Department of Informatics
Technical University of Munich

Master's Thesis in Data Engineering and Analytics

Accident Prevention Backend Framework to Support Autonomous Driving

Backend Framework für das Vermeiden von Verkehrsunfällen zur Unterstützung des autonomen Fahrens

Supervisor	Prof. Dr.-Ing. habil. Alois C. Knoll
Advisor	Christian Creß, M.Sc. Walter Zimmer, M.Sc.
Author	Noir Nigmatov
Date	May 11, 2022 in Garching

Disclaimer

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

Garching, May 11, 2022

(Noir Nigmatov)

Abstract

The environmental perception and resulting scene and situation understanding of an autonomous vehicle are limited by the available sensor ranges and object detection performance. Even in the vicinity of the vehicle, the existence of occlusions leads to incomplete information about its environment. The resulting uncertainties pose a safety threat not only to the autonomous vehicle itself but also to the other road users. This incomplete information results in impaired driving comfort, as the vehicle must stay alert to spontaneously react to unforeseen scenarios.

Intelligent Infrastructure Systems (IIS) can alleviate these problems by providing autonomous vehicles - as well as conventional vehicles and drivers - at operating time the complementing information about each road participant and the overall traffic situation, thereby greatly extending their perception range as well.

This thesis is part of the extensive 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.

With the infrastructure provided by the IIS Providentia, this thesis project is aimed at designing and implementing a backend module that will enhance the system by providing existing road users a possibility to connect to it by the means of a mobile application over conventional cellular data networks. Besides that, the backend module should actually be a framework that is client-agnostic, implying that any client can connect to the system as long as it implements the connection and data exchange protocols. The users will have access to the digital twin of the testbed. Furthermore, different accident prevention mechanisms can be implemented on top of this module: lane change/average speed recommendations, warnings about vehicles on the entry ramp, standing vehicles, ghost drivers, jam/slowdown warnings, accident/collision warnings, etc.

Zusammenfassung

Die Umgebungswahrnehmung und das daraus resultierende Szenen- und Situationsverständnis eines autonomen Fahrzeugs sind durch die verfügbaren Sensorreichweiten und die Objekterkennungsleistung begrenzt. Auch in der Nähe des Fahrzeugs führt das Vorhandensein von Verdeckungen zu unvollständigen Informationen über seine Umgebung. Die daraus resultierenden Unsicherheiten stellen nicht nur ein Sicherheitsrisiko für das autonome Fahrzeug selbst dar, sondern auch für die anderen Verkehrsteilnehmer. Diese unvollständigen Informationen beeinträchtigen den Fahrkomfort, da das Fahrzeug wachsam bleiben muss, um spontan auf unvorhergesehene Szenarien reagieren zu können.

Intelligente Infrastruktursysteme (IIS) können diese Probleme entschärfen, indem sie autonomen Fahrzeugen - sowie konventionellen Fahrzeugen und Fahrern - zur Betriebszeit die ergänzenden Informationen über jeden Verkehrsteilnehmer und die Gesamtverkehrssituation zur Verfügung stellen und damit deren Nutzung erheblich erweitern.

Diese Arbeit ist Teil des umfangreichen Providentia++-Projekts - ein Forschungsprojekt der Technischen Universität München mit dem Ziel, den Verkehrsfluss und die Verkehrssicherheit zu verbessern, indem die Einschränkungen lokaler Sensorsysteme eines einzelnen Fahrzeugs überwunden werden.

Mit der von IIS Providentia bereitgestellten Infrastruktur zielt diese Arbeit darauf ab, ein Backend-Modul zu entwerfen und zu implementieren, das das System erweitert, indem es bestehenden Verkehrsteilnehmern die Möglichkeit bietet, sich mit einer mobilen Anwendung über die herkömmlichen Mobilfunkdatennetze zu verbinden. Abgesehen davon sollte das Backend-Modul tatsächlich ein Framework sein, das Client-agnostisch ist, was bedeutet, dass sich jeder Client mit dem System verbinden kann, solange er die Verbindungs- und Datenaustauschprotokolle implementiert. Die Nutzer haben Zugriff auf den digitalen Zwilling des Testbeds. Darüber hinaus können auf diesem Modul verschiedene Unfallverhütungsmechanismen implementiert werden:
Spurwechsel-/Durchschnittsgeschwindigkeitsempfehlungen, Warnungen vor Fahrzeugen auf der Einfahrt, stehenden Fahrzeugen, Geisterfahrern, Stau-/Verzögerungswarnungen, Unfall-/Kollisionswarnungen usw.

Contents

Chapter 1

Introduction

1.1 Motivation

The environmental perception and resulting scene and situation understanding of an autonomous vehicle are limited by the available sensor ranges and object detection performance. Even in the vicinity of the vehicle, the existence of occlusions leads to incomplete information about its environment. The resulting uncertainties pose a safety threat not only to the autonomous vehicle itself but also to the other road users. To enable it to operate safely, one of the conservative policies can be reducing the driving speed, which in turn slows down entire traffic, which causes economical and ecological impacts. Furthermore, this incomplete information results in impaired driving comfort, as the vehicle must stay alert to spontaneously react to unforeseen scenarios.

Intelligent Infrastructure Systems (IIS) can alleviate these problems by providing autonomous vehicles - as well as conventional vehicles and drivers - at operating time the complementing information about each road participant and the overall traffic situation [QA13; Men+17], thereby greatly extending their perception range as well. In particular, an IIS can observe and detect all road participants from multiple superior perspectives, with extended coverage compared to that of an individual vehicle. Then providing a vehicle with this additional information gives it a better spatial understanding of its surrounding scene and enables it to plan its maneuvers more safely and proactively. Furthermore, an IIS with the described capabilities enables a multitude of services that further support decision making.

1.2 Providentia Project

This thesis is part of the extensive 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 [Krä+19]. IIS Providentia is a research project that has been funded by the Federal Ministry of Transport and Digital Infrastructure (BMVI) in early 2017. In the beginning of 2020, it has been continued under the name Providentia++ with the Chair of Robotics, Artificial Intelligence and Real-Time Systems at the Technical University of Munich's Department of Informatics serving as the consortium leader. Additional cooperative partners supporting the project are fortiss, Valeo, Intel, Cognition Factory, Elektrobit Automotive, Huawei Technologies Deutschland, IBM Deutschland, 3D Mapping Solutions, brighter AI, Siemens, and Volkswagen.
The architecture of the Providentia++ Project, hereinafter introduced in more detail, includes both the system's hardware as well as the software to operate it. The road infrastructure of a particular testbed is enhanced with various sensors to collect traffic data. The software stack

with detection and fusion algorithms is used to generate an accurate and consistent virtual model of the testbed, called Digital Twin. In their paper, Krämmer et al. demonstrate that the provided digital twin information to an autonomous driving research vehicle can be used to extend the limits of the vehicle's perception far beyond its on-board sensors.

1.2.1 Road Infrastructure

The testbed of the project includes a stretch of the A9 Highway and an extension into the surrounding urban area of Garching to the north of Munich depicted in Figure 1.1.

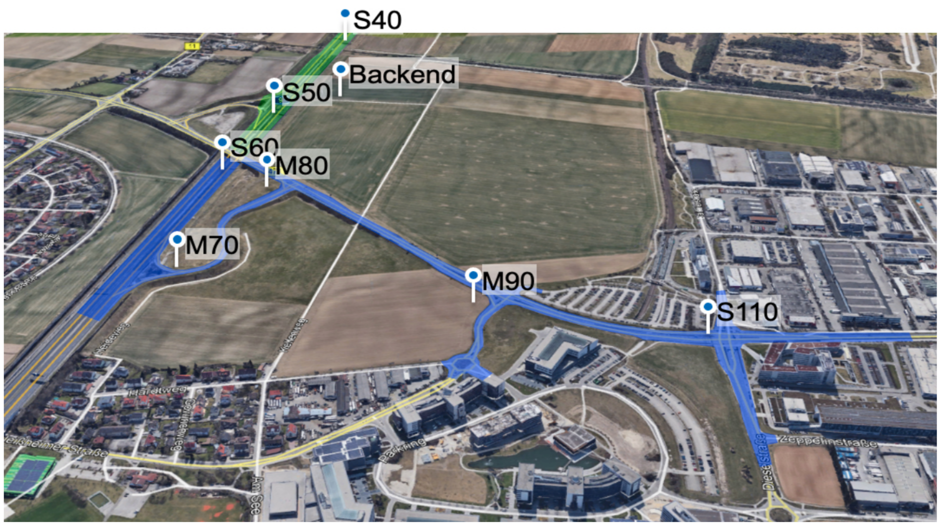


Figure 1.1: A map view of the Providentia road infrastructure, consisting of multiple measurement stations: three on a highway (S40/50/60) and four surrounding urban area of Garching (M70/80/90 and S110). The corresponding sections colored blue (Providentia++) and green (Providentia 1).

The infrastructure is a constellation of 7 sensor stations equipped with more than 60 state-of-the-art and multi-model sensors, providing a road network coverage of approximately 3.5 kilometers. Its primary purpose is to provide a real-time and reliable digital twin of the current road traffic at any given time or day of the year, for use in a variety of applications. The range of types of sensors used for measurements includes optical cameras, Light and Radio Detection and Ranging sensors (LiDAR and radar).
Each measurement point comprises eight sensors with two cameras and two radars per viewing direction. In each direction, one radar covers the right-hand side while the other covers the left-hand side of the highway.
All the sensors at a single measurement point are connected to a Data Fusion Unit (DFU), which serves as a local edge computing unit and runs with Ubuntu 16.04 Server. It is equipped with two INTEL Xeon E5-2630v4 2.2 GHz CPUs with 64 GB RAM and two NVIDIA Tesla V100 SXM2 GPUs. All sensor measurements from the cameras and radars are fed into the detection and data fusion toolchain running on this edge computing unit. This results in object lists containing all the road users tracked in the field of view (FoV) of that measurement point. Each DFU transmits this object list to a backend machine via a fibre optic network, where they are finally fused into the digital twin that covers the entire observed highway stretch. Figure 1.2 demonstrates the sensor setup of a single measurement point.

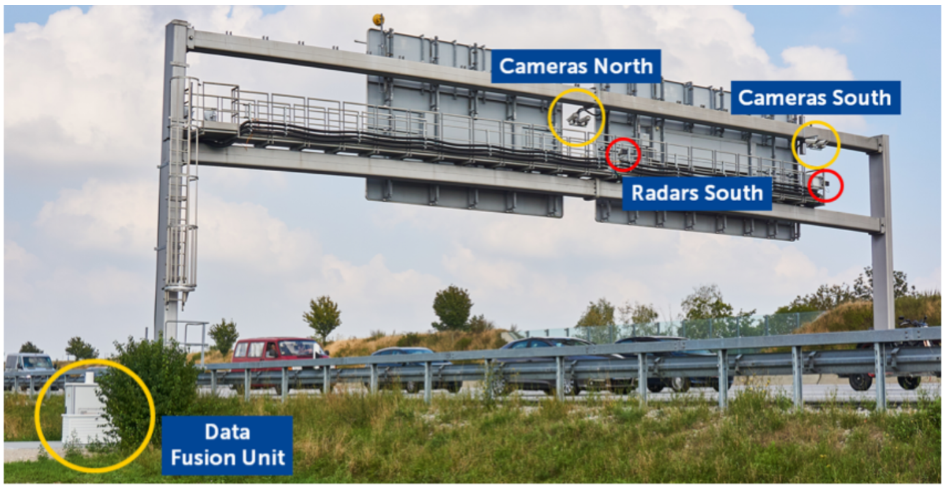


Figure 1.2: An image of a single Providentia measurement point on the A9 highway. Two optical cameras pointing in both road directions are visible (yellow circles on top). Two radars pointing to the south can also be seen (red circles). Also, a Data Fusion Unit (DFU) collecting and fusing the sensory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

1.2.2 Digital Twin

Summarizing information in the previous section, 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, 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 further research purposes and to complete and extend a vehicle's perception and to provide information that enables the implementation of various algorithms and applications based on that digital copy. The digital twin initially includes information such as position, velocity, vehicle type and a unique identifier for every observed vehicle. Figure 1.3 shows a visual example of the digital twin of traffic computed by the system.

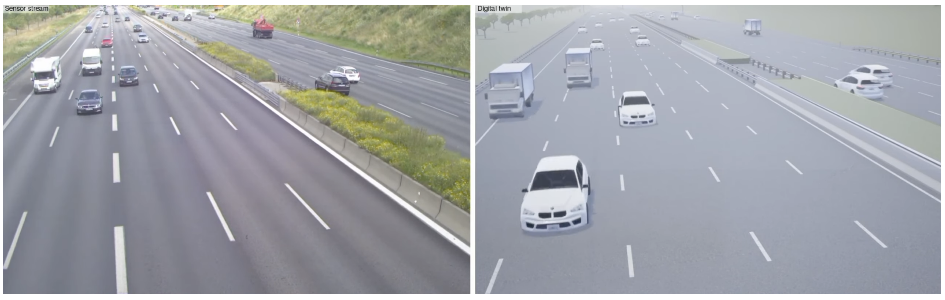


Figure 1.3: Qualitative example of how the system captures the real world (left) in a digital twin (right). The scene is recreated in CARLA driving simulator. During operation, all information is sent to the autonomous vehicle in form of a sparse object list.

As of the writing of this thesis, the live production system provided digital twin information for the stretch s40s50 (approximately 440 meters) highlighted in green in Figure 1.1. Therefore, all the work outlined and described in this paper is using digital twin modeled for that stretch only.

1.3 Project Goals

With the infrastructure provided by the IIS Providentia, this thesis project is aimed at designing and implementing a backend module that will enhance the system by providing existing road users a possibility to connect to it by the means of a mobile application over the conventional cellular data networks. Besides that, the backend module should actually be a framework that is client-agnostic, implying that any client can connect to the system as long as it implements the connection and data exchange protocols described in section 3.3.1. The users will have access to the digital twin of the testbed. Furthermore, different accident prevention mechanisms can be implemented on top of this module: lane change/average speed recommendations, warnings about vehicles on the entry ramp, standing vehicles, ghost drivers, jam/slowdown warnings, accident/collision warnings, etc.

1.4 Contributions

Thus, this work consists of two contributions:

1. Implemented fully functional backend framework that provides real-time communication server that handles multiple clients to effectively distribute relevant traffic data. Also, the system has a direct connection with the ROS backend to receive digital twin.

2. A module with a set of warnings/recommendations as a working concept of an accident prevention mechanism that is implemented on top of this backend framework. See Section 3.6 for more details.

Most of this project has been implemented in cooperation with Mohammad Naanaa's bachelor thesis *Accident Prevention Frontend Framework to Support Autonomous Driving*. He developed an iOS native application that connects to this backend according to the data exchange specifications in section 3.3.1. The mobile application uses Google Maps SDK (Software Development Kit) to visualize digital twin information directly overlaid over the actual map of the testbed [Naa22]. The module with a set of frame-level and object-level warnings (Section 3.6) was developed after the completion of Naanaa's thesis, so the mobile application does not have support for it.
Also, the backend is using scenarios implemented by Aaron Kaefer in his Master thesis *Deep Traffic Scenario Mining, Detection, Classification and Generation on the Autonomous Driving Test Stretch using the CARLA Simulator* [Aar22]. 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". The backend receives this extended digital twin output and relays the detected scenarios in the additional object field called `scenarios` (Table 3.3) to the clients to generate appropriate warnings directly on the frontend.

The design and implementation of the system is described in great detail in the following chapters.

Chapter 2

Related Work

This chapter reveals and elaborates on works related to the project. The works include research papers, projects, and review articles, all of which have had the most influence and contribution in various aspects: from conceptual perception and problem understanding to overall architecture design and implementation.

ITS. First ideas and concepts for assisting vehicles, as well as monitoring and managing road traffic in the form of an Intelligent Transportation System (ITS) have been formalized in the PATH [Shl92] and PROMETHEUS [BR95]. The California PATH Program was established in 1986 by the Institute of Transportation Studies of the University of California at Berkeley, under the sponsorship of the California Department of Transportation (Caltrans). This was the first research program in North America focused on the subject of ITS. Its mission was to conduct the research and development work needed to establish the foundation for applying advanced technologies to improve the operation of the state's transportation system. Similarly, in the same year, the Programme for a European Traffic with Highest Efficiency and Unlimited Safety (PROMETHEUS) was established. It is a research program to elaborate the technical base for advance in the development of road transport. The strategic objective is to create concepts and solutions which will make traffic perceptibly safer, more economical, with less impacts on environment, and thus render the traffic system more efficient. Both research projects conceptualized the architecture of ITS with intelligent vehicles equipped with radars, vision cameras, GPS receivers, and onboard computers to process the sensor measurements. Communication is via WLAN using mobile ad hoc networks between vehicles and roadside routers. Such ITS, on one end, enables the vehicle to be driven safely by means of "electronic sight" which increases the perception area of the driver, and on the other end, it enables higher level traffic management for better efficiency.

IIS. Recently, with the growing efforts of industry and academia to realize autonomous driving, the need for intelligent infrastructure systems (IIS) that are able to support autonomous vehicles has further increased. Several new projects have been initiated, and Providentia is one of them. However, the focuses of the projects differ widely in scope.

