Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Accident Prevention Frontend Framework to Support Autonomous Driving

Frontend Framework für das Vermeiden von Verkehrsunfällen und zur Unterstützung des Autonomen Fahrens

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Zimmer Walter, M.Sc.
Creß Christian, M.Sc.

Author Mohammad Naanaa

Date March 15, 2022 in Garching

Disclaimer

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, March 15, 2022

(Mohammad Naanaa)

Abstract

Over the last years, autonomous driving technology has moved from the realm of science fiction to working prototypes [Cha17]. This promising development is expected to improve road safety, reduce traffic congestion, and reduce ecological footprint [Nas+20]. However, complex traffic scenarios, such as intersections, roundabouts, or exits from motorways with high traffic density and various weather conditions, are still considered challenges for this new technology [SXC16].

Providentia++ Project - a research project of the Technical University of Munich - aims to improve traffic flow and road safety in such traffic scenarios by overcoming the limitations of local sensor systems of a single vehicle. This goal is achieved by making the road infrastructure "smart" with various sensors. This enhanced infrastructure can then build a real-time virtual twin of the road that can be used for research and as a basis for the development of various services.

As a part of the Providentia project, this work strives to increase the trust and acceptance of autonomous vehicles and to improve road safety for human drivers. For this purpose, the project aims to design and implement a mobile app that will connect to the Providentia system to utilize its detection capabilities. The goal is to build a mobile platform capable of processing different accident prevention mechanisms and displaying useful information to the user, like jam/slowdown warning, accident/collision warning, and lane change recommendations. The mobile platform to be used is iOS.

Zusammenfassung

In den letzten Jahren hat die Technologie des autonomen Fahrens den Bereich der Science-Fiction verlassen und ist zu einem funktionierenden Prototyp geworden [Cha17]. Von dieser vielversprechenden Entwicklung wird erwartet, dass sie die Verkehrssicherheit erhöht, Staus reduziert und den ökologischen Fußabdruck verringert [Nas+20]. Komplexe Verkehrsszenarien wie Kreuzungen, Kreisverkehre oder Autobahnausfahrten mit hoher Verkehrsdichte und unterschiedlichen Wetterbedingungen gelten jedoch nach wie vor als Herausforderung für diese neue Technologie [SXC16].

Das Projekt Providentia++ - ein Forschungsprojekt der Technischen Universität München - zielt darauf ab, den Verkehrsfluss und die Verkehrssicherheit in solchen Verkehrsszenarien zu verbessern, indem die Limitierungen lokaler Sensorsysteme eines einzelnen Fahrzeugs überwunden werden. Dieses Ziel wird erreicht, indem die Straßeninfrastruktur mit verschiedenen Sensoren "intelligent" gemacht wird. Diese verbesserte Infrastruktur ist dann in der Lage, einen virtuellen Zwilling der Straße in Echtzeit zu erstellen, der sowohl für Forschungszwecke als auch als Grundlage zur Entwicklung weiterer Dienste genutzt werden kann.

Diese Arbeit im Rahmen des Providentia-Projekts zielen darauf ab, das Vertrauen und die Akzeptanz von autonomen Fahrzeugen zu erhöhen, aber auch die Verkehrssicherheit für menschliche Fahrer zu verbessern. Zu diesem Zweck wird für das Projekt als Ziel gesetzt, eine mobile App zu entwickeln und zu implementieren, die sich mit dem Providentia-System verbindet, um dessen Erkennungsfunktionen zu nutzen. Darauf basierend wird eine mobile Plattform entwickelt, die in der Lage ist, verschiedene Unfallpräventionsmechanismen zu verarbeiten und dem Benutzer nützliche Informationen anzuzeigen, wie z.B. Stau-/Verlangsamung-Warnung, Unfall-/Kollisionswarnung und Empfehlungen zum Fahrspurwechsel. Die dabei zu verwendende mobile Plattform ist iOS.

Contents

Chapter 1

Introduction

1.1 Providentia Project

This thesis is written in cooperation with Providentia++ Project - a research project of the Technical University of Munich aimed at improving traffic flow and road safety by overcoming the limitations of local sensor systems of a single vehicle. Providentia++ was founded in 2017 and since 2020 the project has been led by the Chair of Robotics, Artificial Intelligence and Real-time Systems at the Technical University of Munich's Department of Informatics. It is funded by the Federal Ministry of Transport and Digital Infrastructure (BMVI). Additional cooperative partners supporting the project are Fortiss, Valeo, Intel, Cognition Factory, Elektrobit, Huawei, 3D Mapping Solutions, brighter AI, Siemens, and Volkswagen.

This project comprises two significant aspects, hereinafter introduced in more detail: the enhancement of the present road infrastructure with various sensors to collect traffic data and the transformation of this data into a virtual copy - the so-called *Digital Twin*.

1.1.1 Road Infrastructure

The road infrastructure of this project includes a part of the **A9 Highway** and an extension into the **surrounding urban area** of Garching to the north of Munich depicted in Figure 1.1.

Figure 1.1: A map view of the *Providentia* road infrastructure, consisting of multiple measurement stations: three on a highway (S40/50/60) and four surrounding urban area of Garching (M70/80/90 & S110). The corresponding sections colored blue and green. S40-S50 section is highlighted in green as it will be the main stretch used for this project. Its backend is also displayed on the map.

The infrastructure is a constellation of 7 sensor stations equipped with more than 60 state-of-the-art and multi-modal sensors, providing a road network coverage of approximately 3.5 kilometers. The list of sensors used for measurements includes optical cameras, Light and Radio Detection and Ranging sensors (*LiDaR* and *Radar*). Figure 1.2 demonstrates the sensor setup of a single measurement point.

Figure 1.2: An image of a single Providentia measurement point on the A9 highway. Two optical cameras pointing in both road directions are visible (yellow circles on top). Two Radars pointing to the south can also be seen (red circles). Also, a *Data Fusion Unit* (DFU) collecting and fusing the sensory data is displayed on the image in the bottom left corner. (Source: [Krä+19].)

1.1.2 Digital Twin

The purpose of the road infrastructure is to collect sensor data, perform a data fusion of the measurements from different sensors to improve the detection capabilities, and finally detect vehicles and additional meta-data to map these into a virtual road model called *Digital Twin*. The idea of the Digital Twin is then to represent a relevant subset of the road section that can later be used for research purposes but also to offer some high-level applications and services.

1.2 Providentia App Development Goals

With the infrastructure provided by the Providentia project, this thesis extends the project by developing a proof-of-concept "**Providentia App**" application to provide the data from the Digital Twin to both human-driven and autonomous vehicles.

Over the last years, autonomous driving technology has moved from the realm of science fiction to working prototypes [Cha17]. However, complex traffic scenarios, such as intersections, roundabouts, or exits from motorways with high traffic density and various weather conditions remain one of the biggest challenges for both humans and this new technology [SXC16]. The restricted field of inner vision of a single vehicle is frequently not sufficient to evaluate these situations, and this is where the app integration will occur.

The abstract goal of the app is to approach this problem by utilizing the surrounding smart infrastructure enhanced by the Digital Twin. By adding the "external" knowledge from various road sensors, both humans and autonomous driving vehicles should be assisted by the app to improve road safety. This work focuses on two aspects of that idea:

1. "How can this app be connected to the existing infrastructure?"
2. "What safety-improving features are possible to implement on a mobile app?"

These questions are addressed in detail and solved by the contributions made in this work in the following chapters.

Chapter 2

Related Work

2.1 Providentia – A Large-Scale Sensor System for the Assistance of Autonomous Vehicles and Its Evaluation

The *Providentia* project introduces both an intelligent road infrastructure system and a *digital road twin* - a virtual copy of the road infrastructure, including detected vehicles. This thesis relies on the infrastructure provided by this project in all of its aspects. Therefore, the results of this thesis are seen as an extension of this project offering additional services built upon the original infrastructure.

2.2 Designing Human-Machine Interface for Autonomous Vehicles

The paper written by Debernard, Chauvin, Pokam, and Langlois and titled "*Designing Human-Machine Interface for Autonomous Vehicles*" aims to answer the question "What information should be displayed to the driver, how, and when?". The results of this work - which relies on the Cognitive Work Analysis framework - are used in the following sections of this project to make a better decision on how to present traffic data and warnings to the app user.

2.3 Deep Traffic Scenario Mining, Detection, Classification and Generation

The results of Aaron Kaefer's Master's Thesis titled "*Deep Traffic Scenario Mining, Detection, Classification and Generation on the Autonomous Driving Test Stretch using the CARLA Simulator*" are used as a foundation for this project. The goal of this work was to "create a collection of diverse driving scenarios, which are automatically classified and labeled by an algorithm that is capable of detecting various driving maneuvers and traffic scenes." Using these results, a risk scenarios detection mechanism was implemented and used by the Providentia system to generate warnings. The Providentia App builds upon this warning detection mechanism by allowing this information to be processed by third parties (application/autonomous vehicle).

Chapter 3

Connection to Infrastructure

The software architecture of this project involves three major components: **Data Fusion Units** ("DFU"s), **Bridge Server** (hereafter "backend"), and **Mobile App** (hereafter "frontend").

A DFU is responsible for running local sensors collecting raw data, sensor data fusion, and transformation of this data into a digital twin. The processed digital twin from a DFU is then published to the bridge server.

Each DFU is responsible for one road section (e.g., *S40-50*) and does the processing independently of other DFUs. It enables multiple DFUs to asynchronously publish their local traffic's digital twin updates to the bridge server, making the process scalable for an industrial application with numerous DFUs that allow the Providentia system to monitor many road sections simultaneously.

Upon receiving this data, the backend processes it and provides it to multiple frontend users that request it. The backend's main purpose is therefore to be a "single point of contact" for the frontend so that no knowledge of the actual DFUs setup on the roads is required.

Figure 3.1 displays the described software architecture.

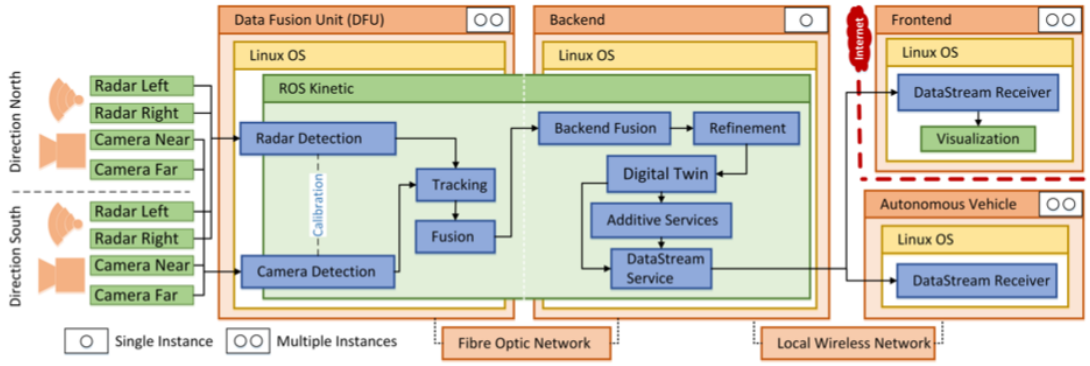

Figure 3.1: Platform architecture of the Providentia system. Three distinct components (*DFU*, *Backend*, and *Frontend/Autonomous Vehicle*) can be seen with the inter-component data flow. A *DFU* is connected to sensors on the left and processes the raw data. The data is used to construct a *digital twin*. Additive Services are then provided using that twin. These services are offered to the *Frontend*. (Source: [Krä+19].)

