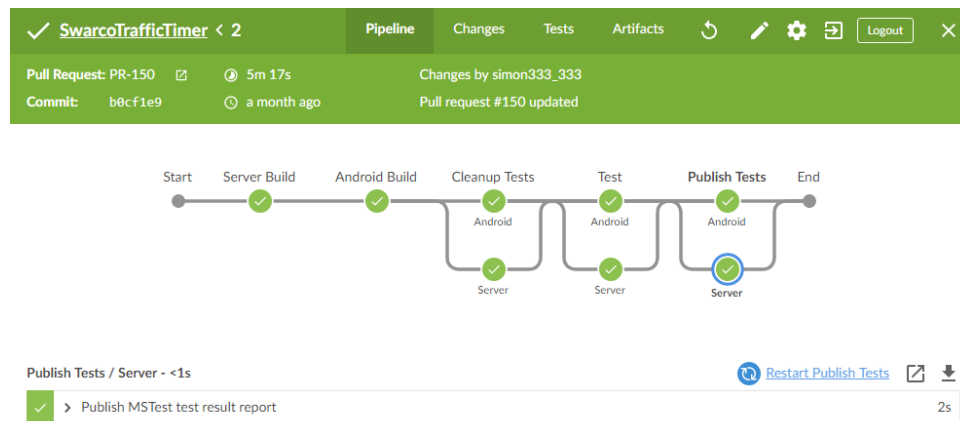


Figure A.3: Continuous integration using Jenkins

As a pull request is made a web request is made to a Jenkins server. Notifying the server that a certain pull request needs to be built and tested. Figure A.3 shows the build pipeline that any pull request must go through. The project consists of two parts that must be built and tested separately; the Service Fabric server and the Android client. The first step is to build the two solutions and clean up any previous test runs. When the build and cleanup finishes the server and client solutions will be tested concurrently and finally the tests will be published. Publishing the tests will make all the results of the tests available in the "Tests" tab inside Jenkins. If any tests fail - or any build for that sake - then the green colors will change to red. This will happen from time to time and is useful for catching annoying errors before the pull request is merged into a release branch on Git. If that happens then all new feature branches from the release branch will "inherit" the errors - which is not a favorable scenario as it is time consuming to fix the errors in a multitude of branches.

Better Code Hub: Better Code Hub is a tool for analysing the source code of a project. This is the second check that a pull request must go through before it can be merged into another branch. Better Code Hub will evaluate the quality of the code in the code base in order to provide maintainable code for future development.

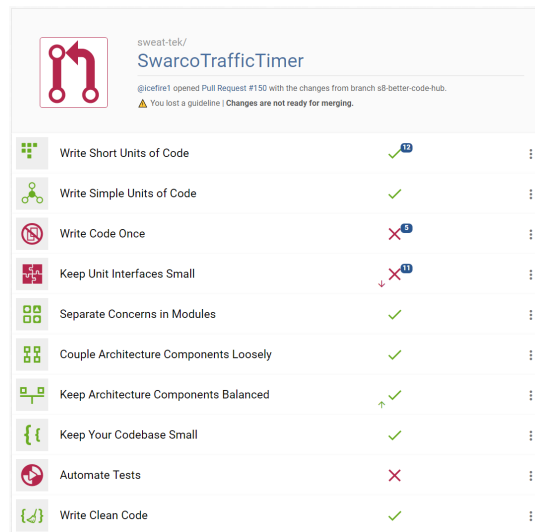


Figure A.4: Better Code Hub analysis of the server code base

Figure A.4 shows the analysis of code quality in the server solution. It is seen how the pull request resulted in losing a guideline. Thus resulting in three failing guidelines. Each of the guidelines are there to ensure the best possible code quality of a solution. The first of which ensures that shorts units of code are written in methods; the goal in which is to keep the methods below 15 lines of code. The second guideline evaluates the complexity of all units of code; the goal of which is to reduce the branch points in the code (fewer than 4 if-statements etc.). For every guideline there are set goals and if too much of the code violates the goals then the guideline will fail. Failing the guidelines leads to code that is more difficult to maintain and it is preferable to pass all guidelines.

Service Fabric Explorer: The server solution required the development of a Service Fabric cluster with several Service Fabric services. In order to deploy this a Service Fabric explorer was available on both the development environments and the production environments.