Some IIS projects primarily focus on the communication aspects between the vehicle and infrastructure, and sometimes additionally on the vehicle-to-vehicle communication. The research project DIGINET-PS [Ber] by the Technical University of Berlin focuses on the communication of traffic signals, occupancy of parking spaces, traffic queues, weather and road conditions, environment measurements for air quality. For real-time communication roadside routers with the ETSI ITS-G5 standard based on the IEEE 802.11p WLAN [ETSa] are used to directly transmit raw information to the vehicle. Intelligent vehicles are equipped with a selection of different sensors in the form of cameras, radar, LiDAR, GPS, as well as actuators for lateral and longitudinal guidance, signalization control, vehicle gear control and pedal

level transmitter. A powerful on-board computer is responsible for the rapid processing of all the information. For this purpose, a software stack is provided. The stack brings together all collected information about the current situation, derives predictions of the future states, plans suitable driving maneuvers and controls the vehicle accordingly. This costly hardware and software extension greatly limits the practical aspect of the system.
Similarly, the Antwerp Smart Highway [Ant18] project focuses on vehicle-to-everything (V2X) communication and distributed edge computing. V2X communication (incorporates vehicle-to-vehicle and vehicle-to-infrastructure) entails the use of the Collective Perception Service which enables vehicles to share information about other road users and obstacles that were detected by local perception sensors such as radars, cameras and alike. In that sense, it aims at increasing awareness between vehicles by mutually contributing information about their perceived objects to the individual knowledge base of the vehicle by disseminating Collective Perception Messages (CPM) [ETSb]. The test site consists of a highway strip of 4 km equipped with road side communication units (RSUs). The RSUs are connected to the fiber network of the road operator, able to fetch information about electronic traffic signs on the highway. The data is transmitted using the ETSI ITS-G5 over WLAN technology to the specially equipped car.
The New York City Connected Vehicle Project [DOT15] is much greater in scope and practical application. The primary goal of the project is to eliminate the traffic related deaths and reduce crash related injuries and damage to both the vehicles and infrastructure. The project deployment is primarily focused on safety applications – which rely on vehicle-to-vehicle, vehicle-to-infrastructure and infrastructure-to-pedestrian communications. These applications provide drivers with alerts so that the driver can take action to avoid a crash or reduce the severity of injuries or damage to vehicles and infrastructure. Similarly to the previous projects, the dedicated short-range communication protocol on top of WLAN is used for data transmission among road participants and RSUs. However, the infrastructure-to-pedestrian communication is done over cellular networks with pedestrians using the special mobile application installable on regular smartphones. The main functionality of the mobile app is to send the pedestrian presence information to the system, which in turn broadcasts it to the vehicles approaching the crosswalk.

Other IIS projects, similar to Providentia, focus on roadside perception. The paper by Fleck et al. [Fle+18] presents the concept, realization and evaluation of a flexible and scalable setup for smart infrastructure at the example of the Test Area Autonomous Driving Baden-Württemberg [Ver]. In particular, the system is perceiving a cross-road with two high resolution cameras and creates a digital twin. However, an autonomous vehicle with the onboard hardware (numerous cameras, sensors, GPS antennas, and laser scanners) and complex software bears the bulk of the computation and evaluation of the information. The deployed roadside system is much smaller than Providentia infrastructure and cannot operate at night as it only uses cameras.
Another example of such an IIS is the test field of 1.2 km on the motorway A2 near Graz, Austria, operated by ASFINAG, the Austrian motorway operator. Seebacher at al. in their work *Infrastructure data fusion for validation and future enhancements of autonomous vehicles' perception on Austrian motorways* [See+19] focus primarily on post-validation of driving maneuvers by comparing an autonomous vehicle's on-board sensor data with the traffic data captured by roadside infrastructure sensors. However, the application outlook described in the paper involves future real-time enhancement of autonomous vehicle's perception range.

Application server. An article *Getting Started with Building Realtime API Infrastructure* [Bak17] describes rather an abstract model of the high-level conceptual design of a real-time application server. The author focuses on the true essence of the real-time systems. Specifically, he argues that realtime does not necessarily mean that something is updated instantly, as there is no universal practical definition of "instantly". But realtime is about pushing data as fast as possible. Thus, the author accentuates the focus of building realtime systems on the mechanism of pushing the changes. Another article, *Building Scalable Web Application for Your Project: Best Principles and Practices* [KS21], gives an extensive overview of principles and guidelines in building scalable web application from different perspectives.

Network communication. A recent article by Eduardo Ribeiro [Rib21] gives an overview of network protocols to use in the server-client model. The article explains each protocol with a detailed description of how it operates on the request, response, and message levels. It provides both advantages and disadvantages based on the use cases. Whereas, a more scholarly article by Ogundeyi K.E. [Ogu19] specifically advocates for the use of WebSockets for real-time communication by providing sufficient technical comparison with other protocols such as HTTP and SSE(Server-Sent Events).

Chapter 3

Implementation

3.1 Conceptual Approach

Providentia's digital twin is implemented in ROS1 Noetic (A.1). The general approach is to create a backend server that communicates both with the digital twin to get data, and the mobile applications to send the data. Figure 3.1 gives schematic view of this concept. The backend system should provide seamless, fast, real-time integration with the Providentia's digital twin. Furthermore, the application server should make real-time connections with the clients to relay the relevant information quickly. The backend should have a public static IP address to allow mobile devices to directly communicate with it over the Internet. Accordingly, mobile endpoints are connected to the Internet through the wide-area communication technologies such as 3G, 4G, and 5G provided by cellular carriers. Each mobile device constantly shares its GPS location with the backend server and in turn receives the traffic information specific to the location. Precise data exchange format specification (3.3.1) designed at the start of the project serves as the building ground for the entire communication protocol. The mobile platform can be of any type. The iOS mobile application specifically designed by Mohammad Naanaa was used during the development and testing phases.

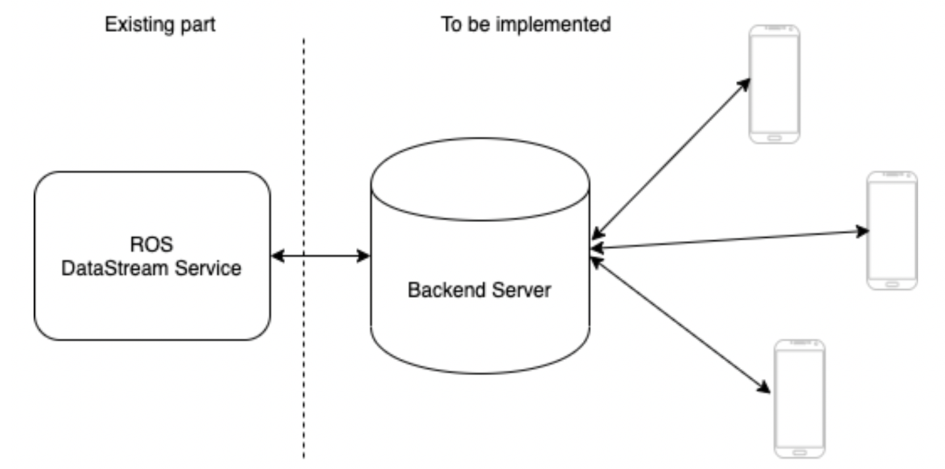


Figure 3.1: Module architecture

In a nutshell, the design of the backend system should address three following aspects:

1. Connection to the ROS backend to receive digital twin;
2. Network protocol for frontend-backend connection;

3. Distribution of relevant data.

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

3.2 General Application Server Requirements

3.2.1 Connection to the ROS Backend

So, first of all, the analysis of the Providentia's digital twin outbound interface is performed.

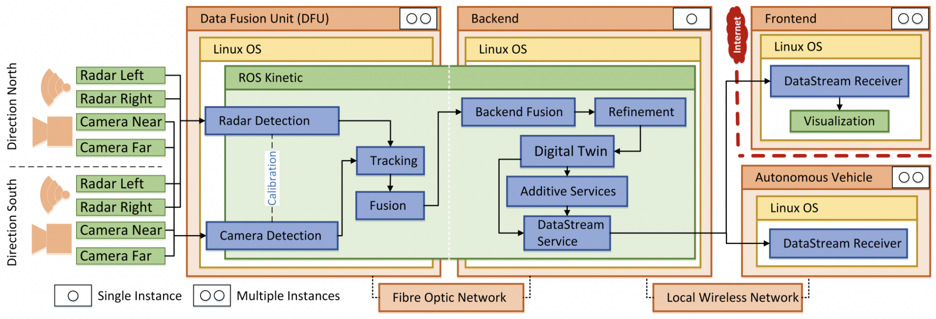


Figure 3.2: Platform architecture of the Providentia system. Three distinct components (DFU, Backend, and Frontend/Autonomous Vehicle) can be seen with the inter-component data flow. A DFU is connected to sensors on the left and processes the raw data. The data is used to construct a digital twin. Additive Services are then provided using that twin. These services are offered to the Frontend ([Krä+19]).

Figure 3.2 displays the platform architecture of the Providentia system. The software architecture involves three major components: Data Fusion Unit (DFU), Backend (hereafter "ROS backend"), and Frontend (hereafter "backend" for mobile applications), on the creation on which this work is focused.

A DFU is responsible for running local sensors collecting raw data, performing sensor data fusion, object detection and classification. The processed data is then pipelined further into the ROS backend.

Each DFU is responsible for one road section (e.g., s40s50) and does the processing independently of other DFUs. Thus, it enables multiple DFUs to asynchronously publish their local traffic data to the ROS backend, making the process scalable for an industrial application with numerous DFUs that allow the Providentia system to monitor many road sections simultaneously.

The ROS backend does final global fusion of the traffic data, refinement to finally produce the digital twin information in the form of sparse list of objects for each road section. Since ROS is used on all computing units, the digital twin for s40s50 stretch is published as the ROS topic */s40/s50/tracker/estimates/throttled*. Thus, the ultimate consumption of the list of objects is accomplished by creating a ROS node and subscribing to the ROS topic. The rate at which the messages are published to the topic is approximately 25 Hz.

3.2.2 Web interfacing capability

Apparently, the backend for mobile applications should be able to allow and sustain web connections from the clients. The only network technology that is widespread and easily available to the mobile applications is the Internet through the wide-area communication technologies such as 3G, 4G, and 5G provided by cellular carriers. Therefore, the backend should additionally contain a web server to make interaction between a user and backend software running on the application server possible.

3.2.3 Comparison and Selection of a Technological Framework

The right backend technology goes a long way in enhancing an app development project. An excellent will enhance development speed, increase app responsiveness, and give room for scaling when the need arises.

Apparently, the backend server can be written in any server-side programming language, the most used ones among which are server-side JavaScript, Python, Java, and C++. Nowadays, all of these languages also support web interfaces. The actual selection of the target framework depends on several factors. Considering the aspects described in previous sections, the factors that heavily affect the choice can be summarized in the following list, ordered by priority:

1. **Availability of the ROS client library**. This is the most important point to consider, for the development of such library using ROS API is out of scope of this work and would require a lot of time.

2. **Programming model.** There are basically two types of programming models: event-driven (or asynchronous) and sequential. Two different methods to support two different needs. In this project, when the backend receives a message from ROS, it should process it immediately and relay further to the clients. Likewise, when the backend receives GPS locations from the clients, it should react to this event by processing the information without any delay. Clearly, the event-driven paradigm is more appropriate in such case as all the processing (application logic) is bound to external events. Moreover, the event-driven programming model is the best fit to create real-time web applications (Figure 3.3). Sequential programming is more often found in batch processing.

3. **Strong community.** Strong community of developers behind any open-source technology defines its maturity, popularity and widespread use. There is abundant documentation online and applications written in such languages can be easily maintained in the future. Moreover, these languages have a lot of developed add-on modules that can be easily used to handle various core functionalities.

4. **Fast prototyping.** The language and the framework inside which it runs should allow for fast prototyping and development of applications. Low level languages, e.g., C and Go, require too much effort to learn and also to produce the code.

The following analysis includes a few server-side technologies with the most potential to sustain real-time communication.

C++ It is rather a low-level programming language with the support of object-oriented paradigm. Being an extension of C, it is extremely fast in execution. There is a ROS library client *roscpp* (A.3). Originally C++ supports sequential programming model. However, there exist Boost libraries (A.2) that support event-driven style for managing real-time communication through the web interface. C++ is one of the oldest robust programming languages. There is a strong community of peer developers behind it. One major disadvantage though, is that it is not suitable for fast prototyping. The development in C++ is quite long and expensive.

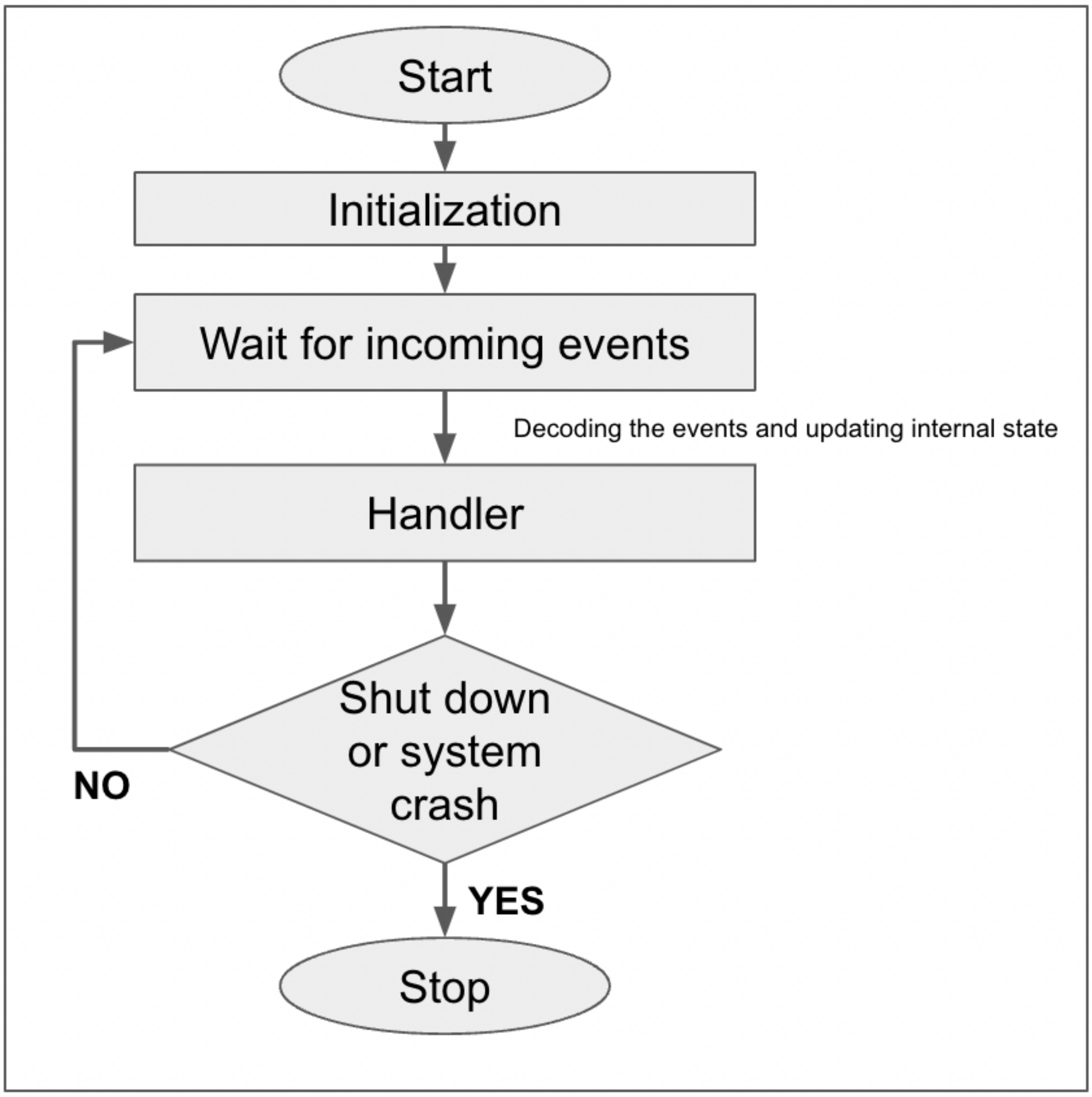

Start
Initialization
Wait for incoming events
Decoding the events and updating internal state
Handler
Shut down or system crash
NO
YES
Stop

Figure 3.3: Event-driven architecture. Event-driven programming depends upon an event loop that is always listening for the new incoming events [Tei12].

Java There is a ROS library client *rosjava* (A.4). Java's support of the event-driven programming model depends on the high-end framework used to create the application. Java is quite old, it was released to public in 1995. It is the most popular server programming language in the corporate world due to various open-source and proprietary frameworks with in-built security features. The major disadvantage of using this language, is that it needs high-end systems to run excellently, making it expensive to implement Java backend. Moreover, writing Java programs can be time-consuming compared to other languages. This is mostly due to the complexity of Java application servers with no unified specification. So, first additional step is to learn the application framework which can be quite costly in terms of time.