3.1 Data Exchange

For scalability purposes, a mobile app user should dynamically receive only the relevant part of the traffic data for the current position. This approach limits the amount of data that has to be received by the frontend and therefore improves both backend's and mobile device's

network performance. However, for this approach to work, a notion of data relevance has to be developed.

In this project, the relevance of the data is determined based on the principle of locality - while driving, only the current and potentially adjacent road sections in more complex road situations, e.g. intersections, are meaningful for the user. This principle requires a bidirectional exchange between the backend and frontend with a well-defined specification that is considered and described in the following sections.

3.1.1 Transmitted data specification

For the backend to decide which traffic data is relevant for the requesting user, the frontend has to specify its location first. This requires the frontend to transmit the user's GPS location that the backend server will process. Additionally, some meta-data, e.g. user's velocity and heading direction, is sent to allow for even better decisions of what data to send on the backend side. The complete transmission specification from the frontend's side is listed in Table 3.1.

Name	Type	Description
`timestamp`	`Double`	The interval between the date value and 00:00:00 UTC on 1 January 1970
`locationAccuracy`	`Double`	The accuracy of the course value, measured in degrees
`course`	`Double`	The direction in which the device is traveling, measured in degrees and relative to due north
`speed`	`Double`	The instantaneous speed of the device, measured in meters per second
`speedAccuracy`	`Double`	The accuracy of the speed value, measured in meters per second
`longitude`	`Double`	The longitude in degrees with positive values extending east of the meridian and negative values extending west of the meridian
`latitude`	`Double`	The longitude in degrees with positive values extending north of the equator and negative values extending south of the equator

Table 3.1: Transmitted data specification from the frontend to the backend.

3.1.2 Received data specification

Upon receiving user's GPS location, the server has to decide what part of the traffic data is relevant based on the user's location and send it back. As for the data specification, the backend server mirrors the format received from the DFUs but adds additional valuable meta-data for the mobile app.

The received message represents a traffic snapshot from a single road section. This `Traffic` message contains a list of detected vehicles of type `Vehicle` and additional data for identification (`id` and `section` fields), testing, and analyzing performance (several time-of-capture entries called `timestamp`). This structure is described in Table 3.2. Each `Vehicle`

from the aforementioned list also holds its locally unique `id`, a vehicle's `category`, `speed`, `shape`, and `position` described in Table 3.4. The corresponding vehicle's fields `category`, `position`, `speed`, and `shape` are also data structures holding vehicle's specific data. The `Category` field enumerates all possible vehicle categories a DFU can detect. The `Position` field stores the position of a vehicle in two dimensions - latitude and longitude, described in Table 3.5. The `Speed` field stores the speed of a vehicle in two dimensions - latitudinal and longitudinal - and the derived heading direction, described in Table 3.6. The `Shape` field stores the shape of a vehicle in three dimensions (width, length, and height), described in Table 3.7. And finally, the `Scenarios` field is an extension for a vehicle storing flags for all detectable hazardous scenarios associated with a vehicle, described in Table 3.3.

The complete specification of a single message is listed in Tables 3.2 and 3.8.

Name	Type*	Description
`id`	`Int`	The ID of the whole traffic sequence of one road section
`section`	`String`	The name of this section, e.g. `"S40-50"`
`timestampSecs`	`String`	The timestamp of this message, originating from a DFU, in s
`timestampNsecs`	`String`	The timestamp of this message, originating from a DFU, in ns
`timestampFull`	`String`	The timestamp of this message, originating from a DFU, formatted as `sec.nsec`
`vehicles`	`[Vehicle]`	The list of all detected `Vehicle` objects

Table 3.2: `JSON` specification of a single `Traffic` message received from the backend. *The data is transmitted in a stringified `JSON` format - the 'Type' column specifies the actual data type after parsing.

Name	Type*	Description
`wrongWay`	`Bool`	This vehicle is driving in a wrong way
`tailGateLevel`	`Int`	This vehicle is tailgating. Severity $\in \{0, 1, 2, 3\}$
`speeding`	`Bool`	This vehicle is exceeding the speed limit
`standing`	`Bool`	This vehicle is standing
`laneChangeLeft`	`Bool`	This vehicle is changing its lane to the left
`laneChangeRight`	`Bool`	This vehicle is changing its lane to the right
`cutInLeft`	`Bool`	This vehicle cuts into a lane to the left
`cutInRight`	`Bool`	This vehicle cuts into a lane to the right
`cutOutLeft`	`Bool`	This vehicle cuts out from a lane to the left
`cutOutRight`	`Bool`	This vehicle cuts out from a lane to the right

Table 3.3: `JSON` specification of a `Scenarios` class - an extension for `Vehicle` for describing risk scenarios.

Name	Type*	Description
`id`	`Int`	The ID of a single detected vehicle
`category`	`Category`	The category of a vehicle. Possible values $\in$ {"*bus*","*car*","*truck*","*motorcycle*"}
`position`	`Position`	The position of a vehicle as a vector $v \in \mathbb{R}^3$
`speed`	`Speed`	The speed of a vehicle as a vector $v \in \mathbb{R}^3$
`shape`	`Shape`	The shape of a vehicle as a vector $v \in \mathbb{R}^3$
`scenarios`	`Scenarios`	A container storing flags for all risk-scenarios associated with a vehicle

Table 3.4: `JSON` specification of a `Vehicle` class.

Name	Type*	Description
`position.x`	`Double`	The latitude of a vehicle in degrees with positive values extending east of the meridian and negative values extending west of the meridian
`position.y`	`Double`	The longitude of a vehicle in degrees with positive values extending north of the equator and negative values extending south of the equator
`position.z`	`Double`	The altitude of a vehicle - **Unused field set to** 0

Table 3.5: `JSON` specification of a `Position` class.

Name	Type*	Description
`speed.x`	`Double`	x-component of the vector
`speed.y`	`Double`	y-component of the vector
`speed.z`	`Double`	Angle (in radians) between the north $(0,1)$-vector and this speed vector indicating movement direction

Table 3.6: `JSON` specification of a `Speed` class.

Name	Type*	Description
`length`	`Double`	Length of a rectangle corresponding to a vehicle
`width`	`Double`	Width of a rectangle corresponding to a vehicle
`height`	`Double`	Height of a rectangle corresponding to a vehicle **Note:** Might be set to zero if computation is impossible, i.e. in bad weather conditions or darkness

Table 3.7: `JSON` specification of a `Shape` class.

Table 3.8: The complete specification of data stored in a `Vehicle`. A single `Traffic` message stores a list of such vehicles that were detected in a snapshot of a single road section from one DFU. *The data is transmitted in a stringified `JSON` format - the 'Type' column specifies the actual data type after parsing.

3.2 Frontend-Backend connection requirements

Following the described specification, a bridge server and a mobile application therefore form a bidirectional communication pair - after establishing a connection, the mobile app sends its user GPS location and the bridge server sends a relevant subset of the traffic data back.

To implement this communication, a connection protocol has to fulfill several crucial requirements:

- **Bidirectionality**
- **Low latency**
- **Security**

Bidirectionality is required as both sides of the communication have to send and receive data - in one direction the user's GPS data and in the other a part of the traffic data relevant for the given GPS location are sent.

Additionally, low latency is essential - traffic data is expected to be transmitted very frequently. The data publishing frequency of each DFU is approx. 25 Hz, i.e. approx. every 40 ms a DFU publishes a new portion of its traffic data. Moreover, as the user's vehicle continuously moves, a single vehicle can require data from multiple DFUs as multiple adjacent sections of the highway that can be simultaneously relevant. For a given single DFU s_i, its update frequency t_i is $40\,\text{ms} + \mathcal{L}$, where $\mathcal{L}$ denotes *additional transmission latency* analyzed in Section **??**. The actual update time T_{upd}, given multiple DFUs $s_1, ..., s_n$ and their respective frequencies $t_1, ..., t_n$, is therefore:

$$T_{upd} = min\{t_1, ..., t_n\} < 40\,\text{ms} + \mathcal{L}$$

A further concern for the data transfer is security. The frontend will send user's GPS position, while the backend will send the GPS positions of all vehicles of a certain part of traffic - both representing location data. Location data is considered as sensitive data subject to protection under European Union's *General Data Protection Regulation (GDPR)*, and hence regulation-conform handling of the data is required [Eur18].

3.3 Selection of protocols

With all aforementioned requirements, several protocol candidates had to be considered on two distinguished layers: the low-level *Data Transport Layer* and the more abstract *Application Layer*.

3.3.1 Data Transport Layer

For the underlying data transport layer, the selection of a protocol is straightforward. As both connection endpoints have their `IP` address, the underlying network protocol was chosen to be the `TCP` as it provides reliable, ordered, and error-checked delivery of a byte stream between applications running on hosts communicating via an IP network.

3.3.2 Application Layer

The more challenging part is the selection of the application layer data exchange protocol. The connection between two points can be separated into two phases: initial handshake to establish a connection and the continuous data exchange for maintaining the connection.

For the initial handshake, HTTP is considered a standard and is therefore also used in this project [Int22].

However, although HTTP is used for the initial handshake, it is not a good candidate for the listed requirements as it is a stateless protocol that does not allow continuous bidirectional exchange. Additionally, certain design features of HTTP interact badly with TCP for this project's described use cases, causing problems with network performance and with server scalability:

- Latency problems are caused by opening a single connection per request, through connection setup and slow-start costs.
- Further avoidable latency is incurred due to the protocol only returning a single object per request.
- Scalability problems are caused by TCP requiring a server to maintain state for all recently closed connections.

For such a bidirectional data exchange, a different popular application layer protocol was considered - the Websocket (ws) protocol. Websocket is a full-duplex communication protocol that supports communication over a single TCP connection.

3.3.3 Security

However, Websocket is known to have some considerations regarding the security aspect. As opposed to HTTP requests, Websocket requests are not restricted by the same-origin policy, i.e. a vanilla implementation of a Websocket server could expose a vulnerability to cross-site hijacking attacks [Kuo16]. To improve the security aspect and mitigate that risk, two security-improving decisions were made:

- Switching to the secure version of the Websocket protocol, called Websocket Secure (wss)
- Setting up an SSH tunnel between two endpoints

The secured wss version of the protocol offers two benefits for security: It encrypts the data between the frontend and the backend, preventing capturing or tampering of the sensitive data in the middle, and it avoids issues with Websockets on networks that employ so-called intermediaries (proxies, caches, firewalls). The latter is especially relevant for mobile operator networks, which are expected to be the primary network source since the ProvidentiaApp will mainly be used by users in moving vehicles, naturally constraining other connectivity types such as Wi-Fi [Kuo16].

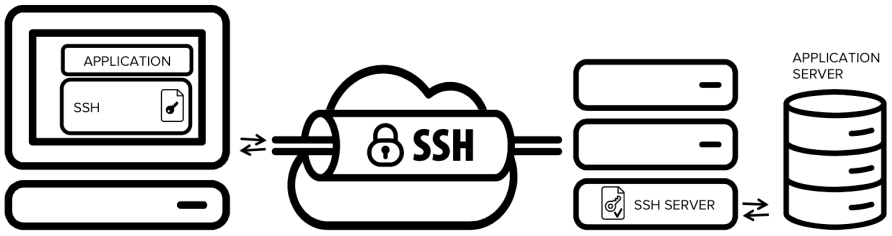

Figure 3.2: A visual representation of the connection between the frontend (seen here on the left side) and backend (seen here on the right side consisting of two abstract parts: SSH handler and the actual application server). In between of the two endpoints a bidirectional connection achieved by `Websocket Secure (wss)` is depicted, wrapped in an SSH tunnel isolating it from third-party listeners. Public keys required for the SSH tunneling of both parties can also be seen on the respective sides. (Source: [SSH]).