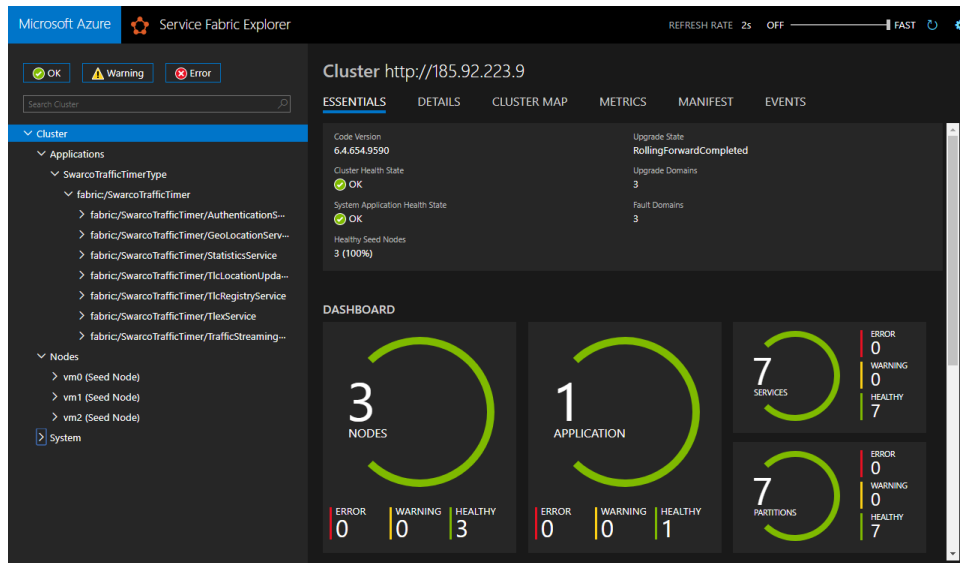


Figure A.5: Service Fabric cluster explorer

Figure A.5 shows the Service Fabric explorer on a production server. It is seen that the server is currently in a healthy state; all three nodes are up and running. Within the three nodes it is seen how one application is hosted on the cluster. This one application is then hosted seven service instances which all resides within their own partition - making seven partitions. Besides seeing the health of the cluster it is also possible to see the configuration of the cluster as well as the application. Amongst which the endpoints of all the services and the port that needs to be used to contact the cluster’s APIs and reverse proxies. Finally it is possible to see the load metrics of all the services - this helps when working with auto-scaling policies as they depend on load metrics.

Elastic stack: The Elastic stack contains, amongst other things, the Elasticsearch NoSQL database as well as a front-end for accessing the data - called Kibana. All logs will be sent from the server and client (through the server) to the Elasticsearch database. Kibana will present the log messages live from all the service instances on all the nodes. However, not only logs will be sent to the Elastic database. Statistics and loads metrics are sent to the database as well. This data can also be seen live in Kibana. Figure A.6 shows the logging dashboard which presents a view of statistics logs and server logs.

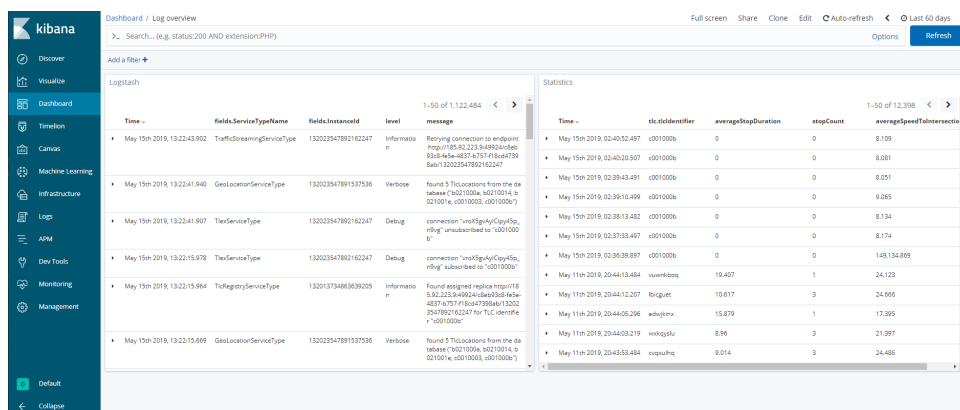


Figure A.6: Kibana front-end that is used to access Elasticsearch

Visual Studio and Visual Studio Code: The server is developed with C# and to support this development the Visual Studio editor is the preferred editor and is the editor that is mainly used through the project. Some testing applications have been made in JavaScript and occasionally other files than .cs files are opened. These files are typically edited in Visual Studio code. As this editor has an abundance of plugins that supports whatever file format you can find. It is has quick loading time making it a fast solution for quickly looking through a file. However, it is not as advanced as Visual Studio, which allows for much more advanced way to write and refactor code.