Python Python is a popular, multi-purpose programming language developed in 1991. It offers a simple and easy-to-use backend. There is a ROS client library *rospy* (A.5). The event-driven programming is done through the use of *Asyncio* module, which provides infrastructure for writing single-threaded concurrent code using co-routines (A.6). Python is completely open-source and has a large community of developers. It is suitable for fast development, however, it is required to learn the *Asyncio* module first, since pure Python fits the sequential programming model.

Node.js It is an open-source, cross-platform, back-end JavaScript runtime environment. There is a ROS client library *rosnodejs* (A.7). Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time web applications. Although, Node.js is relatively young, its initial release was in 2009, there is a strong industrial and community support for it. Documentation is abundant, as well as, there are thousands of open-source libraries for Node.js, most of which are hosted on the *npm* website (A.8). Node.js brings event-driven programming to web servers, enabling development of fast web servers in JavaScript [Tei12]. Due to the simplicity of JavaScript syntax and easy deployment of Node.js, this technology is ideal for fast prototyping and development.

So, the obvious choice is Node.js. It is primarily used to build network programs such as Web servers. The most significant difference between Node.js and C++ and Java is that most functions in the latter two block until completion (commands execute only after previous commands finish), while Node.js functions are non-blocking (commands execute concurrently and use callbacks to signal completion or failure). Technically, non-blocking programming is actually possible in inherently blocking C++ and Java, but it requires additional in-depth knowledge of low-level functionality and coding skills, making it practically impossible for this project.
Although, Python and Node.js are somewhat similar candidates, the main advantage of the latter is that its event-loop does not need to be called explicitly (unlike when using *Asyncio* module in Python). Instead, callbacks are defined, and the server automatically enters the event loop at the end of the callback definition. Node.js exits the event loop when there are no further callbacks to be performed. Thus, the code for Node.js has more simplistic style, provides better readability, and can be easily maintained in the future.
In conclusion, Node.js operates on a single-thread event loop, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context switching [Tei12].

3.3 Frontend-Backend connection requirements

3.3.1 Data Exchange

For scalability purposes, a mobile app user should dynamically receive only the relevant part of the traffic data for its current position. This technique limits the amount of data that has to be received by the client and therefore improves both backend's and mobile device's network resources consumption. However, for this approach to work, a notion of data relevance has to be defined.

In this work, the relevance of the data is determined based on the principle of locality. This principle requires a bidirectional exchange between the backend and frontend with a well-defined specification outlined and described in the following two sections. The chosen format of the messages is JSON. Important to note, this specification defines the minimum, i.e. the mandatory fields for the exchange, as the entire functionality of the backend and frontend application software depends on this information. Due to the used JSON format, extensions of messages are possible with backwards compatibility.

Received Data Specification

For the backend to decide which traffic data is relevant for each user, the frontend has to reveal 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 required to be sent to allow for additional processing, especially in the case of client-specific warnings described in section 3.6. The complete message specification from the frontend's side is listed in Table 3.1

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

Table 3.1: Received data specification from the device to the backend

Transmitted Data Specification

The backend constantly receives (approximate rate is 25 Hz) the digital twin data in the form of sparse object list, i.e. all the detected and classified objects with additional attributes for each road section. Currently, the testbed consists of just one stretch *s40s50*, and the ROS topic is */s40/s50/tracker/estimates/throttled*. Each received message is of type *BackendOutput* (A.10), which contains a list of objects of type *DetectedObject* (A.11). Thus, the backend mostly mirrors the data received in the digital twin, and also adds additional metadata for the mobile app.

Upon receiving user's GPS data, the server stores it in the user's attributes. When the next event, i.e. receiving of the frame message from the ROS, happens, the backend server then decides what part of the traffic data is relevant based on each user's location and sends the corresponding data to each client.

The transmitted frame message contains a list of detected objects of type `Vehicle` and additional fields for metadata and the frame-level warnings and recommendations: recommended speed and weather condition (see Section 3.6). Table 3.2 contains the full specification of the message.

Name	Type	Description
msg_type	String	Message type. The value is 'frame'.
seq	String	Unique sequence identifier coming from the ROS Header type
timestamp_secs	String	The timestamp of this message, originating from the ROS, number of seconds
timestamp_nsecs	String	The timestamp of this message, originating from the ROS, remaining number of nanoseconds
timestamp_full	String	The timestamp of this message, originating from the ROS, formatted as secs.nsecs
num_detected	String	Number of detected objects in the list of objects
objects	[Vehicle]	The list of all detected Vehicle objects
weather_condition	String	Values: rainy, foggy, snowy, cloudy, sunny
speed_rec	String	Recommended speed based on the average speed of the vehicles in the list
warnings	String	A string that contains different frame-level warnings separated by a semicolon

Table 3.2: Transmitted frame message specification from the backend to the frontend

Each `Vehicle` from the aforementioned list also holds its unqiue global `id`, a vehicle's `category`, `position`, `speed`, `shape`, and `scenarios` (see Table 3.3).

Name	Type	Description
id	String	Unique id of the object.
category	String	The category of a vehicle. Possible values: {'bus','car','truck','motorcycle','pedestrian','special_vehicle'}
position	Position	The position of the object as a 3-dimensional vector
speed	Speed	The speed of the object as a 3-dimensional vector
shape	Shape	The shape of the object as a 3-dimensional vector
scenarios	Scenarios	A container storing various risk-scenarios associated with the object

Table 3.3: Specification of the `Vehicle` type

The corresponding vehicle's `position`, `speed`, `shape`, and `scenarios` are also data structures holding specific information. The `position` field contains the GPS position of a vehicle - latitude and longitude, described in Table 3.4.

The `speed` field stores the speed of a vehicle in two dimensions - latitudinal and longitu-

Name	Type	Description
position.x	String	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	String	The latitude of a vehicle in degrees with positive values extending north of the equator and negative values extending south of the equator
position.z	String	The altitude of a vehicle - Unused field set to 0

Table 3.4: Specification of the `Position` type

dinal - and the derived heading direction, described in Table 3.5. The magnitude of the first two dimensions gives the absolute speed value in `m/s`.

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

Table 3.5: Specification of the `Speed` type

The `shape` field stores the shape of a vehicle in three dimensions: width, length, and height, described in Table 3.6.

Name	Type	Description
length	String	Length of a rectangle corresponding to a vehicle
width	String	Width of a rectangle corresponding to a vehicle
height	String	Height of a rectangle corresponding to a vehicle

Table 3.6: Specification of the `Shape` type. **Note**: the values might be set to zero if computation is impossible, i.e. in bad weather conditions or darkness

The `scenarios` field is an extension for a vehicle storing information for all detectable hazardous scenarios associated with a vehicle (see Aaron Kaefer's Master thesis *Deep Traffic Scenario Mining, Detection, Classification and Generation on the Autonomous Driving Test Stretch using the CARLA Simulator*). Table 3.7 describes the `Scenarios` type.

3.3.2 Comparison and Selection of a Network Protocol

Following the described data exchange specification, the backend server and a mobile application therefore form a bidirectional communication channel through the Internet, as both constantly send its corresponding messages to each other.

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

- **Bidirectionality**
- **Low latency real-time**

Name	Type	Description
lane_id	String	Detected lane id of a vehicle
lane_change_left	String	This vehicle is changing its lane to the left
lane_change_right	String	This vehicle is changing its lane to the right
cut_in_left	String	This vehicle cuts into a lane to the left
cut_in_right	String	This vehicle cuts into a lane to the right
cut_out_left	String	This vehicle cuts out from a lane to the left
cut_out_right	String	This vehicle cuts out from a lane to the right
tail_gate_level	String	This vehicle is tailgating. Severity $\in \{0, 1, 2, 3\}$
speeding	String	This vehicle is exceeding the speed limit
standing	String	This vehicle is standing
wrong_way	String	This vehicle is driving in a wrong way

Table 3.7: Specification of the `Scenarios` type. Boolean values: '`True`' or '`False`'

- **Good security model**

Low latency real-time connection is essential - traffic data is expected to be transmitted very frequently. The data publishing frequency of each DFU is approximately 25 Hz, i.e. around every 40 ms the ROS backend publishes new frame message for each road section. Moreover, as the vehicle moves on the highway with no official speed limit, the location information sent to the backend should reach it quickly, to ensure the server has the most accurate position information of a vehicle. This is necessary for the correct traffic data to be delivered to the device incurring no stale data.

A further concern for the data transfer is security. The frontend will send user's GPS position, while the backend will send the GPS position of all vehicles of a certain part of traffic - both representing location data. Location data is considered a sensitive data subject to protection under European Union's *General Data Protection Regulation (GDPR)*, and hence regulation-conformant handling of the data is required [Com18].

With the aforementioned requirements, protocol candidates has to be considered on two distinguished layers: the low-level *Data Transport Layer* - layer 4 in the OSI model (A.12), and the more abstract *Application Layer* - layer 7.

Data Transport Layer

For the underlying data transport layer, the selection of a protocol is straightforward. As both connection endpoints are connected to the Internet, the underlying network protocol was chosen to be the connection-oriented `TCP` as it provides reliable, ordered, and error-checked delivery of a byte stream between applications running on hosts communicating via an IP network. Another alternative is connectionless `UDP`. However, it has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network; there is no guarantee of delivery, ordering, or duplicate protection.

Application Layer

At this level, there are basically 3 options to consider for the final network protocol to be used in the project. Since this is the last layer in the OSI model (A.12), the direct exposure to the programming API's of the lower levels allows to actually develop own protocol to suit specific needs. However, custom development of such protocol is out of scope of this project, and would require a lot of time. Important to note, at this stage of the project, there aren't any specific needs to be addressed for the communication between the backend and the mobile application.

REST API over HTTP REST, i.e. Representational State Transfer, defines a set of constraints to be utilized for creating web services. It is one of the architectural styles to create REST endpoints using HTTP in a web application. RESTful endpoints are being called, which would invoke APIs that are RESTful in nature and give an HTTP response (A.13). This is the classic client-server communication model that fits most of the cases. It is largely used in the server-browser setup. However, in real-time applications it is already assumed that the information is needed from the server as soon as it becomes available - and, fundamentally, the classic HTTP request/response paradigm isn't up to the job. That's because the server will be silent, new data or not, unless or until a consumer requests an update.

That limitation saw the emergence of all manner of hacks and workarounds as developers sought to adapt that request/response model to the demands of a more dynamic, real-time web – some of which became formalized and pretty widely adopted.

All these technologies and approaches – from Comet (A.14) to HTTP long polling – have one thing in common: Essentially, they set out to create the illusion of truly real-time (event-driven) data exchange/communication, so when the server has some new data, it sends a response. Figure 3.4 shows how it works.

Even though HTTP is not an event-driven protocol, so is not truly real-time, these approaches actually work quite well in specific use cases, Gmail chat for instance. Problems emerge, however, in low-latency applications or at scale, mainly because of the processing demands associated with HTTP.

That is, with HTTP you have to continuously request updates (and get a response back), which is very resource-intensive: a client establishes a connection, requests an update, gets a response from the server, then closes the connection. Imagine this process being repeated endlessly, by thousands of concurrent users – it's incredibly taxing on the server at scale.

Aggressive polling keeps the app responsive, but leads to larger server resource utilization. Any bugs in the polling frequency result in significant backend load and degradation. As the number of features with real-time dynamic data needs might increase in future, this approach will prove infeasible as it would continue to add significant load on the backend.

Consequently, polling leads to faster battery drain, app sluggishness, and network-level congestion. This is especially evident in places with 2G/3G networks or spotty networks across the terrain where the app might retry multiple times for each polling attempt.

Thus, this technique fails at both bidirectionality and low latency real-time constraints: as the HTTP works only in one way - from client to the server and the connection overhead is huge considering that for every request/response pair there is a new connection to be established.

Robust security can be achieved by using secure version of HTTP - HTTPS (A.17).

Server-sent events (SSE) SSE is server push technology (A.15) enabling a client to receive automatic updates from a server via an HTTP connection, and describes how servers can initiate data transmission towards clients once an initial client connection has been established. They are commonly used to send message updates or continuous data streams to a browser client and designed to enhance native, cross-browser streaming

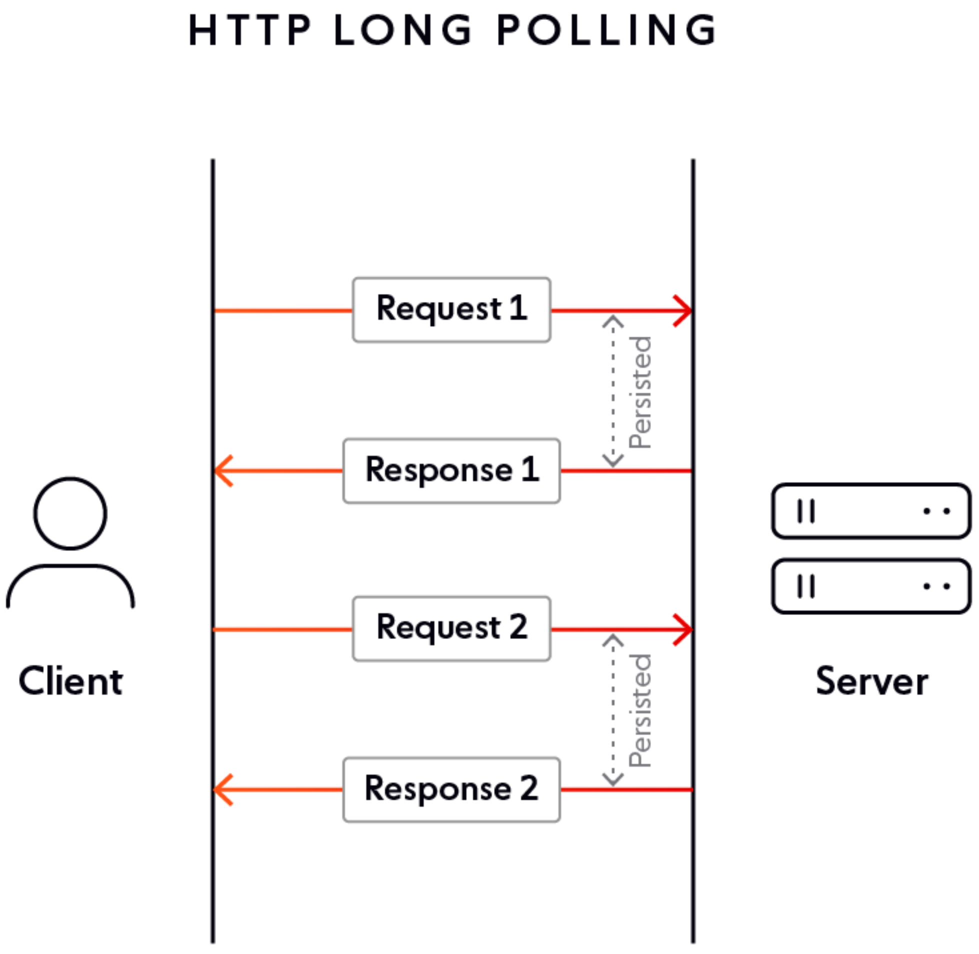


Figure 3.4: Long polling using HTTP diagram

through a JavaScript API called EventSource (A.16), through which a client requests a particular URL in order to receive an event stream.

This technology essentially removes the overhead associated with constant polling for updates and improves latency, as persistent connection is established once. However, SSE is a mono-directional protocol. After doing the initial request/response step, the server is then able to push messages to the client. But, the client cannot send any information to the server, except initiating a new request. Thus, there is no way for the client to efficiently send its location data to the backend on a periodic basis. If new requests to be made on each client position update, then it basically boils down to the same REST API over HTTP model. Figure 3.5 shows the underlying behavior of SSE.

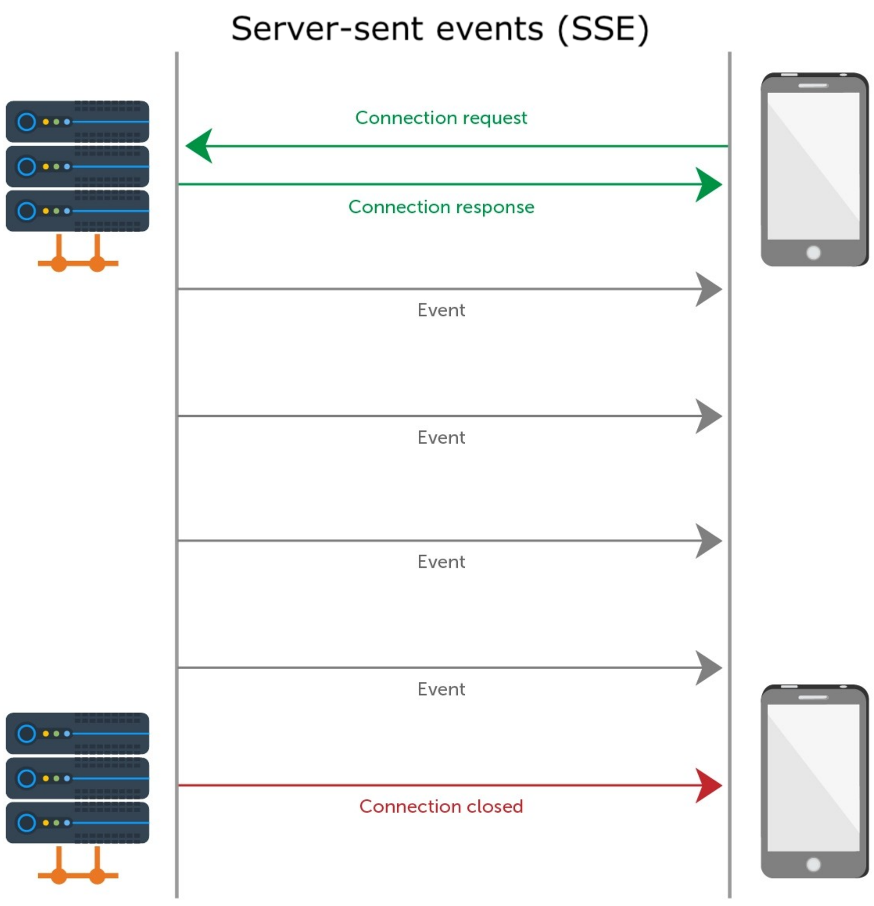


Figure 3.5: SSE diagram

WebSocket Around the middle of 2008, the limitations of using continuous two-way server/ browser interaction system Comet were being felt particularly keenly by developers Michael Carter and Ian Hickson. Through collaboration on IRC and W3C mailing lists, they hatched a plan to introduce a new standard for modern real-time on the web via bi-directional communication, thus creating the name “WebSocket”. The idea made its way into the W3C HTML draft standard and, shortly after, the first version of the protocol was introduced.