Utilizing SSH tunneling addresses the concern of `Websocket`'s vulnerability - after a tunnel between two endpoints is established, the server is accepting data only coming from it, eliminating a possibility of cross-site attacks. This final connection setup used in this project between the frontend and the backend is depicted in Figure 3.2.

Chapter 4

Providentia App

4.1 Target Platform

One of the major decisions that had to be made prior to the application development was the selection of the target platform. Following the goals described in Section 1.2, an application that can be used by a driver and potentially by an autonomous driving vehicle should be developed. Under this constraint, two approaches for software development are considered to achieve the goals: a native mobile platform or a web solution. Considering the requirements on time-constrained connectivity and performance imposed in Section 3.2, a less-performant web solution would be impractical. Hence, the decision was made to develop an application for a mobile platform.

Although various mobile platforms exist, two of them dominate the market share worldwide being installed on over 99% of all smartphones - *iOS* (with 28.27%) and *Android* (with 70.97%) [Glo22]. It is therefore sufficient to limit the mobile target selection to one of the three options:

- Native *iOS*, e.g. in `Swift`
- Native *Android*, e.g. in `Kotlin`
- *Cross-platform* solution for both platforms, e.g. in `Flutter`

The cross-platform approach seems to be the most practical as it allows developing for both platforms simultaneously and additionally offers code reusability and easy maintenance. However, as cross-platform apps work by forming an independent layer on top of the native solutions, this approach also introduces some challenges:

- **Integration**: Apps have inconsistent communication between the abstract system-independent code and their target operating system.
- **Performance**: Apps have an additional overhead caused by the extra layer between the device's native and non-native components.
- **User Experience**: Apps are not able to take full advantage of native-only features to provide target-OS specific user experience - different for *iOS*- and *Android*-paradigms.

For this reason and taking into consideration the connectivity requirements froms Section 3.2 that require optimal performance and low-level network tuning, native solutions are preferred. Finally, having a major prior experience in *iOS* mobile applications development, the decision was made to develop natively for Apple's operating system. With the specified target platform, the list of application features is identified in the next section.

4.2 App Features

Derived from the development goals, the Providentia App offers two significant features enhancing driving:

- Map Visualization of the current surrounding road traffic
- Displaying warnings for potential risks

Both of these features are described in detail in the following. Additionally, their implementation details are introduced and discussed later in Section 4.4.

4.2.1 Road Traffic Map Visualization

As described in the *Data Exchange* Section 3.1, based on the user's GPS location transmitted to the backend, a subset of the traffic that is relevant for the user is received back. This traffic data, containing (among other information) a list of vehicles, should be displayed on a map around the user according to the following principles:

- Each vehicle is *transformed* to a marker on a map set to its detected GPS coordinates
- Each marker is *uniquely identifiable*
- Each marker is *translated* with the vehicle's movement

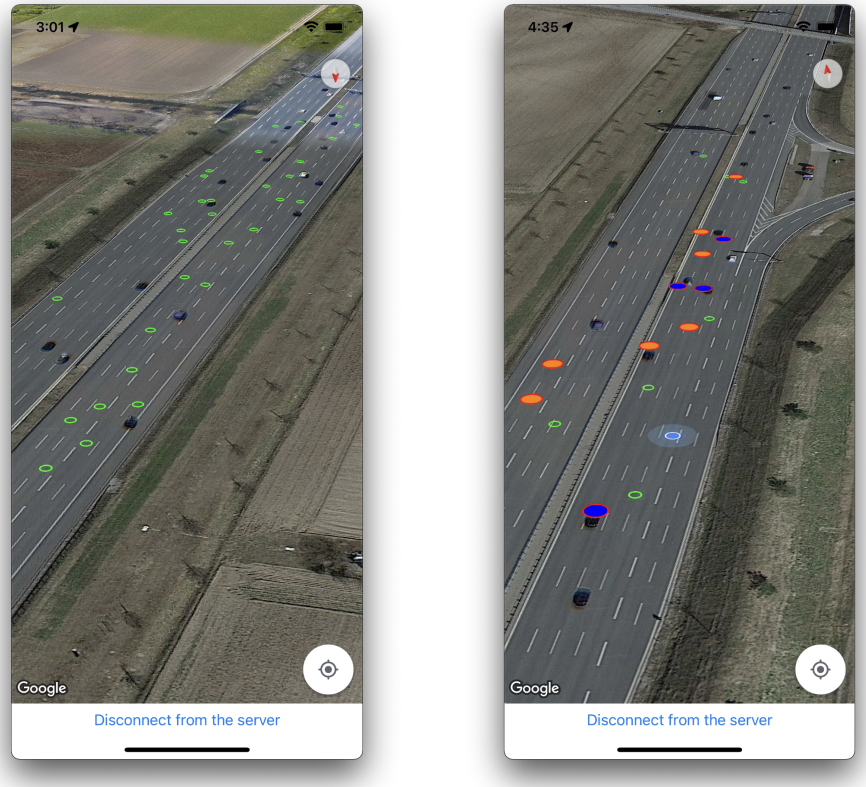

Figure 4.1: UI screenshots (*iPhone 13*) displaying two different approaches for visualizing detected vehicles on a map. The map on the left draws all vehicles with a single marker type indicating only vehicle's position. The map on the right generates scenario-based markers for each vehicle based on some additional meta-data (e.g. speed/type/risk scenarios).

Figure 4.1 demonstrates two approaches for visualizing vehicles on a map: uniform (all markers equal) or scenario-based (each marker can have different appearance based on vehicle's properties). Additionally, the user should be able to interact with the map in multiple ways:

- The map can be *moved around* preserving the positions of the markers
- The map can be *rotated* and *zoomed in and out* to ease navigating
- The map can be *centered back* to the user with a single button
- The map can be set to *follow the moving user*

4.2.2 Scenario-based Warnings

The Providentia project offers more than just a visualization of the surrounding vehicles. The system is also capable of detecting various potential hazards on the road. These scenarios are vehicle-specific, i.e. each vehicle can be associated with various (also multiple) scenarios. The list of all scenarios includes the most common risks that can be detected to improve safety: *standing, tailgating, driving in the wrong way, speeding*, and *lane changing*.

The goal of the app is to display warnings to the user in a meaningful way. The "meaningful way" means this notification should be apparent, however, not distract the driver from the actual vehicle steering.

In general, various notification methods exist. Since the app utilizes maps already, the decision was made to indicate potential hazards by modifying the cars' markers involved in a warning on the map. As described in Table 4.1, each detectable scenario has its own associated color. In addition to that, the hazardous vehicles' markers are larger by a factor of 2 compared to the non-hazardous marker and also pulsate, making them more visible.

Note: For vehicles associated with multiple scenarios, the largest risk's associated color is selected. This approach implies an order relation between different scenarios, and the table illustrates that by sorting the risks in descending order starting from the most dangerous.

Name	Marker	Description
`Wrong Way`		The vehicle is moving in the wrong direction
`Standing`		The vehicle is standing
`Tailgating`		The vehicle is tailgating another vehicle in front of it
`Speeding`		The vehicle's speed is exceeding the local limit
`Lane Change`		The vehicle is changing the lane
`None`		The vehicle is driving properly

Table 4.1: The complete table of all detectable vehicle-specific hazards sorted by their risk in descending order with their brief description. The table indicates how vehicle's marker color changes depending on the detected scenario. For reference, the standard "safe" marker with no detected hazard is also included at the bottom. Additionally, the table displays the difference in marker's radii based on risk detection - the hazardous vehicles' markers are larger and filled with color, whereas the safe one is smaller and void.

Figure 4.2 demonstrates the visual warning animation in practice for a vehicle that was tailgating another (also speeding) vehicle.

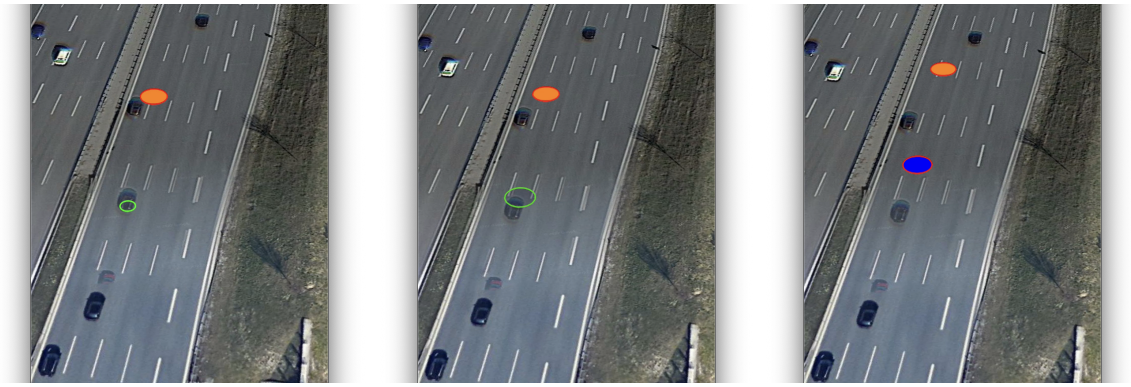

Figure 4.2: A sequence of images (left-to-right) demonstrating a risk scenario detection by the Providentia system. Two vehicles are moving in the upper direction and the first in the front is `speeding` (hence marked in orange). The second vehicle enters the road section (drawn as a green circle in the first image) and then gets too close to the first vehicle. The `tailgate` scenario is detected, and the marker becomes bigger (second image) and changes its color to dark blue (third image).

4.3 Overall Architecture

Having specified the features that have to be implemented, the next step of a software development is the design of system architecture.

4.3.1 Components

In an abstract way, the goal of the app is to receive traffic data, process it and transform it into useful (visual) information. From this abstract definition, several major components can be derived that would handle each of the aforementioned steps:

- **Data Model**
- **Connectivity Module**
- **Location Module**
- **Map Module**

Data Model

The *Data Model* is the most straightforward module to construct - this component should store the data processed by other components. Therefore, it is sufficient to mirror the specification of the transmitted and received data described in Tables 3.1 and 3.2.

Location Module

The *Location* module is responsible for providing user's location. This component fetches GPS data from the device's hardware and manages location data access permissions. Additionally, it encapsulates a state machine that handles the interaction between other components requiring location data and the device's hardware.

Connectivity Module

The *Connectivity* module manages network connectivity with the backend. This component implements the `Websocket Secure` connection mechanism as defined in Section 3.3. It also manages a state machine that supervises connection status changes and notifies all components requiring connectivity about these changes.

Map Module

The *Map* module is used to offer every functionality related to the map. Displaying roads, moving vehicle markers, and visualizing user's position are all tasks handled by this component.

4.3.2 Data Flow

The derived components are connected together by the data flow between them:

1. The *Location* module publishes user's location every time it is updated.
2. This location data is consumed by two modules - the *Connectivity* module (to send to backend) and the *Data Model* (to propagate to map).
3. The *Connectivity* module consumes that location data and sends it to the backend.
4. It also receives the traffic data from the backend passed down to the *Data Model*.
5. Aggregating data from the steps 2. and 4., the *Data Model* publishes the user's location and surrounding traffic data to the *Map* module.
6. The *Map* module displays that information and enables user interaction.

The resulting architecture with the data flow is depicted in Figure 4.3.