Android Studio: The client is developed in Android Studio and built as an Android application. Android Studio is the most obvious IDE to use for Android development - no other IDE really compares. Xamarin was considered for cross-platform support, but even though Xamarin provides native apps for Android the support to do so is not as good as Android Studio.

Unity3d: The Android application mainly consists of a Unity3d application. Unity3d is used to create the 3d visualization that is used to present the speed recommendation to the drivers.

Gradle: Gradle was used as the dependency manager for the Android client project.

NuGet: NuGet was used as the dependency manager for the C# server project.

Git: As a versioning control tool Git was used. Git was the obvious choice for the development team as the development had prior experience with using Git. The main online repository provider also use Git as a versioning control system, such as GitHub and Bitbucket.

Google Docs: Google Docs was used to create timetables for the project as well as for writing the project report. Google Docs is easily to collaborate inside and enables the use of review through comments.

Microsoft Word: As the collaboration in report writing was deemed "not as essential" any longer the writing was moved to Microsoft Word. Word has far superior grammar rules and spelling checks for both Danish and English. It also has support for online collaboration, however, not as easily as Google Docs as it often produces conflicts when multiple users are editing the same document.

Overleaf: To make the report look good and feel more "standardized" LaTeX was used to produce the final report product. Overleaf was used as a collaborative LaTeX editor and compilation engine.

Summary: *The process of developing the product in the product report has been accounted for. The agile scrum process method was used to develop the product in several sprints - a timetable was presented to show the organization of these sprints. The tools to enable the scrum process has been covered as well as the other necessary development tools.*

Appendix B

User Stories

ID	Name	User story	Status
US01	Subscribe to relevant TLCs and select upcoming intersections	As a driver, I want to subscribe to updates from relevant TLCs so that I can get speed recommendations and countdowns on intersection I'm passing through	Done
US02	Generate and forwards CAM message from the Android client	As a driver, I want to send data to the intersection I am driving towards in order to ensure that the intersection knows about my presence and can adapt its state accordingly	Done
US03	Create a statistics logging mechanism	As a customer, I want to view statistics about traffic flow so that I will know if an intersection must be optimized for better traffic flow	Done
US04	Create a GeoLocation service	As a driver, I want to know which TLCs are in the area, so that I have information about relevant TLCs	Done
US05	Integrate with Open Street Maps on Android	As a driver, I want to see the intersections I am driving towards on a geographical map, so that I will know when I will have to drive through an intersection	Done
US06	Create a TLEX service	As a driver, I want to subscribe to certain TLCs so that I can get live traffic data	Done
US07	Calculate and select approaches for the lanes in map file	As a driver, I want the system to divide intersection lanes into approaches, so that I am not shown information from lanes which are not on my road	Done
US08	Show speed recommendations	As a driver, I want to cross intersections in a 'green wave' by being given a speed recommendation so that I can get through traffic as fast as possible	Done
US09	Select the appropriate lane based on an approach	As a driver, I want to be assigned a specific lane that fits my path history so that I can get countdown and speed suggestions based on a specific signal group	Done
US10	Create a UI abstraction of the intersection	As a driver, I want to see the speed recommendation of all signal groups in a simple graphical interface, so that I can easily decide my speed	Done
US11	Create a Traffic Streaming service	As a driver, I want to subscribe to certain TLCs so that I can get live traffic data	Done
US12	Create a simulation mode for the TLEX service	As an admin, I want to simulate message from the TLEX Platform, to make sure the system operates optimally before the system is deployed to production	Done