The WebSocket protocol enables full-duplex interaction between a web browser (or other client application) and a web server with lower overhead than half-duplex alternatives such as HTTP polling, facilitating real-time data transfer from and to the server.

This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. Figure 3.6 shows this process.

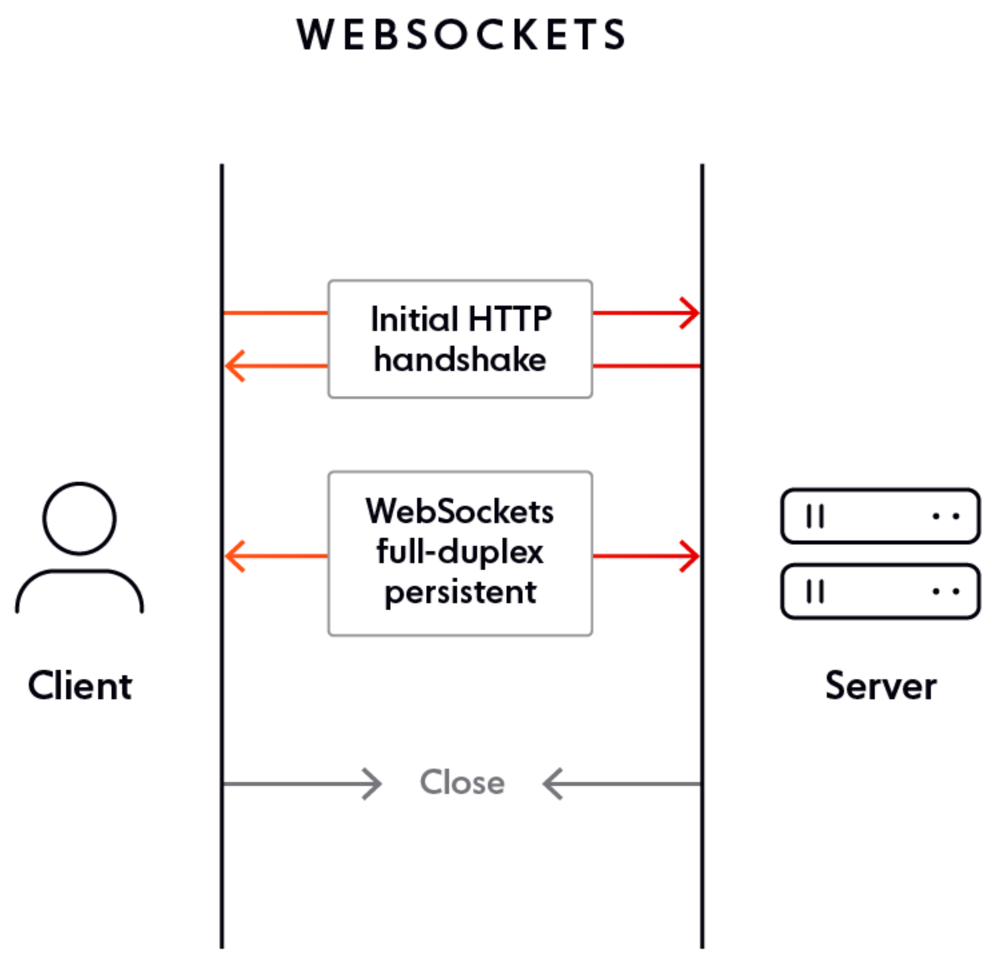


Figure 3.6: WebSocket diagram

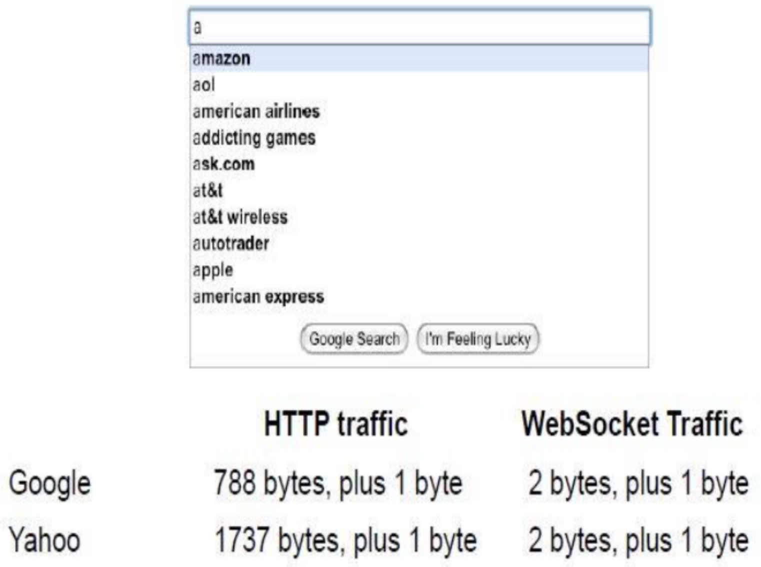
Moreover, WebSocket is designed to work over HTTP ports 443 and 80 as well as to support HTTP proxies and intermediaries, thus making it compatible with HTTP. To achieve compatibility, the WebSocket handshake uses the HTTP Upgrade header to change from the HTTP protocol to the WebSocket protocol.

Although WebSocket is relatively new technology, the protocol was standardized by the Internet Engineering Task Force (IETF) as RFC6455 ([MF11]) in 2011. Since then the protocol has been implemented in all browsers, and all server-side application frameworks fully support the specification. In addition, mobile platforms have introduced full native support of the protocol ([Dia20b], [Dia20a]).

Similar to HTTP, WebSocket has a secure version of it called WebSocket Secure.

Obviously, the WebSocket is the most suitable option to go with the network protocol between the backend and the mobile application. To summarize:

- Websockets are event-driven (unlike HTTP). Arguably, event-driven is a prerequisite for true real-time.
- Full-duplex asynchronous messaging. In other words, both the client and the server can stream messages to each other anytime independently from each other.
- WebSocket keeps a single, persistent connection open while eliminating latency problems that arise with HTTP request/response-based methods due to the overhead. Figure 3.7 shows the comparison of outbound traffic bandwidth for websockets and HTTP [Ogu19].


a
amazon
aol
american airlines
addicting games
ask.com
at&t
at&t wireless
autotrader
apple
american express
Google Search
I'm Feeling Lucky

	HTTP traffic	WebSocket Traffic
Google	788 bytes, plus 1 byte	2 bytes, plus 1 byte
Yahoo	1737 bytes, plus 1 byte	2 bytes, plus 1 byte

Figure 3.7: Total information sent for each character entered into the search bar.

3.3.3 Security

Admittedly, WebSocket protocol is known to have some serious considerations regarding the security aspect. As opposed to the HTTP requests, WebSocket requests are not restricted by the same origin policy, i.e. a vanilla implementation of a WebSocket server could easily expose a vulnerability to cross-site hijacking attacks [Kuo16]. To improve the security aspect and mitigate that risk, as well as to protect private data of the clients (as per GDPR regulation), three security-improving decisions were made:

- Switching to the secure version of the WebSocket protocol, called WebSocket Secure (wss)
- Safe parsing of the client data using `JSON.parse()`
- Validation of user message against the specification outlined in Table 3.1. Connections are dropped immediately, if the validation fails, and also when the location data is not received in the first 2 seconds after establishing a connection.

The secured `wss` (WebSockets over SSL/TLS (A.18)) version of the protocol offers two benefits for the overall security: it encrypts the data between the frontend and the backend, preventing capturing or tampering with the sensitive data in the middle (man-in-the-middle attack (A.19)), and it avoids issues with websockets on networks that employ so-called intermediaries (proxies, caches, firewalls). The latter is especially relevant for the mobile operator networks, which are expected to be the primary network source for clients.

However, because for the encryption of data, wss needs a valid certificate issued by a trusted third-party called certificate authority (CA), for testing purposes during the development it suffices to either use self-signed certificates or use an SSH tunnel (port 22) with enabled port forwarding between the two endpoints, the server and the mobile client. Obviously, in such a case, because the application server is directly connected to the production ROS backend, all the network ports except 22 are disabled. Additionally, only key based logins for a set of predefined public keys is allowed. Thus, utilizing SSH tunneling addresses the concern of WebSocket's vulnerability - after a secure (encrypted) tunnel between two endpoints is established, the server is accepting data only coming from it, eliminating a possibility of cross-origin attacks. This final connection setup is depicted in Figure 3.8

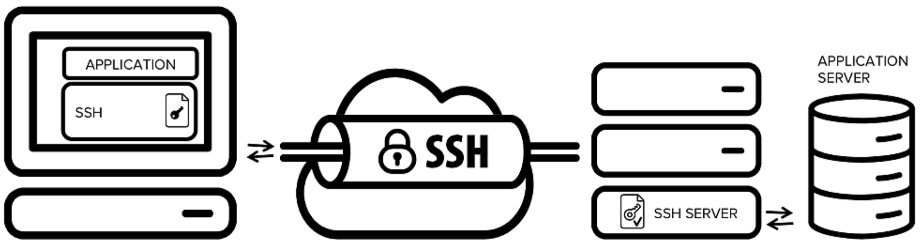


Figure 3.8: A visual representation of the connection between the frontend (seen here on the left side) and backend (seen here on the right side consisting of two abstract parts: SSH handler and the actual application server)

3.4 Distribution of Data

With the right technology and protocols in place, it is now imperative to discuss the core logic of the application server - data distribution. So, on one end, the server receives data from the ROS backend, and on the other it distributes this data to the clients through low latency real-time websocket connections. The main concern is now to efficiently distribute relevant traffic data to all the connected clients. Two different techniques were implemented: brute-force looping and pub/sub model (A.21).

3.4.1 Brute-Force Approach

This approach employs the simplest looping method in its algorithm:

1. A frame message is received from the ROS backend
2. The message handler loops through the object list, and for every client connected to the system, checks if the object is close enough to its last known location. Thus, for every client it forms a temporary list of relevant objects. Any client, which connects to the

system in the middle of this process, starts receiving the updates from the next frame message processing cycle.

3. With the temporary non-empty lists defined, the backend forms the outgoing messages to the corresponding clients and actually sends the information.

The proximity of an object and a vehicle is checked using the radius search - if the object falls inside the radius (configurable parameter) of a circle centered at the vehicle's location, then it is sent to that vehicle as a detected object. Radius search in turn uses the `Haversine` formula to calculate the great-circle distance between two points – that is, the shortest distance over the earth's surface – giving an 'as-the-crow-flies' distance between the points (ignoring any hills they fly over):

$$d = 2 * R * \arcsin \sqrt{\sin^2(\frac{\varphi_2 - \varphi_1}{2}) + \cos\varphi_1 * \cos\varphi_2 * \sin^2(\frac{\lambda_2 - \lambda_1}{2})}$$

where φ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371,000 meters).

The JavaScript implementation:

```
const R = 6371e3; // metres
const φ1 = lat1 * Math.PI/180; // φ, λ in radians
const φ2 = lat2 * Math.PI/180;
const Δφ = (lat2-lat1) * Math.PI/180;
const Δλ = (lon2-lon1) * Math.PI/180;

const a = Math.sin(Δφ/2) * Math.sin(Δφ/2) +Math.cos(φ1) * Math.cos(φ2) *
Math.sin(Δλ/2) * Math.sin(Δλ/2);

const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
const d = R * c; // in metres
```

Important to note, the `Haversine` formula remains particularly well-conditioned for numerical computation even at small distances.

As with any brute-force algorithm which simply loops over all the data and variables, this approach fails to generalize well when the size of data to consider scales up. This technique definitely works fine in the testing scenario: one road section *s40s50* and 1-2 clients connected to the system. However, brute-force looping will fall short in future, as it represents a single centralized computing unit. A different horizontally scalable data distribution algorithm is needed.

3.4.2 Pub/Sub Model

The Providentia infrastructure already divides the testbed into road sections with separate DFUs for each section, and the ROS backend which performs global fusion outputs digital twin for each road stretch in different topics (channels), e.g., *s40s50* for the section between two measurement points *s40* and *s50*. Thus, the ROS pattern of the publisher-subscriber communication among nodes can be mirrored and implemented in this project to achieve scalable data distribution technique. The algorithm then proceeds as follows:

1. A frame message is received from the ROS backend

2. The message handler processes the message by transforming it into the output format and computing additional required meta-data. Then it is published immediately to its appropriate channel. The topic name can be the same as the one for the corresponding road section.

3. The pub/sub broker works independently. Its main duty is to send messages queued in a channel to every recipient subscribed to that channel.

4. Meanwhile, when a location message is received from a client, the backend processes it by identifying which road sections are relevant for the vehicle, and subscribes the client to those channels. Similarly, if the client has subscriptions to channels that are no longer relevant, then the client is unsubscribed from it.

The identification of relevant road sections and non-relevant ones is done through the same location proximity procedure as in the previous section, namely radius search using the `Haversine` formula.

Key point here, is that this pub/sub model is completely transparent to the clients (see Figure 3.9). The pub/sub broker works asynchronously and independently from the incoming message handlers, thus ensuring scalable concurrent processing. Moreover, in such a case, there is no costly looping through the object list of a frame message and all the clients, instead, the entire message is processed and relayed directly to the set of relevant vehicles.

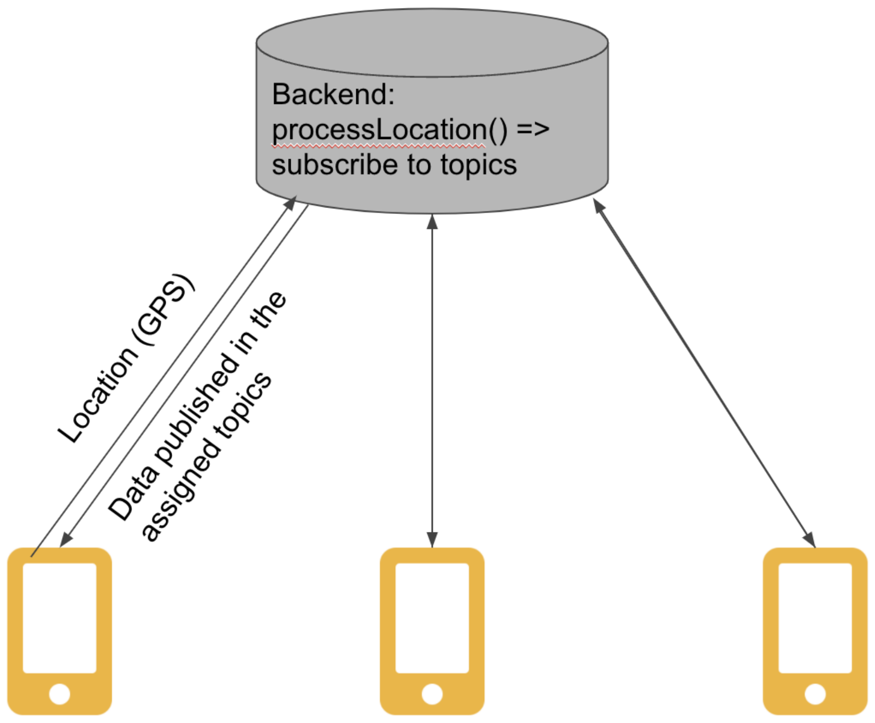


Figure 3.9: Pub/sub model depicting the client connection part. Clients are completely unaware of the underlying model, as the communication setup has not changed.

Furthermore, this data distribution technique is horizontally scalable, as channels can be easily divided among several high-load physical servers. A load balancing can then be applied to distribute incoming client connections based on positional relevance.