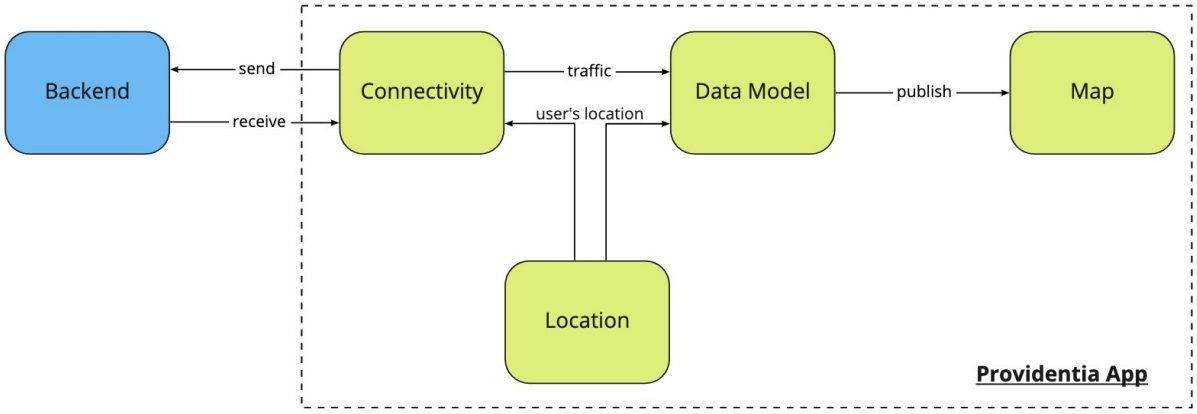

Figure 4.3: The complete architecture of the Providentia App. All internal components of the app are displayed in green. The backend, here in blue, is added to display app's connection with the outer world. The associated data flow between the components is also displayed with arrows indicating the direction of data flow.

4.4 Implementation Details

The aforementioned data flow to connect the app components was implemented using the declarative approach of the `SwiftUI` framework. In addition to that, several decisions regarding the implementation of the components were made that are worth further explanation. These decisions are therefore explained in this section for each relevant component.

4.4.1 Location Module

As stated in the specification, the app must send user's GPS position to the backend. Each currently supported Apple device has an integrated GPS module that is capable of determining user's location. Apple provides that functionality in `Swift` using the `CoreLocation` library. The *Location* module's goal is therefore to offer an interface from the hardware GPS module to the other components requiring that data. This task includes three important aspects, covered in the following:

1. Requesting access permissions
2. Fetching the GPS data from hardware
3. Publishing the GPS data to other components

Requesting access permissions

Location data is considered private data. It is therefore required by the system to ask the user for permission to access their location. The implementation requires configuring the so-called "*Purpose Strings*" for the project.

The *Purpose Strings* belong to the `Xcode` project settings, and they are used to describe the purpose of the authorization. In this case, a simple description "*For communicating with the road system*" is added to the "*Privacy - Location Always and When in Use Usage Description*"" purpose string in the `Info.plist` file managing the settings shown in Figure 4.4.

Key	Type	Value
Application supports indirect input events	Boolean	YES
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
Bundle name	String	$(PRODUCT_NAME)
InfoDictionary version	String	6.0
Privacy - Location When In Use Usage Description	String	For communicating with the road system
Bundle version	String	$(CURRENT_PROJECT_VERSION)
> Application Scene Manifest	Dictionary	(1 item)
Application requires iPhone environment	Boolean	YES
Executable file	String	$(EXECUTABLE_NAME)
> Supported interface orientations	Array	(1 item)
> Supported interface orientations (iPhone)	Array	(3 items)
Privacy - Location Always and When In Use Usage D...	String	For communicating with the road system
> LSApplicationQueriesSchemes	Array	(2 items)
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
> Launch Screen	Dictionary	(1 item)
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
> Supported interface orientations (iPad)	Array	(4 items)
Bundle version string (short)	String	$(MARKETING_VERSION)

Figure 4.4: A screenshot of the `Info.plist` file managing project settings in `Xcode`. The purpose string related to accessing location data is highlighted in blue.

Fetching and publishing the GPS data

After specifying *Purpose Strings*, it is now possible to request access to the user's location data. Following Apple's specification [App22], the standard approach is to define a wrapper for the `CLLocationManager` object provided by Apple's `CoreLocation` framework. The idea of the wrapping is that the `CLLocationManager` object offers OS-level control for interacting with GPS hardware, whereas the wrapper serves as a state machine for different possible GPS tracking states. Additionally, the `LocationManager` wrapper can be extended to publish the data to other components.

This implementation is demonstrated and explained in detail in Code 1.

```
1  // A wrapper for the `CLLocationManager` to manage the user location data
2  class LocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {
3      // The actual system location manager wrapped
4      private let locationManager = CLLocationManager()
5
6      // A webController that consumes the location updates
7      private var webController: WebSocketController
8
9      // Initialize the wrapper
10     override init() {
11         // Settings determining the accuracy and other aspects
12         locationManager.desiredAccuracy = kCLLocationAccuracyBestForNavigation
13         ...
14
15         // Tell the system that this wrapper will handle all updates
16         locationManager.delegate = self
17     }
18
19     // executed whenever the location authorization rights are changed
20     // implements logic to react for changed authorization status
21     func locationManager(_ manager: CLLocationManager,
22         didChangeAuthorization status: CLAuthorizationStatus) { ... }
23
24     // executed whenever the location was updated
25     func locationManager(_ manager: CLLocationManager,
26         didUpdateLocations locations: [CLLocation]) {
27         // logic to process the new location fetched from GPS module
28         ...
29         // after processing, propagate to the webController
30         webController.sendLocation(locations.last)
31     }
32 }
```

Listing 1: A code snippet of the `LocationManager.swift` wrapper for the `CLLocationManager` object provided by Apple's `CoreLocation` framework. This wrapper initializes the actual GPS location manager in `init()` and declares itself as a delegate to react to changes from the `CLLocationManager`. There are two possible types of changes that this wrapper must react to: changes in access rights (using `locationManager(_:didChangeAuthorization)`) and user location updates (using `locationManager(_:didUpdateLocations)`). The latter method is used to propagate the location updates to the `WebController` that sends these to the backend.

4.4.2 Connectivity Module

Another essential component of the Providentia App is the *Connectivity* module as both main features of the app require a continuous connection to the backend as specified in the requirements in Section 3.2. This component is responsible for multiple connectivity aspects described in more detail in the following:

- Connection management using `WebSocket Secure` protocol
- En- and decoding for sending/receiving data
- Publishing the new data to other components consuming these updates

Connection management using `WebSocket Secure` protocol

The most important part of the *Connectivity* module is clearly the connection management. This process includes:

1. Establishing an initial connection over `HTTP`
2. `HTTP` upgrade to the `WebSocket Secure` protocol
3. connection supervision covering edge cases (e.g. connection loss)

The implementation of these connection management aspects is described in detail in Code 2.

En- and decoding for sending/receiving data

After establishing the connection to the backend, the core task of the *Connectivity* module can be performed - sending and receiving data. This task includes working with the socket connection but also processing the data. The data is sent and received as a `stringified JSON` message, so the processing involves en- and decoding the message for sending/receiving, respectively. The implementation of this aspect is described in detail in lines `1-32` of Code 2.

Publishing the new data to other components consuming these updates

Finally, the received data should be automatically propagated to other components that consume this update - e.g. the map. This functionality is achieved using `SwiftUI`'s declarative data flow approach. The implementation of this aspect is also described in detail in the subset of `WebSocketController` implementation in lines `33-46` of Code 2.

Connectivity using `SSH tunneling` on an iPhone

Another important aspect of the *Connectivity* module implementation is the integration of the `SSH tunneling`. Although the measures defined in the *Security* Section 3.3.3 significantly improve the data transfer security between the app and the backend, they also introduce an implementation challenge as the *iOS* limits the usage of external port forwarding tools running in the background to only 30*s*. This limitation requires implementing a custom `SSH tunneling`, which - given the scope of this project and the time constraints - is left to be implemented in the future iterations of this project.

```
1  class WebSocketController: NSObject, URLSessionWebSocketDelegate {
2      // The `URL` of the server that this controller should connect to
3      private let serverURL: String
4      // Session responsible for `WebSocket` connection
5      private var session: URLSession!
6      // Socket used for the connection
7      private var socket: URLSessionWebSocketTask!
8      // Queue on which the delegate will observe the connection status changes
9      private let delegateQueue = OperationQueue()
10
11     // Connect to the server using `serverURL`
12     public func connect() {
13         // start a default (HTTP) session
14         self.session = URLSession(configuration: .default,
15                                   delegate: self, delegateQueue: delegateQueue)
16         // prepare to upgrade to a web socket connection
17         self.socket = session.webSocketTask(with: URL(string: self.serverURL)!)
18         // establish a connection by initiating the handshake
19         self.socket.resume()
20     }
21
22     // Disconnect from the server
23     public func disconnect() {
24         self.socket.cancel(with: .goingAway, reason: nil)
25     }
26
27     // Detect successful connection
28     internal func urlSession(..., didOpenWithProtocol) { ... }
29     // Detect clean disconnect
30     internal func urlSession(..., didCloseWith, reason) { ... }
31     // Detect connection invalidation with error
32     internal func urlSession(..., didBecomeInvalidWithError) { ... }
33
34
35     // Data decoded as `Traffic` to publish to other components
36     @Published public var traffic: Traffic?
37
38     // Data coders for sending and receiving
39     let decoder = JSONDecoder()
40     let encoder = JSONEncoder()
41
42     // Send a stringified message using encoder
43     func send(text: String) { ... }
44
45     // Listen for incoming messages and decode these
46     func listen() { ... }
47 }
```

Listing 2: A code snippet of the `WebSocketController` class implementation demonstrating its relevant functionality. The first block (lines 1–32) implements connectivity management, including `connect()` and `disconnect()` methods for establishing and terminating a connection. A reaction mechanism for detected connection status changes - opening, closing and invalidating - is also implemented here. The second block (lines 35–46) implements the communication with the backend and propagation of received data. Two methods - `send(...)` and `listen()` - allow for bidirectional data exchange. The data is transmitted as a `stringified` JSON, so sending data involves an encoder and receiving data requires a decoder, both initialized here. The received data is propagated to other components using `SwiftUI`'s `@Published` property wrapper.

4.4.3 Map Module

The **Map** module is one of the largest components as all features of the app utilize it. Therefore, numerous aspects had to be considered during the development. Each aspect is explained in detail in the following.

HD maps usage

To display vehicles on a map, an underlying map has to be loaded. During the development, two approaches for map usage were tested: Having an HD map of the road or downloading necessary map tiles dynamically as required.

The first approach offers support for roads with well-defined lanes and very tidy appearance, as depicted in Figure 4.5. However, it also introduces scalability problems caused by two factors:

- Missing HD maps for most road sections
- Memory or networking problems for bigger infrastructures

Most of the actual roads do not have digitized HD maps. Following this approach, a future extension of the Providentia road network would require an additional and potentially expensive step of creating an HD map for each new section. Even disregarding the cost aspect, it would require either downloading and storing all these sections on the device - causing a memory concern - or dynamically downloading new sections while driving - increasing network pressure. Hence, due to the lacking scalability, this approach was rejected.

Figure 4.5: On the left, a single section of an HD road map with a length of $440.12\,m$. On the right, an HD road map consisting of multiple such sections is overlayed on top of a satellite map. This image also includes vehicles for demonstration purposes, visualized as models by their category (e.g. "car" or "truck") and colored by their speed with red being the slowest and green being the fastest.

Third Party Maps

Instead of using custom and handcrafted maps, it is possible to utilize existing map APIs. For *iOS*, two mapping services were considered: **Apple** maps and **Google** maps offering `MapKit` and `Google Maps SDK` frameworks, respectively.