Appendix B. User Stories

ID	Name	User story	Status
US13	Specify Vehicle	As a driver, I want to specify what vehicle I am driving, so that my CAM data is more precise and the TLC is able to better act on my presence	Done
US14	Create an interface for sending user reports	As a driver, I want to report events in traffic, so that I can help optimize the traffic flow and help ensure traffic safety	Done
US15	Create an authentication server	As a driver, I want to sign in and create a user account so that my data can be persisted and is safe	Done
US16	Show warning traffic signs on the intersection lanes	As a driver, I want to be presented with relevant signs from the intersection so that I am sure not to overlook them	Done
US17	Visualization of lanes for cyclists has to consider car lanes shared with cyclists	As a driver, I want to be presented with relevant signs from the intersection so that I am sure not to overlook them	Done
US18	Change various edge cases of how the lanes are selected for visualization	As a driver, I want to be able to see all the lanes of an approach, so that I am able to better project the visualization onto the real world	Done
US19	Refactor all relevant services to use config files based on release mode	As a customer, I want to ensure that the debug configuration is never mixed with the production configuration, so that the system always behaves as expected	Done
US20	Separate Android logic into Gradle project	As a customer, I want the business logic of the client to be as re-useable as possible, so that multiple visualization implementations can be developed from the same business logic	Done
US21	Implement TLC Registry service	As a customer, I want the system to split the intersection subscriptions amongst the Tlex services, so that no subscription is duplicated across Tlex services	Done
US22	Implement autoscaling and load metrics	As a customer, I want the system to scale dependant on usage, so that I don't waste computing resources	Done
US23	Setup Kibana/Elastic as a logging system	As an admin, I want to be able to aggregate log files across replicas, so that it will be faster to look go through the logs	Done

Appendix C

TLEX Performance Requirements

This appendix contains performance requirements to the TLEX platform.

Time to Live Requirements

The system should discard MAP message after 300 seconds
The system should discard SPaT message after 3 seconds
The system should discard DEN message after 60 seconds
The system should discard SSM message after 10 seconds
The system should discard CAM message after 5 seconds
The system should discard SRM message after 10 seconds

Throughput Requirements

The system should be able to exchange 1300 MAP messages pr. day
The system should be able to exchange 1300 SPaT messages pr. second
The system should be able to exchange 1300 DEN messages pr. hour
The system should be able to exchange 1300 SSM messages pr. minute
The system should be able to exchange 130000 CAM messages pr. second
The system should be able to exchange 1300 SRM messages pr. minute

Appendix D

Software Architecture

The software architecture refers to how a system is structured at a high level. Each architecture type has its own pros and cons, which will make them suitable for different system types. The following sections describes three different software architectures which has been considered for the server solution in the project.

D.1 The Client/Server Architecture

This architecture is the traditional approach to distributed software architecture. The system is divided into two parts the client and server. The users of the system run the client locally on their machine, this client can then interact with the server to e.g. synchronize data across clients [57].

This architecture of the server is simple, and often monolithic. This means that the server runs in a single process. This causes scalability to become an issue, because it is prone to bottlenecks, where the system is becoming ineffective due to a bad scaling part of the system. The only way of scaling this type of system is to deploy the whole system on a new machine [37].

D.2 The Component-based Architecture

This architecture tries to separate the functionality of monolithic applications into independent components, which work together in a single process. It does this using various service contracts which the components fulfill [58].

The separation of functionality has the effect of increasing the maintainability of the system, as each component must be independent of other components. This makes it suitable for larger systems, with multiple developers.

Because of the independence of components, it is possible to reuse entire components, this is very suitable for products which are being customized for different customers, or target groups [58]. Component based architecture also allows for deploying functionality at runtime, so that the system does not have to be taken down in order to run new code [58].

D.3 The Microservice Architecture

This architecture has, like the component-based architecture, separated the system into smaller functionality units called microservices [37, 38]. The difference compared to the component-based architecture is that these units each runs in their own processes. These microservices work in unison to build up the entire system, and communicate with each other, often using network calls [37].

The separation into smaller processes and network communication allows the system to be deployed to multiple computers at once, but still be part of the same system. This is important for scaling, as it allows for adding more resources to the system, by deploying to a new computer. This also allows scaling of only the necessary microservices, instead of the whole system as is the case of the monolithic architecture [38].

The ability to run multiple instances of the same microservice makes it resilient to failures, as it is possible to run a backup instance, which will take over in the event of a failure. The microservice architecture has the property of isolating faults in the system, to mostly only affect the service where the fault is

present, which means that only one part of the system is affected. An example of a fault could be a memory leak, this would only affect the service with the leak, and not spread to other services [38].