Backpressure

Since the pub/sub broker forms queues for each topic, and there are many clients that are subscribed to the same topic, one caveat that needs to be discussed additionally is the `backpressure` [Phe19]. In the software world "`backpressure`" is an analogy borrowed from fluid dynamics, like in automotive exhaust and house plumbing:
Resistance or force opposing the desired flow of fluid through pipes (definition taken from Wikipedia).
In the context of software, the definition could be tweaked to refer to the flow of data within software:
Resistance or force opposing the desired flow of data through software.
The purpose of software is to take input data and turn it into some desired output data. That output data might be JSON from an API, it might be HTML for a webpage, or the pixels displayed on a monitor.
`Backpressure` is when the progress of turning that input to output is resisted in some way. In most cases that resistance is computational speed — trouble computing the output as fast as the input comes in. In the case of this work, the backpressure issue arises when there are slow receivers in the list of subscribers. Nowadays, when simple smartphones are equipped with powerful CPUs and have plenty of RAM, this backpressure is mostly due to the network slowdowns rather than sluggish processing of messages. As the broker quickly sends out a message to all the clients it has to handle slow receivers in some way, so as not to make other fast clients wait for them to receive next message.

Definitely this issue should be tackled from both sides: the backend and a mobile application or frontend. There are several strategies to handle backpressure from the server side:

- **Control** the producer (slow down/speed up is decided by consumer)
- **Buffer** (accumulate outgoing data spikes temporarily)
- **Drop** (either sample out a percentage of the outgoing data, or drop messages that cannot be processed by a client at the moment)

Obviously, the option of slowing down the relay of messages is not viable. Since the backend receives live data from the ROS infrastructure it is imperative to send the processed digital twin data as fast as possible for the clients to receive the traffic info with the minimal delay.

Buffering messages for each connection is a good option as vehicles that move on highways can experience occasional problems with mobile networks causing network slowdown, thus impairing the mobile application to receive timely updates. In such a case, when the vehicle is back to a good coverage area, the buffer will get quickly drained. The size of the buffer is a hyperparameter and has to be set up in advance. However, even with sufficiently large buffer size, it might get full. Bad strategy would be to increase the size of the buffer. The problem of a slow connection can persist for some long time. Additionally, there is no use of handling out the old messages with stale data. So it is better to skip the clients with the full buffer, i.e. drop messages that cannot be processed by them. Dropping random messages is not a good idea at this point, as the client then will experience stuttering without realizing that the problem is with the slow network.

To summarize, the implemented strategy to handle backpressure involves using buffers for each connection and skipping the clients with full buffers. This way, there is no slowdown of the relay of live messages from the ROS infrastructure because of the slow receivers.

3.5 Solution Architecture

The final technological stack of the backend system consists of the following modules:

1. **Node.js** is used as the application server. Its event-driven architecture is particularly well suited for this project. The programming language is JavaScript.
2. `rosnodejs` is the client library for Node.js through which a connection to the ROS system is established (A.7). This library can be easily installed through the `npm` module manager (A.8).
3. **WebSocket** protocol is used for real-time low latency connections between the backend and the clients. Security is maintained by using WebSocket Secure (wss), or SSH tunneling with port forwarding for testing purposes.
4. **Publisher-subscriber** model functions as the core application logic for efficient and scalable data distribution. It runs asynchronously and independently of the incoming message handlers.

Figure 3.10 depicts the final overall solution architecture of the project.

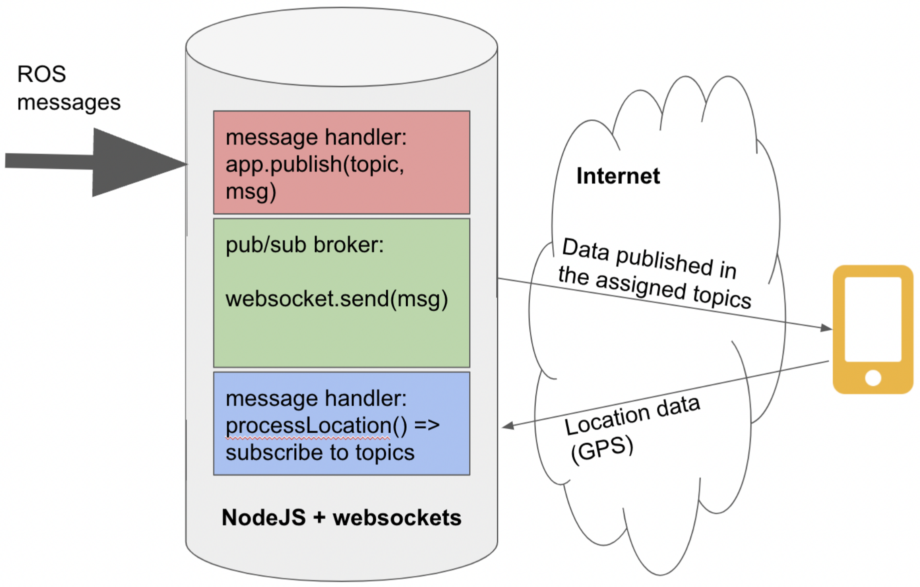

ROS messages
message handler: app.publish(topic, msg)
pub/sub broker:
websocket.send(msg)
message handler: processLocation() => subscribe to topics
NodeJS + websockets
Internet
Data published in the assigned topics
Location data (GPS)

Figure 3.10: Overall solution architecture. The backend is directly connected to the ROS utilizing `rosnodejs` client library. Each client establishes a persistent full-duplex websocket connection with the backend.

3.6 Features

The modularized architectural setup summarized in the previous section gives enough flexibility for the backend system to act as a framework. On top of it additional accident prevention mechanisms can be implemented adding value to the whole system for the end users, i.e. drivers. In particular, for this project a set of warnings/recommendations was implemented to compliment the digital twin information.

Essentially, there are two types of warnings: frame-level and object-level (personalized warnings for the vehicle). The former means that this kind of information can be put together into the frame messages and are to be seen by all the vehicles who receive it. The latter means that this kind of information should be sent personally to a vehicle.

3.6.1 Frame-level warnings

Frame-level warnings include the following (fields are defined in Table 3.2):

1. Weather condition. Field `weather_condition` may contain values: rainy, foggy, snowy, cloudy, and sunny. This information is taken from the ROS backend. For example, when the value is 'rainy', then the frontend can display an aquaplaning warning encouraging a driver to slowdown. Or if it is 'foggy', then display a warning to slowdown and turn on fog lights.
2. Speed recommendation. Field `speed_rec` contains the average speed of the vehicles driving in the same direction of a road section, in km/h.
3. Warning about ghost drivers. Ghost drivers are detected on the backend, when a ROS message is being parsed. The warning text is put into the field `warnings`. Note: `warnings` is a concatenation of different warning descriptions separated by a semicolon. Additionally, each warning type has a unique id (refer to the code base on TUM GitLab https://gitlab.lrz.de/providentiaplusplus/providentia-smartphone-app/-/tree/main/backend). This might be helpful for the frontend to distinguish among different texts.
4. Stationary vehicles. This information can be extracted from the existing field `objects[i].scenarios.standing`. If 'True', then the frontend can highlight this object on the map.
5. Pedestrian on the highway. This information can be extracted from the existing field `objects[i].category`. If 'pedestrian', then the frontend can display an alert and highlight that object on the map.

3.6.2 Object-level warnings

Since Section 3.3.1 (Data Exchange) does not contain any specification for this separate warning messages, it is defined here. Table 3.8 contains the specification.

Name	Type	Description
msg_type	String	Message type. Value = 'warning'
timestamp	String	The interval between the date value and 00:00:00 UTC on 1 January 1970
warnings	String	A string that contains different object-level warnings separated by a semicolon

Table 3.8: Specification of the object-level warning message

The personal warning messages are to be seen only by the vehicles that they are sent to. These include:

1. Overspeeding warning. Although this can be easily implemented by the mobile application, the backend also determines this case, and sends an appropriate warning in the 'warnings' field.
2. Notify vehicles on the right lane if there are vehicles on the entry ramp. A notification message is sent in case if the vehicle is moving along the right lane of the highway, and there is a detected object on the entry ramp (lane 5).
3. Notify vehicles when they enter/exit the Autonomous Driving Test Stretch. An entrance notification is sent to the device when it approaches the testbed. Similarly, an exit notification is sent when the vehicle is about to exit the testbed.
4. Lane recommendation. For this notification, first the lane of the client is detected, then lane recommendation is based on a simple count of objects for each lane. The lane with the minimal number of vehicles is recommended for the driver.

To incorporate this functionality, the architecture of the backend was extended with the additional module for object-level warnings. The module is bound to the external event triggered by the receipt of location information from a client. The message handler first processes the information and then invokes `sendWarnings` module in an asynchronous mode. This ensures the concurrent and independent computation of personalized warnings using the last acknowledged position of the driver. Figure 3.11 depicts the augmented module stack.

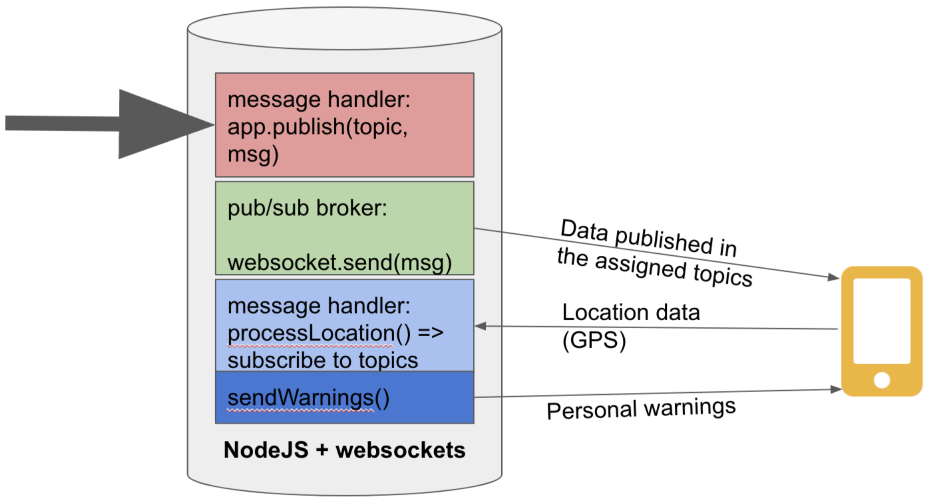

message handler: app.publish(topic, msg)
pub/sub broker:
websocket.send(msg)
message handler: processLocation() => subscribe to topics
sendWarnings()
NodeJS + websockets
Data published in the assigned topics
Location data (GPS)
Personal warnings

Figure 3.11: `sendWarnings` module. Its asynchronous execution is triggered by the receipt of location information from a client.

3.7 Implementation Details

3.7.1 Playing ROS Bags and Direct Deployment as a ROS Node

The final deployment of the backend assumes direct connection with the ROS infrastructure. Using `rosnodejs` library, the server starts its own ROS node under the `/nodejs_server` name, and subscribes to the ROS topics corresponding to road sections.

However, for testing purposes during the active development phase it is more convenient to use prerecorded ROS messages from the live system in the form of a ROS bag file (A.22).

The python script is used to extract each message from the bag file into the corresponding JSON file, which is then used as an input to the backend system.

To summarize, there are two types of input in the development environment: ROS message from the live ROS infrastructure and extracted JSON files.

3.7.2 Configuration file

The backend system needs a configuration file to start the application. At the minimum, it contains configurable parameters such as websocket port, ssl toggle with paths to key and certificate files, backpressure buffer size, and a list of road sections. The format of the file is JSON. Table 3.9 lists the specification of the config.json.

Name	Type	Description
port	Number	Port number to be used for WebSocket connections
ssl	Boolean	Flag that defines the use of the WebSocket Secure protocol
key_file_name	String	Path to the key file. Has to be defined in the case of ssl=true
cert_file_name	String	Path to the certificate. Has to be defined in the case of ssl=true
maxPayloadLength	Number	Maximum length of received message. If a client tries to send a message larger than this, the connection is immediately closed.
maxBackpressure	Number	Maximum length of allowed backpressure per socket when publishing or sending messages. Slow receivers with too high backpressure will be skipped until they catch up or timeout.
radius	Number	Radius (in m) used in defining location proximity
road_sections	[Section]	A list of road sections of type Section

Table 3.9: Specification of the config file. The 'Type' column specifies the actual data type after parsing.

Each road section in turn contains its own properties described in Table 3.10.

Name	Type	Description
stretch	String	Name of the topic to be used in the pub/sub broker
latitude	Number	The latitude in degrees with positive values extending north of the equator and negative values extending south of the equator
longitude	Number	The longitude in degrees with positive values extending east of the meridian and negative values extending west of the meridian
ros_topic	String	Name of the ROS topic
path	String	Path to the folder containing parsed rosbag messages

Table 3.10: Specification of the `Section` type. The 'Type' column specifies the actual data type after parsing.

Important to note, since the format of the configuration file is JSON, any extension, i.e. addition of new parameters is always backwards compatible. However, omission of any parameter specified in the tables leads to error outputs.

3.7.3 WebSocket Module

There are two ways to go about WebSocket implementation:

1. write own native websocket server using the built-in `http` module,
2. or utilize a library with robust implementation of the websocket server.

The option of writing own server is applicable in a case where a specific custom functionality is required. Otherwise, it is the same as re-inventing the wheel. Since, there are no specific implementation requirements, it suffices to go with the second option and choose among the many WebSocket libraries available for Node.js.

μWebSockets.js was chosen due to its three main advantages:

1. It is a standard's compliant websocket library with a perfect Autobahn|Testsuite score since 2016 (A.26), i.e. there is no need for a client to implement any particular subprotocol or stick to the provided client library of the module. Any client with built-in standard support for `WebSocket` protocol is able to connect to the web server.
2. It provides built-in support for pub/sub and TLS 1.3 (A.18).
3. The thoroughly optimized implementation is header-only C++17, cross-platform and compiles down to a tiny binary of a handful kilobytes on any platform. Being written in native code directly targeting the Linux kernel makes it way faster than any JavaScript implementation. In fact, the μWebSockets.js WebSocket server is battle tested as one of the most popular implementations, reaching many millions of end-users daily. It's currently juggling billions of USD in many popular Bitcoin exchanges, every day with outstanding real-world performance.

3.7.4 Coordinates Transformation

The digital twin coming from the ROS backend contains a list of objects. The position of an object is coded as a two-dimensional vector in the road coordinate system. In order to forward this information further to the clients, it has been decided to do the inline transformation of the position into the Earth geographical coordinates, latitude and longitude. Figure 3.12 depicts the global and road reference frames.

The outer leg of the traffic sign gantry bridge (top left corner in the figure) is the global coordinate system (the testbed origin). The inner leg of the traffic sign gantry bridge (bottom right corner in the figure) is the road coordinate system.

The algorithm to convert local position to GPS coordinates is the following:

1. Translate the global origin GPS to the UTM coordinates (A.23) using the `PROJ` software (A.24). Zone 32, northern hemisphere.
2. Apply translation and rotation. $\Delta x = -7.67$ and $\Delta y = -25.89$. The rotation of the coordinate system is -196.5 degrees. This will yield the origin of the road reference frame in UTM coordinates.
3. Add the offset given by an object position to the origin.

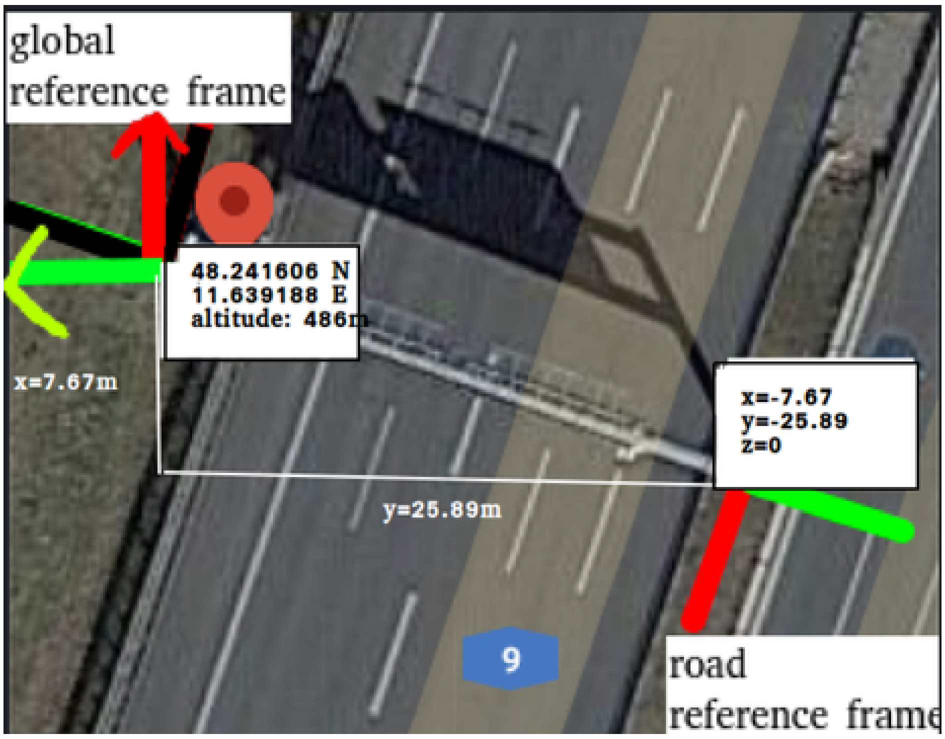


Figure 3.12: Global and road reference frames. x-axis is red, y-axis is green.

4. Project back UTM coordinates to the GPS latitude and longitude using `PROJ`.

Performing position transformation at the backend saves time and battery life of the devices.

Chapter 4

Evaluation Analysis

This chapter describes evaluation analysis of the implemented backend server application.