Both of these services support various map types, including:

- `normal` for displaying human-built features and abstract roads
- `satellite` for satellite photograph data

- `terrain` for topographic data

Figures 4.6 and 4.7 depict the difference between these three map types fetched from `Google Maps SDK` and `Apple Maps MapKit`, respectively.

Figure 4.6: Three different map types from `Google Maps SDK` displaying the same road section near Garching. This road section is a part of the Providentia System. The map types from left to right: `normal`, `satellite`, and `terrain`. No major differences between `normal` and `terrain` map types are recognizable other than coloring as the road section is mostly flat. This comparison demonstrates that only the `satellite` map type (in the middle) displays road lanes.

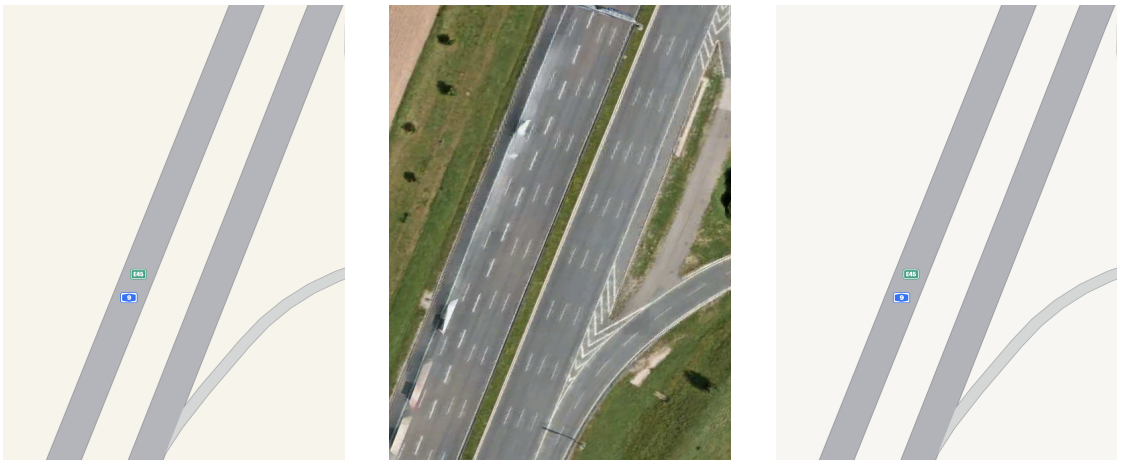

Figure 4.7: Analogous to Figure 4.6, three different map types from `Apple Maps SDK` displaying the same road section near Garching. The map types from left to right: `normal`, `satellite`, and `terrain`. This comparison shows the same pattern where only the `satellite` map type (in the middle) displays road lanes.

For Providentia App's use cases the `satellite` map type is the only one with visible road lanes. Therefore, the decision was made to use `satellite` map type.

Selection between Google Maps and Apple Maps

As can be seen in Figures 4.6 and 4.7, both `Google Maps SDK` and `Apple Maps MapKit` frameworks offer the comparable `satellite` map type. Furthermore, both frameworks offer comparable functionality, such as map interactions and marker generation. Therefore, since the decision was made to use `SwiftUI` GUI-framework for the app development, the selection of the mapping service comes down to the ease of mapping framework's integration.

Apple's `MapKit` is currently undergoing its transition period as it was initially developed to integrate into Apple's old GUI-framework called `UIKit`. The old framework was created with a declarative paradigm of event-driven UI, whereas the new `SwiftUI` framework follows the declarative paradigm. Since these paradigms differ by the way they handle the app's control flow, it is not possible to directly use the old code base with the new GUI-framework. Although Apple has tried to make the transition as imperceptible as possible, this incompatibility currently limits some functionality of `MapKit`'s integration. For example, as of `SwiftUI 3.0` version, it is not possible to change the map type programmatically without using the `UIViewRepresentable` bridge protocol that wraps the old `UIKit` map.

Google's `Map SDK` faces the same challenge when integrating with `SwiftUI`, however, it offers more map interaction functionality when bridging with `UIViewRepresentable`.

It is worth mentioning that Apple's `MapKit` framework is currently being actively developed. With a help of AI, the next versions of this framework will offer three-dimensional HD road maps, as seen in Figure 4.8.

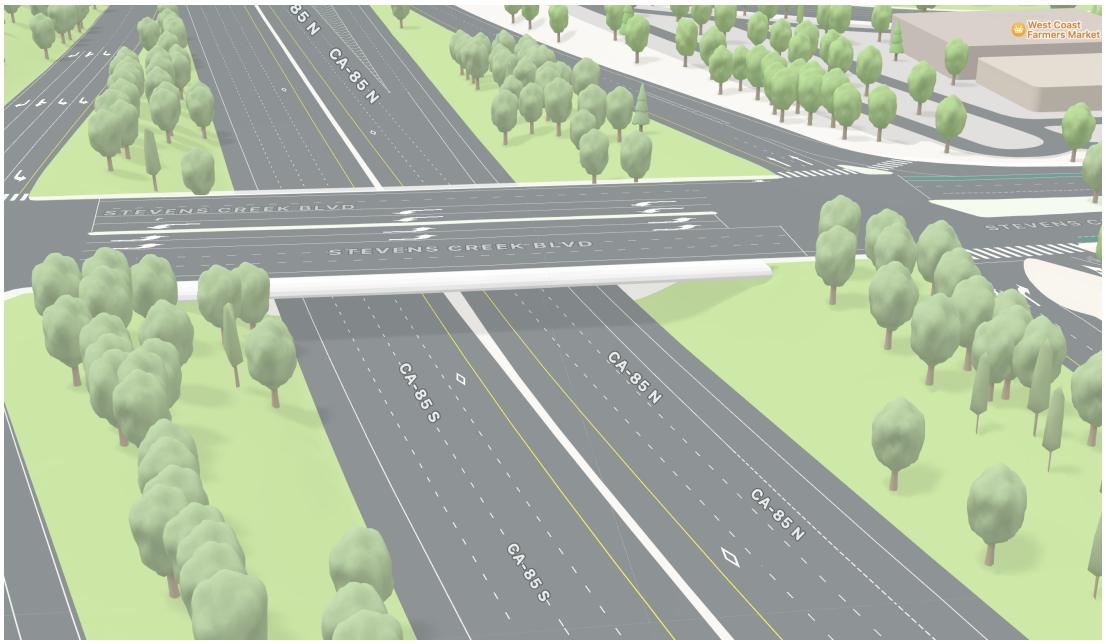

Figure 4.8: A highway intersection fetched from Apple's `MapKit` mapping framework in Cupertino, CA. This image displays a three-dimensional HD road map with lanes and elevation. Also, road surface marking (arrows and separator lanes) is visible. This feature is currently available only in several regions of the United States.

However, the new version is still in development and is currently only available in some parts of the United States. Therefore, due to currently existing limitations, the decision was made to use `Google Maps SDK`.

API Integration of `Google Maps SDK`

In general, `Google Maps SDK` is a paid service following the so-called "pay-as-you-go" pricing model that sets the price depending on the number of requests. However, the base functionality - such as loading a simple map, i.e. instantiation of a `GMSmapView` object - is offered free of charge. This functionality is sufficient for Providentia App's use purposes. Therefore, the API integration of `Google Maps SDK` framework can be broken down into only three steps:

1. Setting up a *Google Cloud* project
2. *Integrating SDK* into the `Xcode` project

3. Adding an *API key* to instantiate objects

The first step is required to manage services, credentials, billing, APIs and SDKs (as *Google* offers more than one mapping service). The complete setup process is described in [Goo22a].

In the second step, specified in [Goo22b], a code dependency for `Google Maps SDK` is integrated into the Providentia App's `Xcode` project. `Google Maps SDK` is a proprietary framework so the integration is achieved by binding binaries into the project. For this purpose, an open source dependency manager called `CocoaPods` is used. This manager is easy to install using `gem` as described in Code 3.

```
sudo gem install cocoapods
```

Listing 3: A `bash` command to install **CocoaPods** using `gem`.

After that, a `Podfile` is created in the project's root directory to install the API and its dependencies. An example setup used for integration in this project is described in Code 4.

```
# Source for CocoaPods specs (always the same)
source 'https://github.com/CocoaPods/Specs.git'
# Specify the target app
target 'ProvidentiaApp' do
# Specify what pod has to be added, here Google Maps SDK and its version
pod 'GoogleMaps', '6.0.1'
end

# Modify configuration
post_install do |installer|
installer.pods_project.build_configurations.each do |config|
    # Exclude arm64 architecture only for the simulator
    # to support intel-architecture-based iPhone simulator
    config.build_settings["EXCLUDED_ARCHS[sdk=iphonesimulator*]"] = "arm64"
    end
end
```

Listing 4: The content of the `Podfile` used to integrate `Google Maps SDK` into the Providentia App `Xcode` project.

Using a terminal, `pod install` can now be run in the `Podfile`'s directory to install the dependency finishing the second step.

In the final third step, the `GoogleMaps` library can now be imported to use in the project. This process requires providing API key generated in step 1. to the app during its initialization. In `SwiftUI`, it requires defining an `UIApplicationDelegate` that implements `UIApplicationDelegate.application(_:didFinishLaunchingWithOptions:)` function. This process is described in more detail in Code 5.

Map integration into UI using `GMSMapView`

After the initial framework integration, a map can now be instantiated using the `GMSMapView` class. As previously mentioned, Google's mapping library was developed to integrate with Apple's old `UIKit` GUI-framework. To make it compatible with the new `SwiftUI` GUI-framework, a wrapper conforming to the `UIViewRepresentable` must be declared. This

```swift
// Import the SDK
import GoogleMaps

// Main App struct
@main
struct ProvidentiaApp: App {
    // Custom `AppDelegate` to ensure correct setup
    @UIApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    // The main web controller of the app
    @StateObject private var webSocketController = WebSocketController(
                            serverURL: "wss://localhost:31500/")
}

// Custom `UIApplicationDelegate` to ensure correct setup
class AppDelegate: NSObject, UIApplicationDelegate {
    func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {
        // Provide API Key generated in step 1. to Google Maps services
        GMSServices.provideAPIKey("YOUR_API_KEY")
        return true
    }
}
```

Listing 5: A code snippet of the `ProvidentiaApp.swift` structure used to initialize the Providentia App. This code snippet demonstrates the integration of the `Google Maps SDK` framework using the API key generated in step 1. of the setup process. `SwiftUI`'s ProvidentiaApp struct is responsible for the app initialization. This structure defines a `UIKit`'s `UIApplicationDelegate` that offers a customizable initializer `func application(...)` allowing custom setup to provide the API key required for mapping services. Additionally, the initialization of the `WebSocketController` object responsible for the connection to the backend required for mapping is shown.

wrapper stores the actual `GMSMapView` and allows `SwiftUI`'s declarative UI handling to render that map on changes by using two wrapper methods:

- `func makeUIView(context:Context) -> GMSMapView`
- `func updateUIView(mapView:GMSMapView, context:Context)`

The `makeUIView(...)` method is used to instantiate the `GMSMapView` only once at the beginning, whereas the `updateUIView(...)` is invoked every time the state of the app binded with the map changes. Code 6 displays the implementation of this wrapper in detail.

```swift
// GUI framework
import SwiftUI
// Mapping framework
import GoogleMaps

// Wrapper for the GMSMapView
struct GoogleMapsView: UIViewRepresentable {
    // (3) A controller that provides all data from backend
    // and stores active markers of this map
    @ObservedObject var webController: WebSocketController

    // (1)
    func makeUIView(context: Context) -> GMSMapView {
        // initialize a full-frame map centered at the hard-coded camera
        let mapView = GMSMapView(frame: CGRect.zero,
            camera: GMSCameraPosition.providentiaCamera)

        // set map's type and other properties
        mapView.mapType = .satellite
        ...

        return mapView
    }

    // (2)
    func updateUIView(_ mapView: GMSMapView, context: Context) {
        // Update logic for the existing map
        // [described in the next section]
        ...
    }
}
```

Listing 6: A code snippet of the `GoogleMapsView.swift` structure wrapping `GMSMapView` to draw a map using `SwiftUI` GUI-framework. This wrapper conforms to the `UIViewRepresentable` protocol by implementing two UI rendering functions commented with (1) and (2). The first function is called to initialize the map during the setup, whereas the second function is called to redraw a map on app's state change. During the setup and following the previously specified software architecture, the wrapper is connected with a `WebSocketController` as seen in (3).