The microservice architecture is not a perfect architecture, it adds a lot of additional complexity to the system, as everything must be distributed applications. The deployment of the system also increases in complexity, as it is not one system which needs deployment, but multiple, with yet more instances of these. Microservices are also prone to consume more system resources, as there is a need for multiple instances of the runtime environment to run the system [38].

Appendix E

Microservice Frameworks Pros and Cons

This chapter will investigate what container orchestration is and what solutions that support it. All of the discovered container orchestrators will be evaluated for pros and cons

E.1 What is Container Orchestration

In 2013 - 6 years prior to these works - docker was released. Since then docker and the concept of containerizing components has spread widely [59]. The microservices architecture and the concept of containerization goes hand-in-hand as both concept requires the concept of isolation. However, managing a microservice system of a high number of containers and application becomes difficult. Imagine managing the up-time of 400 services hosted on 1000 different containers hosted on, potentially, hundreds of networked machines [60].

Container orchestration is all about managing these kinds of complex scenarios. A set of tasks revolving around the concept of container orchestration is presented by Isaac Eldridge [60].

- Provisioning and deployment of containers
- Redundancy and availability of containers
- Scaling up or removing containers to spread application load evenly across host infrastructure
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Allocation of resources between containers
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it

Performing all of these tasks manually would require a tremendous amount of work for a large number of containers, as previously pointed out. So to automate these tasks a number of container orchestrators have been created. All of which have their own strengths and weaknesses.

E.1.1 What Container Orchestrators exists

This section will discuss a number of different container orchestrator solutions. Each of these solutions can be utilized to support a microservices architecture, but to very different extends. These differences will be discussed as well as their strengths and weaknesses. A complete technical insight into the specific implementation of all the solutions will not be provided, as this would require at least some experience working each of solutions - at the time of writing no working experience, with any of the solutions, have been acquired.

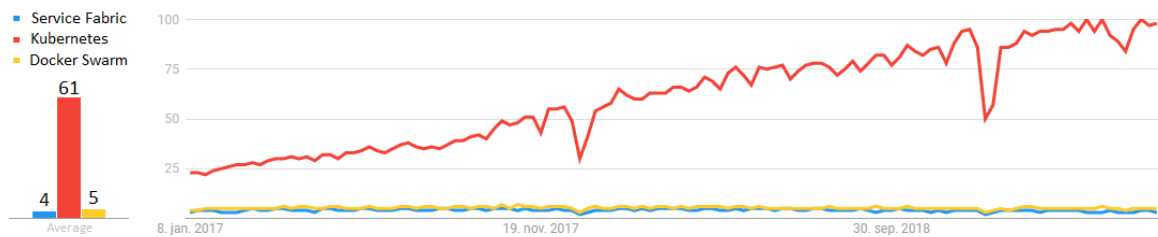


Figure E.1: Google Trends comparison of Service Fabric, Kubernetes, and Docker Swarm from the 8th of January 2017 to the 1st of June 2019

Figure E.1 shows the most trending keyword, according to Google, in the recent two and a half years. It is seen how Kubernetes takes a massive lead in popularity compared to Service Fabric and Docker Swarm. Other keywords have been omitted from the Google Trends comparison, as none of the other relevant keywords reached above one on the chart.

Service Fabric

Service Fabric differentiates itself a bit from the container orchestrator term. Instead, Microsoft promotes their product as being a distributed systems platform. It enables packaging, deploying, and managing scalable and reliable microservices as well as containers. That is; Service Fabric provides a framework specifically for building microservices while keeping containers a second-class citizen.

Service Fabric has been used internally by Microsoft to host a number of services on the Azure portal. This goes to prove that Service Fabric is battle tested as the Azure portal is responsible for hosting a great number of applications.

Service Fabric is built with the ASP.NET environment in mind. The framework easily integrates with all the features that ASP.NET Core MVC provides. This makes it incredible easy to work with if the developers already have experience working with the environment. However, the Service Fabric SDK is also ported fully to Java. This makes it possible to utilize all the features of Service Fabric within Java. Guest containers are also an option, for development teams who wishes to bring legacy application or any other Docker image onto the highly available and reliable container orchestrator.

Service Fabric is completely open source and free to deploy to any on-premise server as required. However, Microsoft also offers the option of handling the infrastructure setup through a Service Fabric application on the Azure portal. Helping developers avoid the mess of setting up the hosting environment themselves. This of course comes at a price.