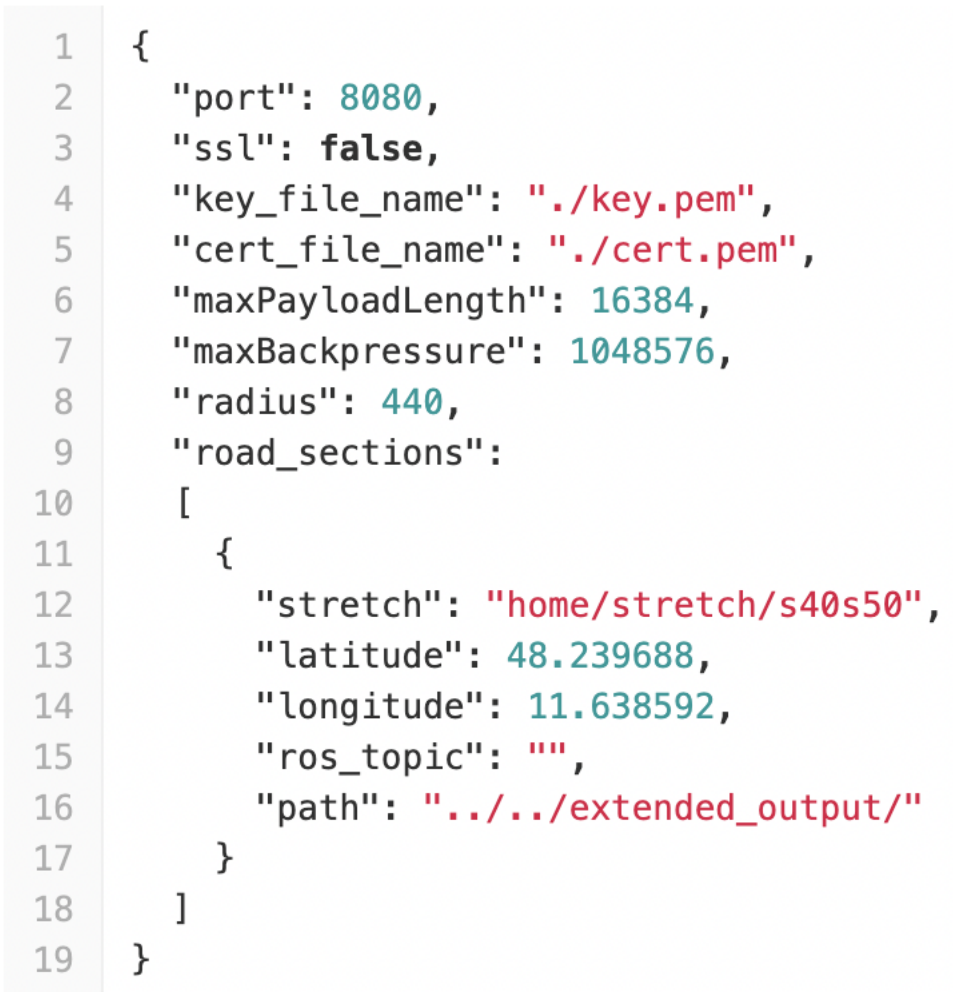
4.1 Experiments

In order to do the evaluation, a set of experiments has been carried out according to the following three testing scenarios:

1. Testing of the backend system on the local development machine using parsed messages from a provided ROS bag (A.22). The already parsed messages are saved as JSON files into a predefined folder. The server application performs synchronous read of the files, orders by time preserving the chronological order, and invokes a ROS message handler by issuing an emit event. The server emulates the ROS publishing rate which is approximately 25 Hz through the use of synchronous timeouts which are set to 40 ms. A dummy client connects through the `localhost` interface and sends its location every 100 `ms` which is set somewhere in the middle of the *s40s50* road section.
2. Testing of the backend system on the local development machine by playing the ROS bag file (A.22 on the local ROS installation. This process emulates the real-case scenario of receiving ROS messages through the `rosnodejs` (A.7) library. The ROS message handler is invoked directly when a message is received. The dummy client connects through the `localhost` interface and sends its location every 100 `ms` which is set somewhere in the middle of the *s40s50* road section.
3. Live testing of the backend system hosted on the premises of the Providentia server infrastructure. The framework is directly connected to the ROS backend and receives the live traffic data. The client is the mobile application implemented in the scope of Mohammad Naanaa's bachelor thesis. The secure connection is established through the use of SSH tunneling with port forwarding enabled (see Section 3.3.3). That is, a direct SSH connection is established using a predefined user and its private/public key pair. Then WebSocket connection is tunneled through it. In such case, no other option is provided, so as not to expose the ROS server infrastructure.

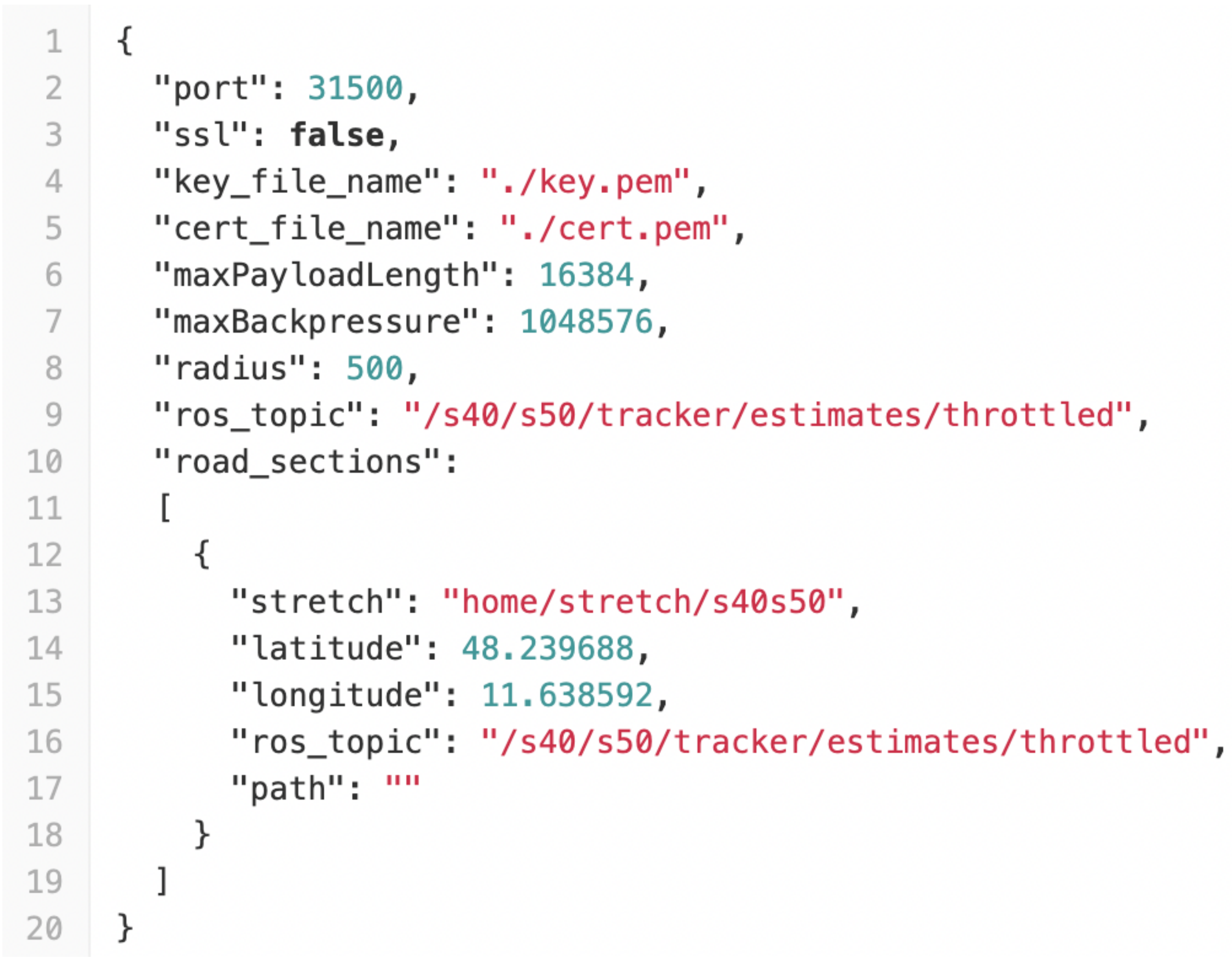
In all the testing cases, the server software was packaged into the same Docker container (A.25) with the base image `noetic-ros-core-focal`. This image comes with pre-installed latest version of ROS called Noetic on top of the Ubuntu 20.04 (Focal) operating system.

Figure 4.1 lists the configuration file (3.7.2) used for the first testing scenario. Whereas, Figure 4.2 lists the configuration file used for the second and the third cases.



```
{
  "port": 8080,
  "ssl": false,
  "key_file_name": "./key.pem",
  "cert_file_name": "./cert.pem",
  "maxPayloadLength": 16384,
  "maxBackpressure": 1048576,
  "radius": 440,
  "road_sections":
  [
    {
      "stretch": "home/stretch/s40s50",
      "latitude": 48.239688,
      "longitude": 11.638592,
      "ros_topic": "",
      "path": "../../extended_output/"
    }
  ]
}
```

Figure 4.1: Configuration file for the case of parsed messages from a ROS bag file. Here in the road section object, the path to the folder containing parsed messages is specified.



```
{
  "port": 31500,
  "ssl": false,
  "key_file_name": "./key.pem",
  "cert_file_name": "./cert.pem",
  "maxPayloadLength": 16384,
  "maxBackpressure": 1048576,
  "radius": 500,
  "ros_topic": "/s40/s50/tracker/estimates/throttled",
  "road_sections":
  [
    {
      "stretch": "home/stretch/s40s50",
      "latitude": 48.239688,
      "longitude": 11.638592,
      "ros_topic": "/s40/s50/tracker/estimates/throttled",
      "path": ""
    }
  ]
}
```

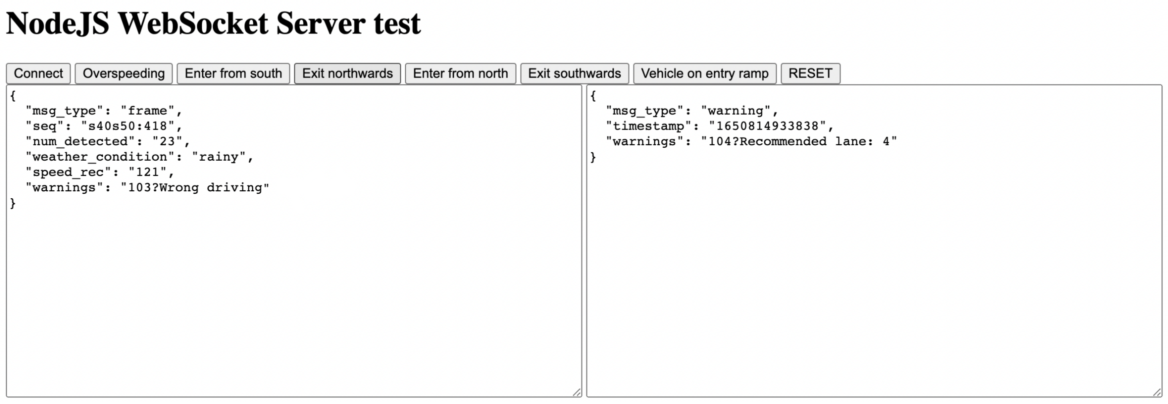
Figure 4.2: Configuration file used for the second and third testing scenarios. The `ros_topic` field is specified in the road section object, as messages are received through the ROS topic subscription interface.

4.1.1 Results

For the first two scenarios that involved testing on the local machine a simple command-line client was implemented. It establishes a websocket connection, sends its position on the testbed, and upon receiving any message from the backend, it outputs it to the console. Testing scenarios involved a concurrent connection of up to 3 dummy clients.

To test the pub/sub model working correctly in the case of just one stretch *s40s50*, it was decided to mock additional *s50s60*, *s60s70*, *s70s80* stretches of the same length to the north of the current testbed by making a translated copy of the *s40s50* digital twin. A client would then send a series of GPS locations progressing northwards along the mocked extended testbed. Setting the search radius to 440 meters, ensured that at any point a client would be subscribed to at most two adjacent road sections.

Additionally, in order to test a set of warnings/recommendations described in Section 3.6 that serve as a proof of concept of an accident prevention mechanism implemented on top of the framework, a web browser client (webpage) was developed using the `html` and JavaScript. Figure 4.3 shows the webpage with a sample output of both frame and warning messages in separate panes.



```
{
  "msg_type": "frame",
  "seq": "s40s50:418",
  "num_detected": "23",
  "weather_condition": "rainy",
  "speed_rec": "121",
  "warnings": "103?Wrong driving"
}
```

```
{
  "msg_type": "warning",
  "timestamp": "1650814933838",
  "warnings": "104?Recommended lane: 4"
}
```

Figure 4.3: The web browser client developed for testing the frame-level and object-level warnings. After connecting to the server, the left pane shows some basic information extracted from a received frame message. The right pane shows the entire warning message upon its receipt. Top-level buttons emulate different vehicle positions which are associated with the scenarios described in Section 3.6.

For the live testing of the backend system, the emulated mobile application prototype developed with the full support of the data exchange protocol described in Section 3.3.1 was used directly on the testbed. The mobile app sent its real GPS position and received the digital twin of the *s40s50* road section. It was able to show live animated traffic data overlaid over Google Maps satellite view of the testbed (Figure 4.4).

The qualitative results of the experiments demonstrate the fully functional backend prototype that fulfills the goals set out at the start of the project. The backend system is able to receive digital twin information. At the same time, the backend is handling real-time concurrent connections from clients and receives location information from it. The implemented pub/sub model properly distributes digital twin information to the connected clients based on the positional proximity.

4.1.2 Performance

Essentially, the main aim of the project is the accident prevention aspect. In the context of the goals specified, it basically boils down to the minimum delay in the processing of

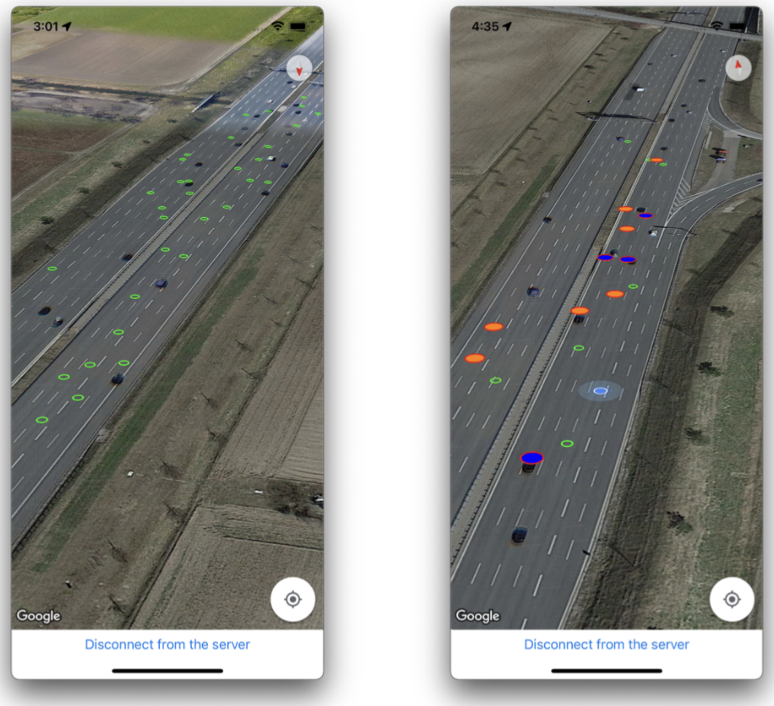


Figure 4.4: UI screenshots (iPhone 13) displaying two different approaches for visualizing detected vehicles on a map. The map on the left draws all vehicles with a single marker type indicating only vehicle's position. The map on the right generates scenario-based markers for each vehicle based on the risk scenarios provided by the ROS backend. [Naa22]

digital twin information. A client should receive traffic data as quickly possible, as well as be notified about potential hazards in advance, thus making the implemented backend system time-critical.

Backend Time Delay

In order to quantitatively measure the performance of the system during the experiments, processing latency was used as the main characteristic.

Local machine testing

In the local testing scenarios it was not logical to time latency incurred by connections, since the communication is done over the `localhost` interface, and it does not generalize to real-world scenarios. So, in such case the latency of the ROS message handling was measured against a ROS bag containing 1480 messages. The average measured latency was 2 `ms`, the minimum and maximum values were 0 and 8 `ms` respectively. Zero milliseconds implies that the message was actually processed in less than one millisecond. Figure 4.5 demonstrates the histogram with a normal curve over it.