Mapping moving objects onto the map

Finally, the two main features of the app, introduced in Section 4.2, are implemented in the `updateUIView(_ mapView:GMSMapView, context:Context)` function.

The `GoogleMapsView` wrapper receives data from the observed `WebSocketController` that handles the connection to the backend. By using the `@ObservedObject` property wrapper on this controller, changes can be published automatically invoking the `updateUIView(...)` function that redraws the map when new traffic data is received from the backend.

Every time new traffic data is received, it contains a list of vehicles. This list can contain both vehicles that (1) were already detected in the previous messages (these vehicles' markers must simply be updated) and (2) newly registered vehicles (requiring new markers to be displayed). This process is described abstractly with an algorithm in pseudo-code 7.

```
func updateUIView(_ mapView: GMSMapView, context: Context) {
    traffic.vehicles.forEach { vehicle in
        // check if it is a new vehicle
        if not alreadyActiveMarkers.contains(vehicle) {
            // generate a new marker and assign it to the map
            let newCarPolygon = Marker()

            // assign map, marker id, color, radius, and GPS position
            // this marker takes into account detected risk scenarios
            newCarPolygon.map = mapView
            ...

            alreadyActiveMarkers.append(newCarPolygon)
        } else {
            // already existing vehicle
            let carPolygon = alreadyActiveMarkers.get(vehicle)

            // animate its color, shape, and GPS position change
            carPolygon.color = determineCarFillColor(using: vehicle)
            ...
        }
    }

    // logic to remove irrelevant and old markers to free memory
    ...
}
```

Listing 7: A pseudo-code snippet for the func updateUIView(...) function that demonstrates the algorithm to draw markers on the map for vehicles received from the backend.

The actual implementation contains a more dedicated memory management - the part that was simplified in the pseudo-code. There is a need for a special data structure that decides what marker is not active anymore and should be removed from the map (e.g. the associated car is already outside of the road section) This data structure must store references to all active markers, and it also must take into account that the app can be connected with multiple road sections sending messages independently. The solution was to use a dictionary of type `{String:[Marker]}` mapping a road section name to a list of its active markers. This data structure is depicted in Figure 4.9.

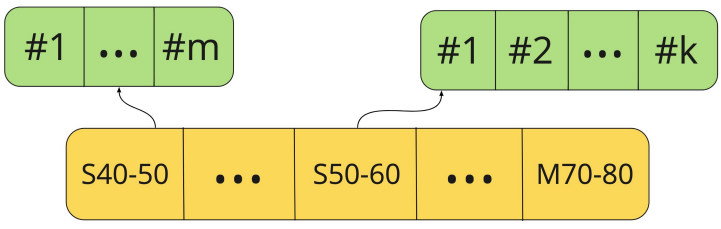

Figure 4.9: An abstract representation of the data structure for the active vehicle markers displayed on the map. The internal structure is a dictionary of type `{String:[GMSCircle]}`. Each key is a string storing a road section name (here, *S40-50, S50-60, M70-80* and others, denoted by "...") The associated value for each key is a list of markers (`GMSCircle`) for active vehicles of the associated road section. In this example, *S40-50* section stores m and *S50-60* stores k active markers as a list.

Chapter 5

Analysis

The analysis chapter can be structured in two major sections: **Feature-specific Performance** and **Mobile Device Performance**. The prior covers performance analysis of app's high-level functionality, whereas the latter comprises device's hardware performance analysis. Both sections are covered in the following.

5.1 Feature-specific Performance

Following the requirements imposed in the *Connection To Infrastructure* Chapter 3, a feature-specific performance analysis can be conducted to measure whether the requirements are fulfilled for the specified features. The following essential aspects - used by both main app features defined in the *App Features* Section 4.2 - are analyzed and covered in detail hereinafter:

- Total time delay (DFU to App)
- System throughput capability
- Connection security on iPhone
- GPS module precision

5.1.1 Total Time Delay

One of the main Providentia App goals is the *accident prevention* aspect. The driver should be notified about potential hazards in advance, making the warning mechanism time-critical. It is therefore essential to minimize the time delay from the generation of a potential warning until this warning could have been received.

$$DFU[\texttt{timestamp}] \Longrightarrow Backend \Longrightarrow AppDecoder \Longrightarrow Map[\texttt{timeToDisplay}]$$

Figure 5.1: A scheme describing the complete schematic path of a single message generated on a data fusion unit. After generation, a `timestamp` is attached to the message and it travels to the backend where it is processed. After processing, the message is sent to the App where it also has to be decoded. After the decoding process, a second `timeToDisplay` is computed as the difference between `Date.now` and the `timestamp` value. The message is now ready to be displayed on the map.

Each portion of the traffic data contains a `UNIX`-based `timestampSecs` field that captures the time the message was generated by a data fusion unit (*DFU*) from the road section's

infrastructure. This message travels to the app and after it is ready to be displayed on the map, a second `UNIX` timestamp called `timeToDisplay` is captured. The complete schematic message path is displayed in Figure 5.1.

For analysis, data was collected from 1095 real vehicles from a single road section *S40-50* and written into the `timeToDisplayLogs.txt` file. The average measured latency was 665 ms, the minimum and maximum values were 128 ms and 1362 ms, respectively. Table 5.1 demonstrates the time delay translation into meters traveled for different speeds for the average latency as well as both `min` and `max` latency edge cases.

Speed (km/h)	s_{min} (m)	s_{avg} (m)	s_{max} (m)
10 km/h	0.36 m	1.85 m	3.78 m
30 km/h	1.07 m	5.54 m	11.35 m
50 km/h	1.78 m	9.24 m	18.92 m
80 km/h	2.84 m	14.78 m	30.27 m
100 km/h	3.60 m	18.50 m	37.80 m
130 km/h	4.62 m	24.02 m	49.18 m
200 km/h	7.20 m	48.04 m	98.36 m

Table 5.1: A table demonstrating the time delay of a single traffic message converted into meters traveled for different vehicle speeds. Typical values for speeds were taken ranging from slow city driving to a highway with no speed limit. For each speed, three distances were calculated for different registered delays: the optimistic 128 ms (denoted as s_{min} for minimal delay), the average 665 ms (s_{avg}), and the pessimistic 1362 ms (s_{max}) meters traveled.

Figure 5.2: An image captured from the delay field test. On the left, a live output from the simulator connected to the backend showing a map with the road section *S40-50* is visible. A reference object (a massive cell tower) is highlighted on the map with a red circle and a perpendicular line to the road is also drawn in red. A truck is highlighted with a violet rectangle around its detected center drawn as a green circle. In the background, the actual road section *S40-50* with the truck (in violet) and the perpendicular line (in red) is seen. From this image, a delay of approx. one truck length can be seen with respect to the red line.

In addition to the theoretical values of meters traveled for the delay, a field test was performed to obtain an empirical result. During the test, a static reference object was selected (a massive cell tower) and a perpendicular line from this object to the road was drawn. A

video was captured with both live footage and the output from an iPhone simulator connected to the backend. The difference between the real vehicle position and the one displayed in the simulator was then estimated using a truck size of 16 m. Based on this approximation, a delay of approx. one truck length was observed, which translates into approx. 16 m meters traveled. With a measured truck velocity of approx. 80km/h (also a standard for trucks on german highways), this result matches the average estimate in the theoretical values Table 5.1 of 14.78 m confirming its validity empirically. Figure 5.2 displays the performed test.

5.1.2 System Throughput Capability

Using the measurements from the previous subsection, an additional metric can be analyzed. The *DFU*s publish data very frequently (approx. 25 Hz), and it is essential to process every message within this time frame. Otherwise, older messages have to either be dropped leading to "jumping" vehicles or enqueued eliminating the real-time aspect of the visualization as the queue might continuously grow. Although from these two options the prior is less destructive and is hence implemented by the Providentia App as a "worst-case" mechanism, it is even better to not have to face this problem at all by processing incoming messages in-time.

From the previously observed `timeToDisplay` data, it can be derived whether the app manages to process the messages in-time. For that, the consequent `timeToDisplay` entries are plotted against time. If the app processes each message before the next one arrives, the `timeToDisplay` values are expected on average to not change over time. It can be tested "optically" by fitting a regression line for the delay over increasing `id` and observing that it's slope is approx. zero indicating no change in average delay over time is occurring. This approach is seen in Figure 5.3.

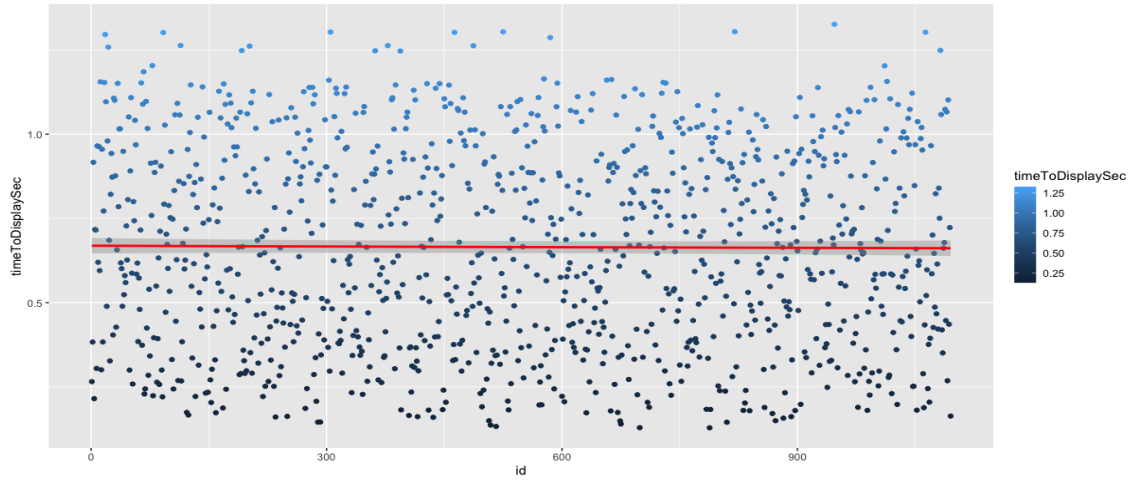

Figure 5.3: A graph demonstrating change of `timeToDisplay` measured in seconds for increasing `id` value. Each consequent message is assigned with a higher `id` value so this field is used to indicate increasing time. The dots are spread equally with no significant outliers suggesting no change in mean over time. Also, a regression line (in red) is fitted that seems to have a zero slope confirming the assumption of no change of mean over time.

However, this can also be proven mathematically by performing a statistical test on a linear regression model to show that the null hypothesis of the zero-mean change cannot be rejected - as demonstrated in Figure 5.4. Both approaches confirm that the app is capable of handling each traffic message within the required time frame.