The big downside to Service Fabric is that it is heavily dependant on Windows Server and PowerShell. This can make server hosting more expensive as Windows Server licenses do come at a relatively high cost. Another downside is that Kubernetes is more popular than Service Fabric.

Docker Swarm

After the release of Docker the Docker team has developed and released a container orchestrator called Docker Swarm. While the company actually fully embraces Kubernetes they still offer Docker swarm. It is slightly less complicated to work with compared to Kubernetes and is still relevant to discuss as a potential container orchestrator.

Docker Swarm is particularly easy to use if the development team already uses Docker containers for all their components. Reason being that Docker Swarm utilizes the same API and networking that is used by the Docker tools.

The Docker Swarm container orchestrator does not come with all the bells and whistle as, for example, Kubernetes or Service Fabric. However, it has a more simple setup and thus easier to use. Although Docker Swarm is open-source it is not nearly as popular as Kubernetes and therefore does not have the same community backing as Kubernetes.

Docker Swarm does however bring most of the essential features that a container orchestrator should have. Such as cluster management, automatic and manual scaling of services, service discovery, and rolling upgrades.

As Docker Swarm is a part of Docker the preferred deployment environment is Linux as Windows certainly is not optimized to run Docker containers. However, Docker Swarm is a free to use orchestrator and bring the necessary features a container orchestrator needs.

Kubernetes

Kubernetes is the most popular player in the field of container orchestration. This is clearly seen when looking at Google Trends, Figure E.1, from 2017 to 2019.

Kubernetes provides all the features of a container orchestrator. All of the tasks, mentioned by Isaac Eldridge, can be accomplished using Kubernetes.

Kubernetes is fully open-sourced and available for free. It can be hosted in mostly any environment, but it is preferred to used with Linux as Docker containers natively run on Linux. All the major cloud platforms such as Google, AWS, and Azure provides Kubernetes as a PaaS (Platform as a Service) which simplifies the deployment of a Kubernetes system by handling the infrastructure setup of required machines.

Nomad

Nomad [61] is, in itself, an orchestrator that enables easy deployment and manageability of any containerized or legacy applications. Nomad differentiates itself from other tools by providing simplicity and flexibility. Nomad is tightly integrated with three other tools; Terraform, Consul, and Vault. Each of these tools provide essential tooling for service discovery, health reported and recovery from faults etc.

According to Nomad they have proven themselves to support cluster scaling as high as 10 thousand nodes in a real-world production environment. The overhead of running Nomad is as low as running a single 75MB binary and is environmentally agnostic. This means that Nomad can be run on both Linux and Windows Server and is not tightly coupled with any cloud providers, like Azure, Google, AWS etc. Supposedly any type of binaries can be hosted through Nomad as the orchestrator is not coupled to any programming frameworks.

Nomad also looks to be a well established tool when looking at their fully open-source repository on GitHub [62]. With over 15 thousand commits, this free to use container orchestrator looks promising.

Helios

Helios [63] is one of the older container orchestrators. It was built before Kubernetes, but was deprecated since the release of Kubernetes. Thus, no new features or implementation is done on Helios. This means that going for this solution would be a poor choice as there will be no future perspective.

Summary: *The chapter introduced the concept of container orchestration and introduced a number of solutions within the concept of container orchestration. All of the container orchestration solutions was*

discussed in relation to each other and it is clearly seen how Docker alongside Kubernetes is widely spread. Service Fabric is deeply involved in the ASP.NET environment, but also provides more of a framework implementation than some of the other container orchestrators.

Appendix F

Service Fabric Introduction

In order to host a server solution using Service Fabric, a Service Fabric Application project must be created. This creates the minimum necessary project setup in order to host a Service Fabric application.

A regular project in C# is called a *solution*, and can consist of many sub-projects. These sub-projects are declared in a *.sln* file. A C# project contains a *.csproj* file, which declares the dependencies, as well as contents of the project.

Figure F.1 shows an overview of the file structure of a Service Fabric solution, containing the essential files necessary for hosting a Service Fabric cluster and a Service Fabric Service. It has essentially the same structure as a regular C# project.