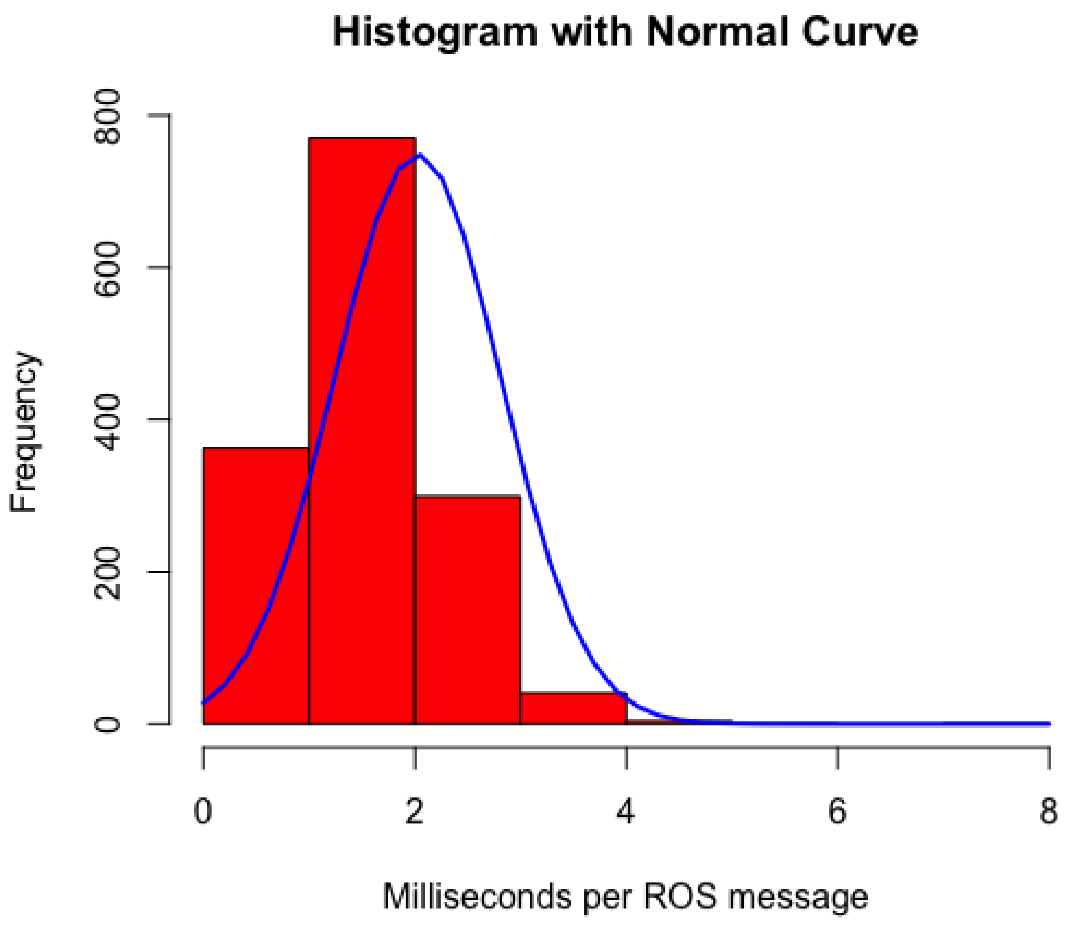


Figure 4.5: Histogram of ROS messages handling latencies. Plot was produced in RStudio.

The average latency of 2 milliseconds is a good result performance-wise. Since, ROS messages are published approximately every 40 `ms`, the information from a message is processed and relayed further in a relatively instant mode. Besides that, it is worth mentioning that the message handling step also includes extraction of frame-level warnings.

Similarly, the timing of the independent asynchronous module `sendWarnings()` that is responsible for object-level warnings/recommendations was produced. This module is triggered every time a client sends its location information. Figure 4.6 displays a histogram of 645 latencies recorded in approximately one minute. The average latency is 0.31 `ms`, the minimum and maximum values are 0 and 3 `ms` respectively.

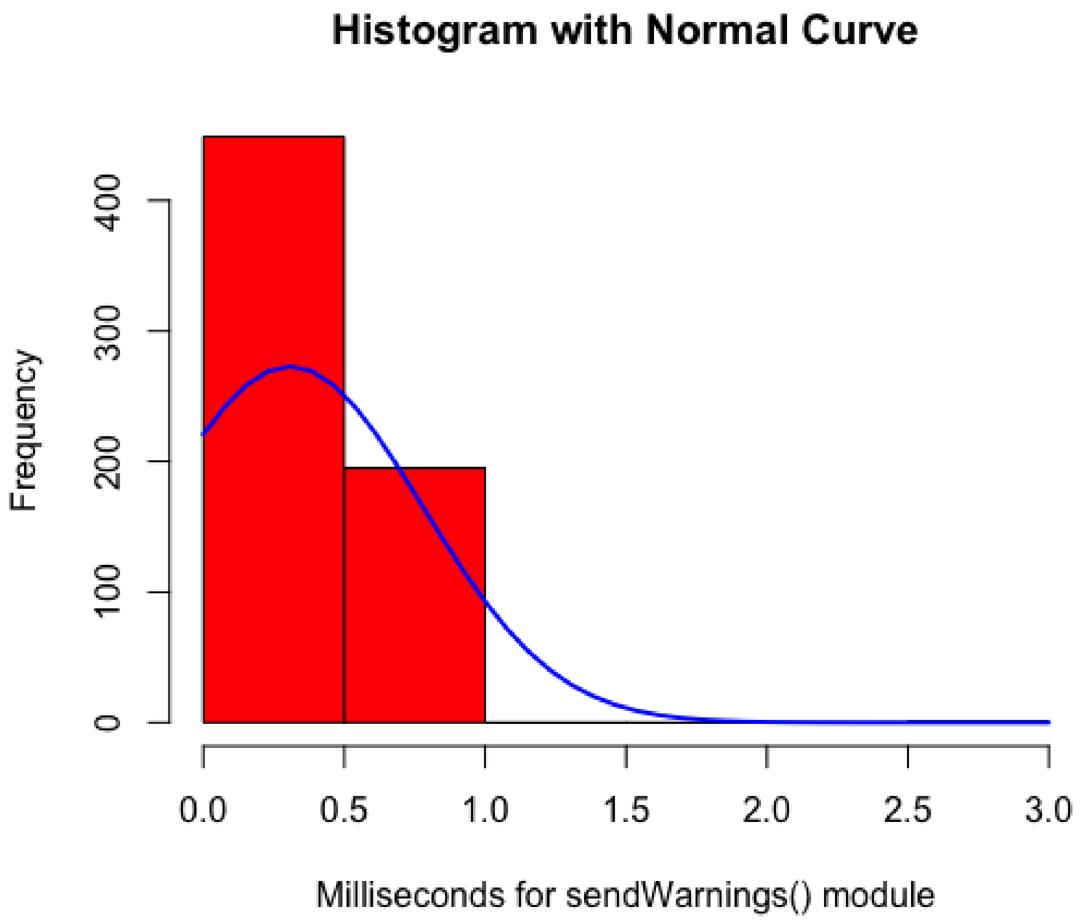


Figure 4.6: Histogram of `sendWarnings()` latencies. Plot was produced in RStudio.

Mobile application

On the contrary, in the live testing mode with the mobile app involved, the latency incurred along the entire path from the ROS backend to the mobile device display was measured (Figure 4.7). ROS messages provide timestamp information, which corresponds to the time a message was processed at the data fusion unit (refer to 3.2). This timestamp is then relayed by the Node.js backend system inside the frame message (Table 3.2).

For the analysis data was collected from 1095 real vehicles from the *s40s50* road stretch. The average measured latency was 665 `ms`, the minimum and maximum values were 128 `ms` and 1326 `ms` respectively. Figure 4.8 displays a histogram of the latencies.

Table 4.1 demonstrates the time delay translation into meters traveled for different speeds

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

Figure 4.7: A scheme describing the complete path of a single message generated by the data fusion unit. After generation, a `timestamp` is attached to the message and it travels to the Node.js backend, which in turn after processing it, relays further to the mobile app. After the decoding process, a `timeToDisplay` is computed as the difference between `Date.now` and the original `timestamp`.

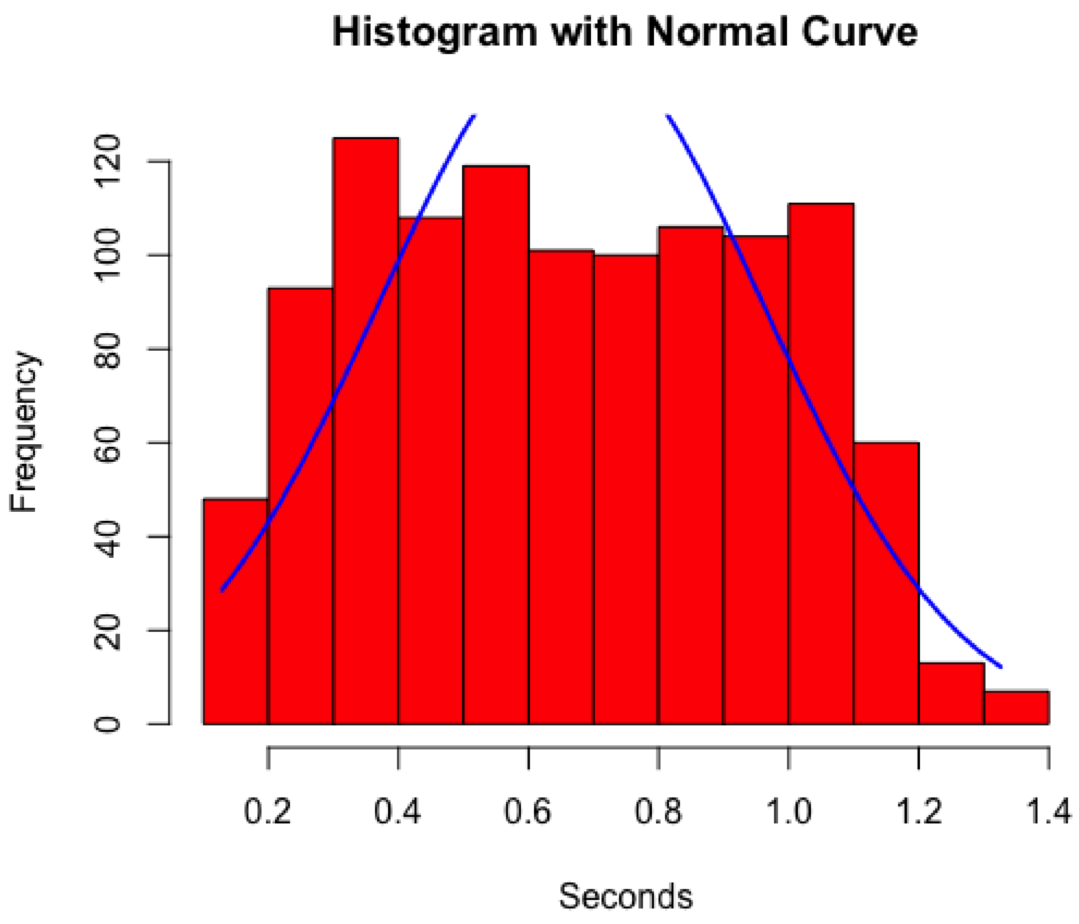


Figure 4.8: Histogram of 1095 latencies recorded on the mobile app from the generation timestamp of the traffic data at DFU.

for the average latency, as well as minimum and maximum latencies.

Speed	s_{min}	s_{avg}	s_{max}
10 km/h	0.36 m	1.85 m	3.68 m
30 km/h	1.07 m	5.54 m	11.05 m
50 km/h	1.78 m	9.24 m	18.42 m
80 km/h	2.84 m	14.78 m	29.47 m
100 km/h	3.55 m	18.47 m	36.83 m
130 km/h	4.62 m	24.01 m	47.88 m
200 km/h	7.11 m	36.94 m	73.67 m

Table 4.1: A table demonstrating distances traveled for different vehicle speeds based on the minimum, average, and maximum time delay of a single frame message. 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 minimum, average, and maximum delays.

In addition to the theoretical values of meters traveled for the delay, a field test was performed to obtain an empirical result [Naa22]. During the test, a static reference object - a massive cell tower - was selected 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 approximately one truck length was observed, which translates accordingly into 16 meters traveled. With a measured truck velocity of approximately 80 km/h, this result matches the average estimate in the theoretical values in Table 4.1 for the average latency. Figure 4.9 displays the performed test.

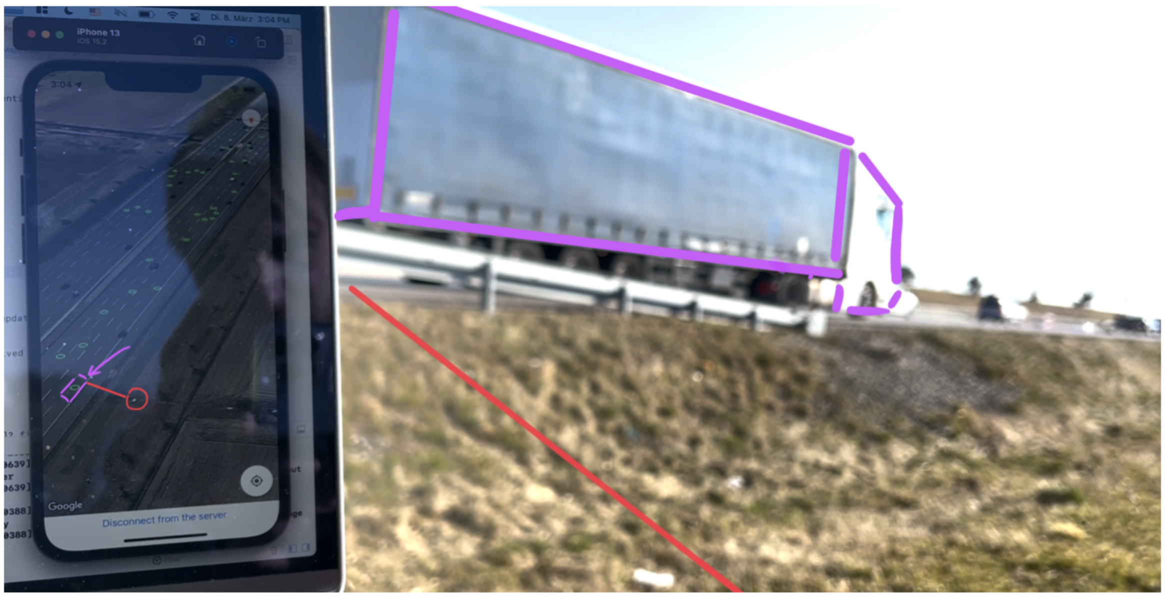


Figure 4.9: Empirical experiment. On the left, a live output from the mobile app simulator with the road section *s40s50* is visible. A reference object (a massive cell tower) is highlighted on the map with a red circle and a perpendicular line to the road is also drawn in red. A truck is highlighted with a violet rectangle around its detected center drawn as a green hollow circle. On the right, the actual road with the truck (in violet) and the perpendicular line (in red) is seen. A delay of approximately one truck length with respect to the red line can be observed. [Naa22]

System Throughput Capability

Extensive load testing has not been conducted due to the lack of server hardware resources. However, the system throughput capability can be estimated theoretically. With the provided ROS bag, where each frame message contains around 23 to 25 detected objects, it has been estimated through the use of Linux command line tools that each parsed message contains approximately 5 kilobytes of data. The size of the separate warning messages and location data from the clients is negligible. Since the rate of incoming ROS messages is 25 Hz, then the outbound bandwidth per client is 1 megabit per second. This estimate accurately corresponds to the actual network bandwidth consumption on a small rented cloud host. One connected client consumed ≈1 `Mbps`, whereas 3 concurrent clients consumed in total 2.7 – 3 `Mbps`. The actual numbers are slightly lower, since the μWebSockets.js library has been optimized for secure TLS 1.3 (A.18) connections and utilizes compression.

The processing burden is negligible, for the current prototype is mostly data-heavy. For each connected client it has to send 40 messages per second. The results of message handling performance measurements (Figure 4.5) further enforce this statement.

The library responsible for managing WebSocket connections and the pub/sub algorithm, μWebSockets.js, has already been practically benchmarked in comparison with the other popular WebSocket library - Socket.IO - on a limited Raspberry Pi 4 hardware. The study showed that μWebSockets.js provided the best possible throughput and was able to sustain 100k secure TLS 1.3 connections, with 50k outgoing messages per second [Hul20].

4.2 Conclusion

In this work, two main accomplishments are achieved. These include:

1. Creation of a backend framework which effectively relays digital twin from the ROS infrastructure to the connected clients in real-time mode. A client runs a mobile application to connect to the server using secure WebSocket protocol. The mobile application advertises its location, and receives the relevant traffic data.

2. Creation of a module with a set of warnings/recommendations on top of the framework. The module processes both traffic data and a client location in order to detect risk scenarios and issue associated warnings (see Section 3.6).

The choices made during the implementation phase proved to be correct. The event-driven architecture of Node.js gave enough flexibility to create a time-critical prototype of the backend framework. Whereas, the WebSocket protocol is used for low-latency real-time communication with the clients.

With appropriate hardware resources available, it is however imperative to conduct further thorough testing and benchmarking of the backend system. Nevertheless, the demonstrated processing latency measurements in Figures 4.5 and 4.6 give a solid good result and an optimistic starting point for further developments and extensions. Important to note, the latencies measured with the mobile application directly on the testbed serve actually as an upper bound due to the following factors:

1. The connection is tunneled through the SSH with port forwarding. This gives a network slowdown factor of at least two, for it is essentially tunneling TCP over TCP.