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  6.829e-01  1.790e-02  38.142   <2e-16
id          -3.269e-05  2.830e-05  -1.155    0.248

(Intercept) ***
id
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.296 on 1093 degrees of freedom
Multiple R-squared:  0.001219,  Adjusted R-squared:  0.0003055
F-statistic: 1.334 on 1 and 1093 DF,  p-value: 0.2483
```

Figure 5.4: An output from fitting a linear regression model using `summary(lm(timeToDisplay ~ id))` command in R. The β_{id} regressor estimate is $-3.269 \cdot 10^{-5} s/id$ indicating negligible change of mean over time. In fact, even this small value is statistically insignificant under 95% confidence having a p-value > 0.05. The F-statistic having a p-value > 0.05 also does not allow to reject the null hypothesis H_0 with 95% confidence indicating no significant change of `timeToDisplay` mean over time.

5.1.3 GPS Module Precision

As defined in the *Transmitted Data Specification* Section 3.1.1, the app receives only the relevant portion of the traffic data based on the locality principle. This approach requires sending user's device GPS location associated with a certain variability that can be analyzed.

Two ways to test the precision of the device's GPS module were considered in the following:

- Internal device accuracy metric
- External observation

Internal Device Accuracy Metric

Each *iOS*-device has a built-in support for GPS location services. As specified in the *Transmission Specification* Table 3.1, the data provided by the GPS module also includes `locationAccuracy` field that measures the precision of the module. Figure 5.5 demonstrates the average accuracy measured over time by an *iPhone 13 Pro* on the highway during test driving.

External Observation

The high precision is confirmed by a highway test drive as depicted in Figure 5.6 demonstrating that this location accuracy is sufficient even to detect a single lane switch.

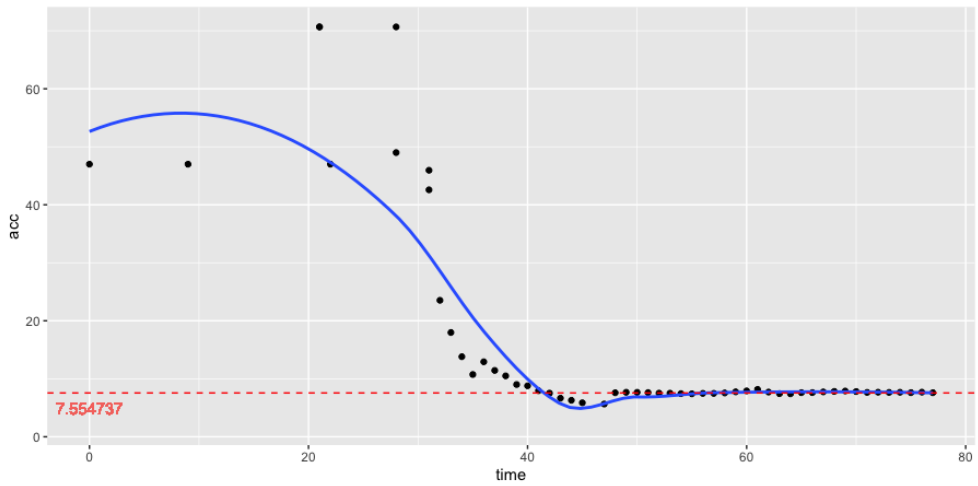

Figure 5.5: A graph demonstrating the change of location accuracy (measured in m as an "uncertainty" diameter) over time (measured in s) after starting the app. A local polynomial regression line (*LOESS*) is also fitted in blue to demonstrate the change of mean accuracy over time. Two phases are recognizable on the graph: the GPS module initialization phase (approx. the first 35 seconds) with an unstable low-accuracy location measurements and the running phase with a stable high-accuracy location precision. After the initialization, a mean accuracy diameter of approx. $7.56\,m$ is observed - highlighted with a red dotted line here.

Figure 5.6: Three screenshots made during a highway test drive demonstrating a lane switch. The user's GPS location is displayed as a blue circle centered in the middle of each screenshot. The vehicle starts in the second to last driving lane (colored red here) [left screenshot] then moves to the adjacent lane on the left (colored green here) and is displayed between the two lanes [middle screenshot]. The maneuver is completed and the vehicle is visible inside the green lane. This sequence confirms the actual location is detected within the road lane bound.

5.2 Mobile Device Performance

Another important aspect to be analyzed is the overall device performance that includes several hardware metric sets:

- Network workload
- Random Access Memory (RAM) load
- Disk utilization

5.2.1 Network

Following the app architecture described in the *Overall Architecture* Section 4.3, all network-related functionality was delegated to the *Connectivity* module. This module is one of the core components as all app features rely heavily on a continuous connection to the backend. In addition to that, a driver is expected to use the device's mobile data as connectivity alternatives such as `Wi-Fi` are not available for a driving vehicle. It was therefore crusial to optimize the network usage. Two types of measures were used for that: **low-level** and **high-level** optimization.

Low-Level Optimization

The idea of optimizing network usage on a hardware level utilizes the flexibility of the `WebSocket` protocol. This protocol allows sending both textual (`UTF-8`-encoded) and binary data. To reduce the potential 'empty bits" overhead of the `UFT-8` encoding, the latter data format can be used.

High-Level Optimization

Eliminating the overhead using a more efficient transmission encoding will not improve the network performance alone if there is still too much data being sent/received. Another approach of improving the network performance is therefore to optimize *what* data exactly is sent/received and *how frequently* this process should happen.

Regarding the first question, the *Received Data Specification* defined in 3.1.2 has a fixed structure potentially containing an "empty" information. This follows from the fact, that most vehicles are expected to not cause any risk situations but a field of empty "`scenario`" flags is sent nevertheless. This approach can be optimized by introducing a more flexible structure that accounts for that and only sends the `scenarios` when applicable, hence reducing the mean message payload and therefore its size.

Assuming a semi-constant sending rate from the road *DFU*s (whose frequency is subject to its own possible optimization) and given the high GPS precision covered in the previous section, it is also possible to address the second question and reduce the frequency of sending updates to the backend. Currently, the app sends user's location upon fetching a new portion of data immediately, which - given the highest possible accuracy set - happens quite frequently. Two ideas to optimize that without a major loss of accuracy are presented here: defining a device-internal *location update radius* that the user must physically exceed to send the next location update to the backend or defining a *timer* that constrains the transmission to every *x ms*.

Given the scope of this project and the time constraints, the two aforementioned *High-Level* optimization mechanisms are left to be implemented for future iterations of Providentia

App. Nevertheless, Figure 5.7 demonstrates a decent network usage graph with only *Low-Level* optimization applied during a real highway test.

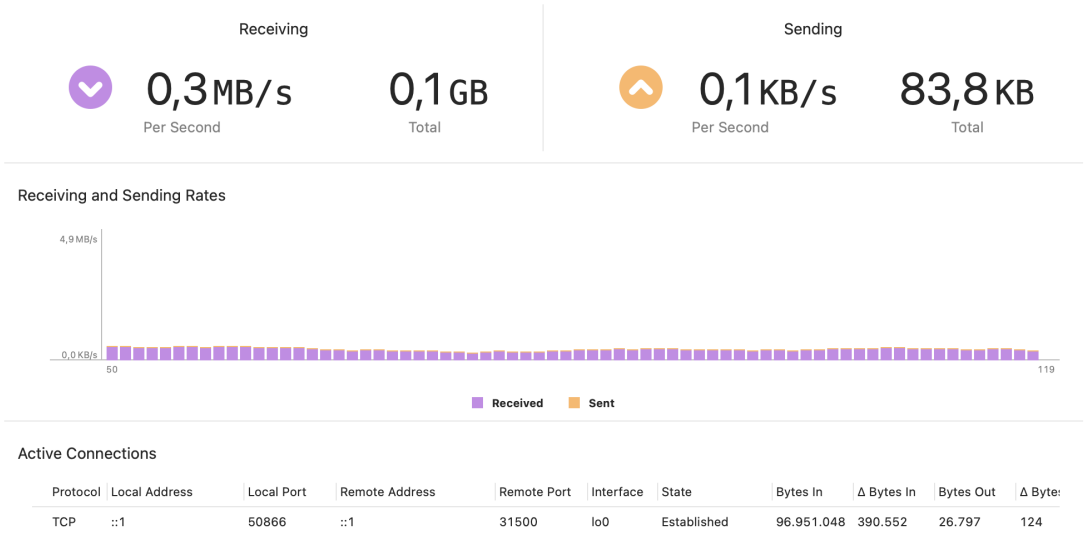

Figure 5.7: An output from the `Xcode` network usage analysis tool during a test drive through the *S40-50* road section. The first row indicates current and total network usage for receiving (on the left, colored violet) and sending (on the right, colored orange). The second row shows all reading and writing requests over time starting from $50s$ to $119s$ - the app was used for almost one minute before the benchmarking to establish a connection with the backend. A consistent network usage is observed on the graph of approx. $0.3\,MB/s$ and $0.1\,KB/s$ for receiving and sending, respectively. This graph also demonstrates that receiving data requires significantly more network than sending (approx. 3000-to-1 ratio), matching the expectation. The bottom "*Active Connections*" section confirms only one connection over TCP was observed, making the analysis valid.

5.2.2 Random Access Memory (RAM)

Among all app components defined in the *Components* Section 4.3.1, the most RAM-consuming one is the map module. This module is required to handle a constant flow of traffic data, managing numerous markers. In addition to that, this process constantly happens in RAM as working with the disk would slow the time-critical process due to the additional read/write overhead. Therefore, the app implements a number of clever solutions to memory management - such as reusing markers for multiple objects or detection and elimination of strong reference pointer cycles [Swi22] - to free up all objects when they are no longer needed. All these measures should lead to an almost constant RAM usage depending only on the number of active vehicles. Figure 5.8 confirms this statement displaying a controllable growth of RAM usage of approx. $700\,MB$ as the vehicle enters the *S40-50* highway road section and drives through it.

Figure 5.8: An output from the Xcode RAM usage analysis tool during a test drive through the *S40-50* road section. The x- and y-axes describe time in seconds and memory usage in MB, respectively. The data was captured for almost four minutes with approx. $160s$ of driving within the road section displayed on the graph. The moment of a rapid increase of RAM usage can be seen in the first seconds of entering the road section, followed by an almost constant RAM usage.

5.2.3 Disk

As the app does not involve any persistent data storage, disk usage does not happen. This is confirmed by the disk usage graph depicted in Figure 5.9 displaying no app-induced reading or writing requests.

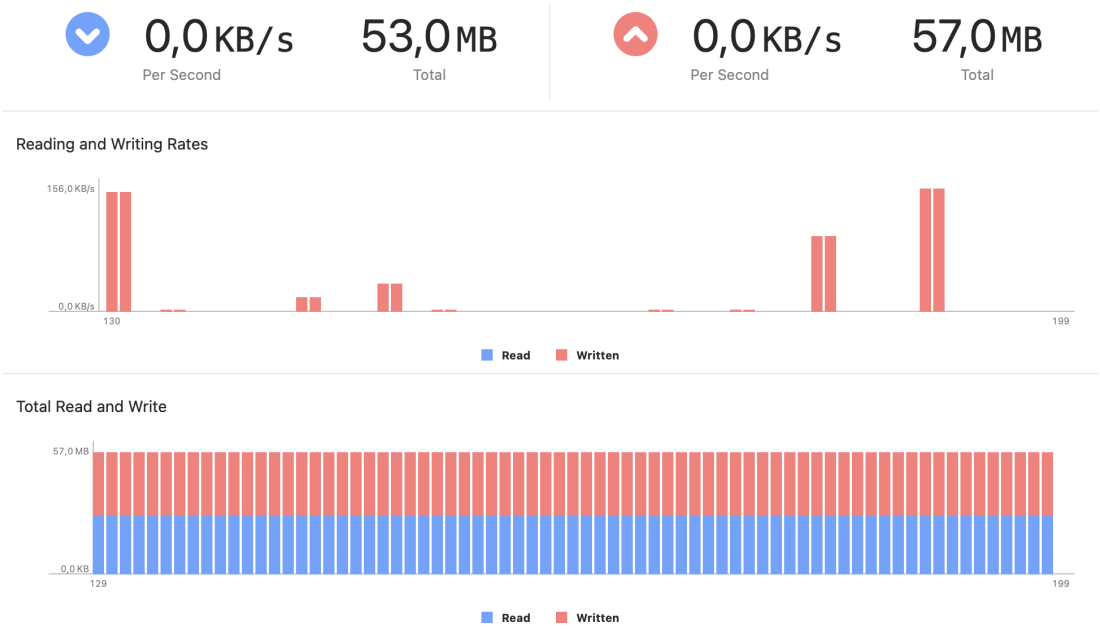

Figure 5.9: An output from the Xcode disk usage analysis tool. The first row indicates current ($0.0 KB/s$) and total disk usage for reading (on the left, colored blue) and writing (on the right, colored red). The second row shows all reading and writing requests over time from 130s to 199s - the app was used for two minutes before the benchmarking to establish a connection with the backend. The last row displays the total accumulated read/write disk requests graph over time from 129s to 199s. From all three metrics it can be seen that the disk is not actively used by the app. The total non-zero read and write values of approx. $55MB$ were caused by initially opening the app as well as logging some debugging messages.

Chapter 6

Summary

The efforts made in this work have dealt with the research questions asked in the introduction:

1. "How can this app be connected to the existing infrastructure?"
2. "What safety-improving features are possible to implement on a mobile app?"

Corresponding to the questions, two major contributions were made in this work covered in detail in the following sections.

6.1 Connection to Infrastructure

In this thesis, a connection interface between the backend and a mobile app was designed and specified. In addition to the interface, various connection protocol alternatives were investigated and compared based on several aspects, including transmission efficiency, scalability, and security. A continuous, efficient, and secure bidirectional data exchange using `WebSocket Secure` protocol between two endpoints was established. An option to use an additional security layer using `SSH tunneling` was investigated and - although only for a simulator but not a physical iPhone due to `iOS` restrictions - successfully tested.

The final connection interface was extensively tested in the *Analysis* chapter proving to fulfill the imposed requirements.

6.2 Application Development