The Service Fabric project, `ServiceFabricApplication1`, is the main project that must be built and published when deploying the solution. For this reason, this project must have a reference to all the services that should be deployed. This reference must be done both in the *.sfproj* file (like in a regular *.csproj* file) and in the *ApplicationManifest.xml* file. The *ApplicationManifest.xml* file declares all the available runtime parameters that are injected into the services loaded. It also declares which services should be deployed in the project.

The parameters are key/values pairs that can be injected into a Service Fabric service. In order to achieve this, the individual services must declare their required parameters in their Package-Root/Config/Settings.xml file. The concrete values of the parameters must be defined in another xml file e.g. *Cloud.xml*. This file is then used by a publish profile that deploys the Service Fabric to a defined destination cluster with the appropriate parameters.

This introduction just described the surface knowledge of how services and the service parameters are declared in a Service Fabric solution.

F.1 Service Types

There are two different service types in Service Fabric, namely the stateless and the stateful services.

Stateless services are services where no state is maintained within the service, it might store its state outside of the service, e.g. in a data store. This means that there is no requirements for synchronisation, replication and persistence of the state, making instances of the same service completely independent from each other [64].

Stateful services consists of Primary replicas (term for stateful instance) and secondary replicas. The primary replicas does all the heavy lifting, and are the *active* replicas, whereas the secondary replicas are merely there as a fail over in the event that the primary encounters an error [64]. Service Fabric provides a toolset called Reliable collections, which is a set of collection types, which handles all the synchronization of state across the primary and secondary services [65].

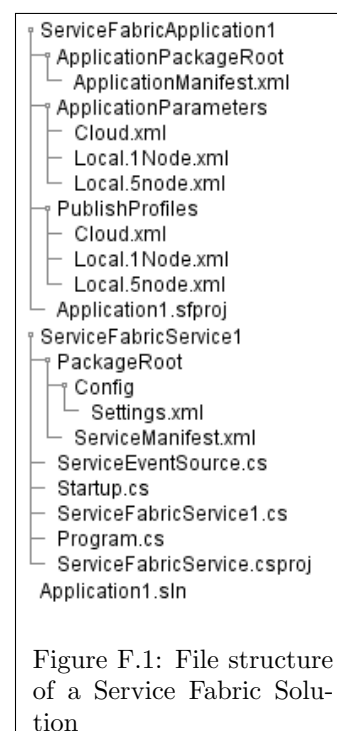


Figure F.1: File structure of a Service Fabric Solution

F.2 Partition Types

Partitions are useful when working with large data sets, as it makes it possible to allocate different sections of the data to different services. Service Fabric makes it possible to make three different partitioning schemes, the ranged integer-based partitioning scheme, the named partitioning scheme, and the singleton partitioning scheme [66].

Ranged Partitioning divides the data into different integer intervals, e.g. splitting the range from the minimum to the maximum values of int64 evenly into n intervals. These intervals are then accessed by providing a partition key, which accesses the partition covering the range, which the key falls into [66].

This can be used to partition data which has an identifier, like the TLC Identifier. In order to create a key from the identifier (which is a string), one would have to run a hash algorithm, like MD5, which can be used to get an integer digest value from virtually anything [66].

Named Partitions divides the data into different logical named groups, e.g. if the data contains a country name, the data could be partitioned into countries. In order to access the partition, one would need to provide a country name as a key. This partitioning is not always balanced, as some partitions may be over represented and others will be under represented [66].

Singleton Partitions can be used for services, which does not require partitioning. It simply defines that no partitioning is used [66].

F.3 Cluster health, metrics, and auto scaling policies

Service Fabric services report various parameters such as the health state and metrics to the cluster. These parameters can be used for monitoring e.g. resource usage.

Cluster Health Reports are reports about the health state of the service. This can be used for monitoring and diagnosis of any issues present in the system [67].

Metric Reports contains data about resources that can have a performance impact on the services. This is particularly useful, as it makes it possible to monitor the utilization of these resources, which can be used for optimization of this, making the services run more smoothly [68].

Scaling Policies makes it possible for the cluster to automatically scale the services based on metric usage [69]. This policy is declared in the *ApplicationManifest.xml* of the Service Fabric project, as shown in section 7.1.

```
1 <ServiceScalingPolicies>
2 <ScalingPolicy>
3 <AveragePartitionLoadScalingTrigger MetricName="RequestsPrMinute" LowerLoadThreshold="6000"
  ↳ UpperLoadThreshold="10000" ScaleIntervalInSeconds="300" />
4 <InstanceCountScalingMechanism MinInstanceCount="1" MaxInstanceCount="5" ScaleIncrement="1"
  ↳ />
5 </ScalingPolicy>
6 </ServiceScalingPolicies>
```

Code Snippet F.1: A Service Scaling Policy

Code Snippet F.1 shows a scaling policy, which scales the service based on the metric *RequestsPrMinute*, used in e.g. *GeoLocationService*. The policy declares a trigger, on line 3, which states that whenever the average metric value, across all service instances, is above 10000 it will scale out, and when it is below 6000 it will scale in, as well as how often this policy should be triggered. It also declares a scaling mechanism, on line 4, specifying the min and max amount of instances, and how many instances it should add to or remove from the cluster when triggered.

Service Fabric allows declaration of scaling policies for the *Named* and *Singleton* partitioning schemes, but not for the *Ranged* partitioning scheme, as it would simply require too much data to be redistributed among the services when the service scales, which is often done in the event of higher loads.

Appendix G

Prototypes

In order to investigate the requirements and required technology, a series of prototypes were developed. Two of which include a performance prototypes for evaluating the best fitting networking protocol and a sample to interact with the third party TLEX-cloud platform. A prototype for GLOSA visualization was also implemented.

G.1 The networking prototypes

In order to better understand the performance requirements of the system a couple of prototypes testing the performance of network communication protocols were made specifically, the HTTP protocol, and WS protocol.

The HTTP protocol is simple to use and almost any type of networked client will be supported. The HTTP protocol goes hand in hand with the REST and RPC architectures which are commonly used and provide well defined interfaces. In contrast the WS protocol is not widely known for a certain architectural design. However, the WS protocol is known for its strength in establishing a two-way connection between a client and a server. Whereas the only way of having a two-way connection with HTTP is the usage of server polling.

Two prototypes were made to evaluate the networking performance of each of the two protocols. Our tests showed that the WS connections would out-perform the HTTP connections as the number of messages increased. This is also supported by the benchmarks published by Arun Gupta [70].

As the two first user stories (see chapter 4) describes there is a requirement for two-way communication. This is where WS really shine as compared to HTTP connections. Reason being that HTTP connections are unidirectional which means that client would have to poll a server for updates.

If the system should be able to perform as well as the TLEX cloud platform documentation [18] demands (shown in appendix C). It would be wise to choose the WS communication protocol.

G.2 TLEX connection prototype

To better understand the third-party boundary, the TLEX cloud platform, a prototype was made to establish a connection to the platform and manage a session. This prototype also provided useful insights to the ASN.1 encoded messages, and the ASN.1 coder library chosen for the task of encoding and decoding messages received and sent to the platform.

G.3 Client Prototype

The front-end application which must provide all the functionality that is described in all the user stories (see appendix B). However, to initially get an impression about what Swarco had in mind for their application to do, a high-fidelity prototype was made.

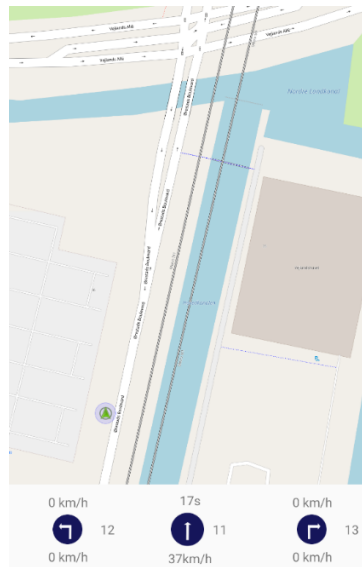


Figure G.1: High Fidelity Prototype

The high-fidelity prototype, shown in Figure G.1, displays a countdown to signal change, on top of the arrows as well as the signal id on the right side of the arrows. Underneath the arrows it shows a speed recommendation to pass the intersection. This is based on the user's distance to the intersection and the timing of the signals in the intersection.

While the prototype was functional, it was not very desirable. The application did not feel safe to use while driving through traffic. Reason being that the application forces the driver to look back and forth between the speedometer, the speed recommendation and the road. This is unsafe as the drivers focus should always be on the road. Hence the requirements regarding safety and "at a glance" was established.

Appendix H

Implementation Component Diagram