As a result of this work, an `iOS` application named **Providentia App** was developed, documented, and tested.

The implementation follows the designed software architecture of Section 4.3 offering connection to the Providentia system. The app implements two safety-improving features available both for a human driver and potentially for an autonomous driving vehicle:

- Live tracking and visualization of the surrounding traffic on a map
- Scenario-based in-app warning system

Finally, in addition to the implemented features covered in the following subsections, another achieved milestone is the modular app architecture design with low coupling allowing for future extensions as described in detail in the *Outlook* chapter. To support that, the project implementation was extensively documented in `Xcode` to further ease future app increments.

6.2.1 Live Traffic Tracking and Visualization

The application supports live traffic tracking and map visualization for the road sections covered by the Providentia systems.

The tracking capability is achieved by the implemented connection to the backend constantly sending the relevant traffic data based on the locality principle. The app implements a data structure representing this data for supported road sections with all detected vehicles and their measured metrics, such as velocity and heading direction.

The visualization process is achieved with markers displaying the detected vehicles on a map. Different vehicle meta-data - like vehicle type or speed - can be used to generate a custom appearance for icons. This custom appearance supports shape, size, and color changes based on various parameters.

6.2.2 Scenario-based warnings

The Providentia App also utilizes the risk-scenarios detection capabilities of the Providentia system. Each vehicle has associated potential risk-scenarios, such as speeding or aggressive lane changes, that can be used to warn the vehicle about potential hazards ahead that might be undetectable by the driver or the autonomous vehicle otherwise. Figure 6.1 demonstrates an example of such locally undetectable risk that could have been detected by the infrastructure generating a warning in advance.

Figure 6.1: Four images from a video footage (left-to-right then top-to-bottom) with a car crash from the road section *S40-50* monitored by the Providentia system. All vehicles captured by the system are overlayed with their digital twin rectangles colored by their type. In the first image, a van (marked with a black circle) is standing in the middle of the left-most driving lane. However, for the quickly incoming car (marked with a red circle) it was not possible to locally detect that because three other vehicles were obstructing the visual contact. In the second image, the incoming driver detects the risk and starts emergency braking that causes the driver to loose control. The images in the bottom row demonstrate the devastating result of this accident. Running this recorded scenario with the app has confirmed to generate a warning that would have been sufficient to start braking much earlier.

Chapter 7

Outlook

This outlook deals with two questions in detail: "*What can be done next?*" and "*What is the utility of this project?*".

7.1 Future Feature Increments

As mentioned in the *Summary* chapter, the app architecture was designed to be as modular as possible. This approach allows for future increments that can be easily integrated into the existing app. These potential functionality extensions can be split in two groups covered hereinafter: **Human-Machine interaction** improvements and **functional** increments.

7.1.1 Improved Human-Machine Interaction

One of the project development goals was to design an app useful not only for autonomous driving vehicles but also convenient for human drivers. Multiple aspects involved in the Human-Machine-Interaction can therefore be improved, such as *map visualization*, *haptic feedback*, or *acoustic warnings*.

Map Visualization

The map visualization using circles of different colors and radii implemented in this project serves as a proof of concept. The visual aspect can be improved by introducing custom markers with a more complex appearance generation logic. This logic can for example use the available meta-data - like vehicle type or speed - to render unique markers.

Haptic Feedback

Each iPhone is equipped with a vibration engine called "*Taptic Engine*". Another candidate for improved interaction with the app is therefore an integration of a haptic feedback produced by this engine.

acoustic Warnings

Another Human-Machine interaction improvement candidate is closely related to the previous one - acoustic warnings. The driver is typically focused on the road and is not expected to keep a visual contact with the app. By using various acoustic warnings, driver's attention can quickly be gained. For example, this can be used to warn about detected hazards ahead so that the driver would have more time to react.

7.1.2 Functional Increments

The potential increments are not limited to the improvement of the Human-Machine interface. Functional increments are also possible to further increase the app utility. Hereinafter, two examples are covered to demonstate the posibilities: *new detectable scenarios* and *driving recommendations*.

New Detectable Scenarios

Currently, only vehicle-specific scenarios (such as speeding or tailgating) are available, however various **meta-scenarios** can also be integrated in the future iterations of the app development. Meta-scenarios can encompass some situations specific for not a single vehicle but a whole road sections - like traffic jam, bad road or weather condition.

Driving Recommendations

In combination with the aforementioned meta-scenarios, the app could be used not only to prevent accidents but also to improve the quality and comfort of driving. For example, the Providentia system could hint the user to switch to another lane because of some construction site ahead. Such local recommendations - especially connected with autonomous driving vehicles - could form a "hive mind" improving the overall traffic flow and safety.

7.2 Deployment

Another important aspect for the outlook is the real-world application of the project. Two major deployment possibilities are discussed in the following: human-centered *app deployment* and *application in autonomous driving*.

7.2.1 App Deployment

As demonstrated in this project, the Providentia app can improve driving for humans making it more predictable and safe. This approach can be scaled by offering navigation companies this detecting and warning functionality as a paid service. A navigation company can then subscribe to this service and following the connection specification designed in this project receive the traffic data. The company can then integrate both traffic information and the warnings into their own apps improving their navigation service. Another benefit of this approach is that it simplifies the project scaling process as the funding for new road sections enhancement can be supported by offering this paid service.

7.2.2 Application in Autonomous Driving

Another promissing deployment target for the project is the integration with autonomous driving vehicles. Autonomous driving vehicles have proven to successfully handle the driving process in both simple and even somewhat complex conditions. However for the most complex scenarios, where AI-driven vehicles inevitably fail and so do human drivers, the Providentia system can provide an additional support. This integration can then happen seamlessly - an autonomous vehicle would approach the complex road section supervised by

the Providentia system, connect to it automatically, and either use the data from the digital twin as an additional sensory data to make decisions or even receive complete path recommendations to follow.

Bibliography

[App22] Apple Inc. *Requesting Authorization for Location Services*. https://developer.apple.com/documentation/corelocation/requesting_authorization_for_location_services. Accessed: 2022-03-06. 2022.

[Cha17] Chan, C.-Y. "Advancements, prospects, and impacts of automated driving systems". In: *International Journal of Transportation Science and Technology* 6.3 (2017). Safer Road Infrastructure and Operation Management, pp. 208–216. ISSN: 2046-0430. DOI: https://doi.org/10.1016/j.ijtst.2017.07.008. URL: https://www.sciencedirect.com/science/article/pii/S2046043017300035.

[Eur18] European Commision. *General Data Protection Regulation (GDPR) and location data*. https://joinup.ec.europa.eu/collection/elise-european-location-interoperability-solutions-e-government/news/gdpr-and-location-data. Accessed: 2022-02-20. 2018.

[Glo22] GlobalStats. *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2022-03-04. 2022.

[Goo22a] Google. *Set up in the Google Cloud for iOS*. https://developers.google.com/maps/documentation/ios-sdk/cloud-setup. Accessed: 2022-03-04. 2022.

[Goo22b] Google LLC. *Set up an Xcode Project*. https://developers.google.com/maps/documentation/ios-sdk/config. Accessed: 2022-03-04. 2022.

[Int22] Internet Assigned Numbers Authority (IANA). *Hypertext Transfer Protocol (HTTP) Upgrade Token Registry*. https://www.iana.org/assignments/http-upgrade-tokens/http-upgrade-tokens.xhtml. Accessed: 2022-02-22. 2022.

[Krä+19] Krämmer, A., Schöller, C., Gulati, D., Lakshminarasimhan, V., Kurz, F., Rosenbaum, D., Lenz, C., and Knoll, A. "Providentia – A Large-Scale Sensor System for the Assistance of Autonomous Vehicles and Its Evaluation". In: *arXiv.org* (2019). DOI: https://doi.org/10.48550/arXiv.1906.06789. URL: https://arxiv.org/abs/1906.06789.

[Kuo16] Kuosmanen, H. "Security Testing of WebSockets". In: *JAMK University of Applied Sciences* (2016), pp. 30–31. URL: https://www.theseus.fi/bitstream/handle/10024/113390/Harri+Kuosmanen+-+Masters+thesis+-+Security+Testing+of+WebSockets+-+Final.pdf?sequence=1.

[Nas+20] Nastjuk, I., Herrenkind, B., Marrone, M., Brendel, A. B., and Kolbe, L. M. "What drives the acceptance of autonomous driving? An investigation of acceptance factors from an end-user's perspective". In: *Technological Forecasting and Social Change* 161 (2020), p. 120319. ISSN: 0040-1625. DOI: https://doi.org/10.1016/j.techfore.2020.120319. URL: https://www.sciencedirect.com/science/article/pii/S0040162520311458.

[SXC16] Song, W., Xiong, G., and Chen, H. “Intention-Aware Autonomous Driving Decision-Making in an Uncontrolled Intersection”. In: *Mathematical Problems in Engineering* 2016 (Apr. 2016), p. 1025349. ISSN: 1024-123X. DOI: 10.1155/2016/1025349. URL: https://doi.org/10.1155/2016/1025349.

[SSH] SSH. *SSH Tunnel*. https://www.ssh.com/academy/ssh/tunneling. Accessed: 2022-02-22.

[Swi22] Swift. *Automatic Reference Counting*. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html. Accessed: 2022-03-08. 2022.