2. For a more accurate timing, the clocks have to have constant synchronization.

3. The latency also includes some time involved in the processing of a message on the ROS backend.

4. The mobile app prototype was run in a simulator on a computer connected to the Internet through wireless Internet tethering by a cell phone. This gives considerable slowdown.

5. The backend system was not hosted on a high-availability server.

Chapter 5

Summary

Given the Providentia infrastructure on the A9 highway near Garching, the aim of this work was to create a physical gateway on one end connected to the Providentia system, on the other end providing virtual access through the Internet. This gateway should allow users (drivers, pedestrians) to connect and receive relevant digital twin information from the Providentia backend. The relevance of the traffic data is determined by the geographical proximity of a client. With the access to the extended coverage provided by the Providentia system a driver now has better spatial understanding of its surrounding scene and is able to plan its maneuvers more safely and proactively. Thus, the result of this work is to approach and tackle the limited perception coverage of a vehicle in a more cost-effective practical way: rather than working only with autonomous vehicles, the gateway provides access to the virtual surrounding scene through the conventional data networks of the cellular carriers.

The efforts made in this work spanned six months of research and development and can be summarized in the following brief outline, which basically corresponds to the structure of the thesis:

- Firstly, relevant works which include research papers, projects, and review articles are scrutinized to deepen the understanding of the project, its motivation and goals, and to obtain a rough conceptual view of the technical approach.
- The Providentia software stack and infrastructure is studied to obtain good knowledge of the underlying technology and to figure out the appropriate way to connect to it. The Providentia backend is run using ROS (A.1) and the digital twin information is provided in the form of ROS messages under a specified topic at 25 Hz rate. ROS developer community provides client libraries for different platforms.
- Having basic understanding and knowledge of building server-client web applications, a high-level schema of the architecture is proposed (Figure 3.1). The design of the backend system should address three main aspects:

  1. Connection to the ROS backend
  2. Network protocol for client connections
  3. Distribution of relevant data

- The data exchange specification (Section 3.3.1) between the backend and a client is defined which serves as a foundation for the following software development work.
- With imposed requirements on the application server (event-driven, real-time), Node.js is chosen as a technological framework. Its event-driven architecture is especially suitable for creating high-load real-time web servers. Importantly, there is a client library for Node.js to connect to the ROS.

- WebSocket among other options is chosen as a network protocol for client connections. It provides low-latency persistent full-duplex connections between two entities, as the backend and a client push data independently from each other.
- Initially the brute-force looping is used as a data distribution algorithm, but later on, the more elegant pub/sub model is introduced. The division of the road into stretches defined by measurement points [Krä+19] is mirrored in the backend. The backend then automatically manages client subscriptions based on the client GPS position.
- A set of frame-level and object-level warnings/recommendations (Section 3.6) is implemented on top of the backend framework.
- Evaluation of the backend system is performed using three testing scenarios:
    1. Local testing using parsed messages from a ROS bag.
    2. Local testing using local ROS installation and playing a ROS bag.
    3. Live testing on the testbed using an iOS mobile application [Naa22].

  The qualitative and quantitative results demonstrate the fully functional backend system that fulfills the goals set out at the start of project. Although, the more extensive testing is required in the live testing scenario in order to determine the lower bound of the latencies.

The implemented backend framework in Node.js provides real-time performance with the negligible processing latency (not including the connection latencies). It also provides flexibility for future performance enhancements and value-adding modules such as various accident prevention mechanisms on top of the framework.

Chapter 6

Outlook

This last chapter provides summarizing insight on the current utility of the implemented backend framework, as well as next possible improvements to reduce the processing latency and make the system more flexible towards further enhancements.

6.1 Deployment

Given the functional ROS infrastructure and the equipped testbed, the implemented backend system is ready to be deployed along the ROS server infrastructure. It is able to receive digital twin information, process it, augment with a set of warnings/recommendations (Section 3.6, and relay further to the clients. The implemented data distribution algorithm through the pub/sub model is robust and scalable. The backend framework is client-agnostic, meaning that any client implementing the data exchange specification and utilizing WebSocket protocol can connect and receive the relevant traffic data and warnings. Thus, any updates and enhancements to the Providentia project can be easily verified down the pipeline directly on the testbed when the digital twin shows up on the mobile application's map.

Furthermore, as already mentioned in the previous chapter, the modularized event-driven architecture of the backend framework (due to the use of Node.js) provides great flexibility for possible future enhancements - various accident prevention mechanisms - on top of the framework. For example, an asynchronous module for object-level warnings (Section 3.6) was implemented as a proof of concept. This module is currently bound to the external event: receipt of GPS position from a client.

The pub/sub model aligns well with the ROS infrastructure by mimicking the data distribution of the ROS system. The implemented pub/sub is scalable, as the algorithm does not perform global brute-force looping through all the detected objects, but rather assigns each client to the relevant road sections. One caveat though: the search of relevant road sections is done by looping through all available stretches and calculating the distance as a measure of relevance. However, since there are very few (currently one) road sections, and its quantity does not change often, and does not significantly increase, the simple looping is preferable to the optimized algorithms (for example, binary search). In the case of more road sections ($n > 100$), the search procedure can be easily optimized using interval trees.

To summarize, since the ROS backend divides road into sections and provides digital twin output by subscribing to corresponding topics, and the implemented backend uses the same pub/sub model, the entire system is therefore horizontally scalable. With the future expansion of the ROS infrastructure, the road sections can be clustered geographically, and clusters can be assigned to separate high-load servers. The load-balancer then reroutes the client connections to the appropriate servers. Moreover, each server can be safely replicated to handle increased load of concurrent connections, i.e. two or more servers serving the same cluster of road sections. This is possible because there is no centralized data and computation critical for the entire system operation.

6.2 Future Improvements

Obviously, further improvements are possible and are in fact mandatory in case of rapid and massive deployment of the Providentia system. The improvements can be categorized into two groups: architectural modifications and refined data model.

6.2.1 Improved Architecture

In order to decrease processing latency the first obvious step is to move ROS message handling logic into a C++ code, thankfully `roscpp` (A.3) is the main client library for ROS. Because Node.js runs on the V8 JavaScript execution engine written completely in C++, there exist native bindings through a C-based API between code written in Node.js and a C++ code loaded by it ([Wik]). The execution in such case of the ROS message handling will significantly speed up, as C++ is directly compiled into optimal machine code.

During the development of the module for object-level warnings, it became apparent that this module and any other possible modules on top of the framework depend on the ROS data. Since, personal warnings and ROS message handling are bound to different events, then the ROS message handler was augmented to store some data, required for personal warnings every time a ROS message is received. It is a sub-optimal solution for the following two reasons:

1. ROS message frequency is 25 Hz, and every time the handler is invoked it runs the same augmented code to compute and store data for personal warnings. However, the frequency of client messages containing GPS position is different. The exact rate is unknown as the mobile device decides the frequency of updates based on several factors, such as remaining battery charge, GPS satellite signal strength, magnitude of distance change, and the like. It is better to compute and store required data once, when it is actually needed.

2. Any further enhancement and development of accident prevention mechanisms on top of the framework will require in-depth analysis of the data required from the digital twin. And the ROS message handler will then need to be augmented accordingly. This is ineffective and prone to bugs and code breaks.

Thus, instead of figuring out what part of data might be needed for any further enhancements and augmenting ROS message handler correspondingly, and also in order to reduce CPU waste, it is advisable to store the entire digital twin data in the fast memory. Every time the ROS message is received, the data is updated. Apparently by storing all the data, a greater flexibility is achieved by decoupling and optimization of processing and relay of information. Two immediate advantages follow:

1. Any further extension on top of the framework can utilize the available ROS data without a need to edit other core modules, e.g., message handlers. Besides this, it solves the problem of the CPU waste, the additional computations by separate data processing modules are only performed at exactly the moment they are needed.

2. Another big and the most important advantage is the ability now to decouple frequencies of inbound and outbound frame messages. In the Chapter 4 *Evaluation Analysis*, it was estimated that the outbound traffic per client is 1 Megabit per second. Each client receives roughly 40 messages in one second. This is a huge load, which will lead to problems with the increasing number of clients. Though horizontal scaling is possible, it is very inefficient to send too much data. Moreover, data processing modules can define its own outbound rates without explicit binding to any external event, such as receiving location messages from clients. Obviously, traffic data is not the stock exchange data, and microsecond updates and reactions are not applicable to it. Thus, decreasing the outbound rate is a valid solution with far-reaching benefits for the entire system.

Additionally, system monitoring and management tools should be developed to complement the backend framework. This will give better system administration using one application console to manage a cluster of servers as a unified resilient system.
The improved schematic architectural stack is depicted in Figure 6.1.

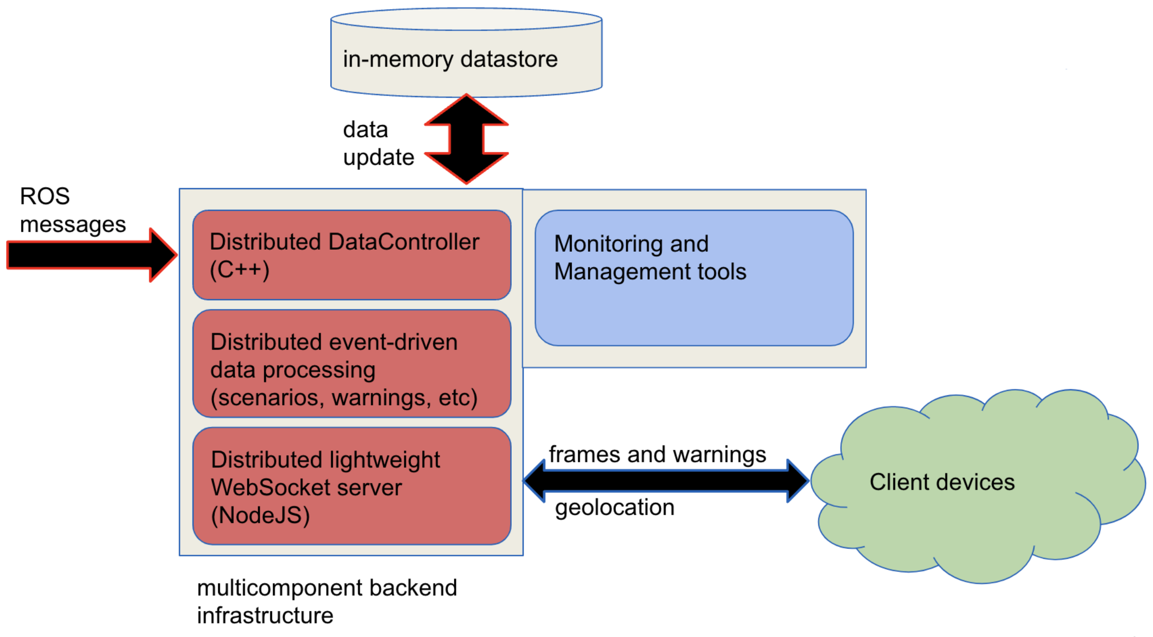

in-memory datastore
data update
ROS messages
Distributed DataController (C++)
Distributed event-driven data processing (scenarios, warnings, etc)
Distributed lightweight WebSocket server (NodeJS)
Monitoring and Management tools
frames and warnings
geolocation
Client devices
multicomponent backend infrastructure

Figure 6.1: Modified architecture of the backend framework. The key novelty is the storing of the digital twin information, better encapsulated modularization, and development of monitoring tools.

6.2.2 Refined Data Model

To further optimize and reduce the outbound bandwidth per client, the granularity of data distribution algorithm can be further refined. Currently, the way it operates, the pub/sub model obviates three apparent inefficiencies:

1. The test stretch *s40s50* is 440 meters, and sending all the traffic data in one frame message is an overhead. If a client is at the end of the stretch, it also receives entire data from the next long stretch, as well as the data that is behind it. It is planned, that the road sections in Providentia will be of roughly the same length.

2. The digital twin for the stretch contains traffic in both directions. A client travelling in one direction on a highway does not actually need the data of the opposite direction.

3. The `radius` property (Table 3.9) is defined globally for all the vehicles. It does not take each vehicle properties such as velocity and location update frequency into consideration. That is, the data received by two vehicles, one of which is slow, and the other one is fast, is the same. Setting the `radius` property to big enough value (in the current implementation it is set to 440 m) still ensures the properly functioning backend system for all clients, however, at the cost of too big outbound bandwidth.

In order to overcome these problems, following improvements can be applied:

1. A road section can be further divided into several short stretches of roughly the same length. For example, stretch *s40s50* can be divided into four stretches of around 110 meters long each. The data distribution algorithm then operates on the short stretches. The predefined remapping of the existing road sections in the ROS system to the short stretches of the Node.js backend system can be configured and executed at the start of the server application. Further, when a ROS message is received, the remapping is applied to dynamically assign detected objects to the corresponding stretches.
2. Additionally, the shorter stretches can be defined for both directions. That is *s40s50* is divided into actually 8 stretches, 4 for each direction. Since the digital twin includes lane id of each detected object, it is very trivial. This will cut the outbound traffic per client in half.
3. A new module inside the backend can be developed to dynamically determine the length of the immediate upcoming traffic distance converted into meters based on the speed and location update rate of a client. This dynamic individual approach ensures the optimal bandwidth consumption. Thus, given the length of the shorter stretches, the system determines the number of those stretches that a client should receive data from. For example, a speeding vehicle will be subscribed to more stretches ahead, than a slow vehicle at the same position.

The first two steps involve creating a directed adjacency graph where each edge is a stretch with a particular moving direction. Figure 6.2 depicts the directed graph for the *s40s50* testbed. Important to note, that this graph is defined once for the entire road, and is loaded and utilized by the backend system with appropriate remapping of the road sections defined by the ROS infrastructure into more fine-grained structure with short sub-stretches in each direction.

Since, the adjacency graph contains the complete reachability information, it also solves the problem of finding the right sub-stretches for a client. Initial search upon the connection can be performed through the optimal interval trees. Whereas, the following searches can be done in a much smarter way by utilizing the last known position and direction of a vehicle. The system knowing the speed and the elapsed time from the previous update, and considering the spatial information in the adjacency graph can very efficiently, in constant time, determine the next relevant sub-stretches.

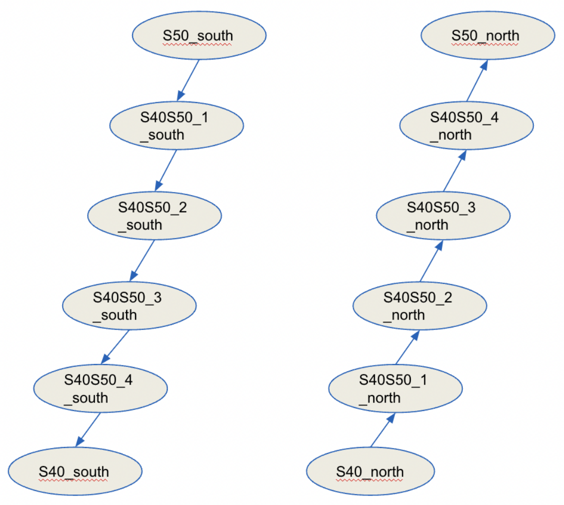


Figure 6.2: The directed graph of the refined *s40s50* road section. Each sub-stretch is 110 meters long. The graph is partitioned into two sub-graphs, as there is no actual possibility to switch directions.

Appendix A

Appendix

A.1 ROS

Robot Operating System (ROS or ros) is an open-source robotics middleware suite. Although ROS is not an operating system (OS) but a set of software frameworks for robot software development, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post, and multiplex sensor data, control, state, planning, actuator, and other messages. See http://wiki.ros.org/ROS/Introduction.

A.2 Boost

Boost provides free peer-reviewed portable C++ source libraries. See https://www.boost.org/.

A.3 roscpp

roscpp is a C++ implementation of ROS. It provides a client library that enables C++ programmers to quickly interface with ROS Topics, Services, and Parameters. roscpp is the most widely used ROS client library and is designed to be the high-performance library for ROS. See http://wiki.ros.org/roscpp

A.4 rosjava

Rosjava provides both a client library for ros communications in java as well as growing list of core tools (e.g. tf, geometry) and drivers (e.g. hokuyo). See http://wiki.ros.org/rosjava

A.5 rospy

rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters. The design of rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS. It is also ideal for non-critical-path code, such as configuration and initialization code. Many of the ROS tools are written in rospy to take advantage of the type introspection capabilities. Many of the ROS tools, such as rostopic and rosservice, are built on top of rospy. See http://wiki.ros.org/rospy

A.6 Asyncio

Asyncio module was added in Python 3.4 and it provides infrastructure for writing single-threaded concurrent code using co-routines. See https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_eventdriven_programming.htm

A.7 rosnodejs

rosnodejs is a pure Node.js implementation of ROS. It provides a client library that enables Node.js programmers to quickly interface with ROS Topics, Services, and Parameters. See http://wiki.ros.org/rosnodejs

A.8 npm

npm (originally short for Node Package Manager) is a package manager for the JavaScript programming language maintained by npm, Inc. See https://www.npmjs.com/

A.9 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. See https://www.json.org/json-en.html

A.10 BackendOutput.msg

Data definition of the frame message with the list of detected objects received from the ROS backend. See https://gitlab.lrz.de/providentiaplusplus/toolchain/-/blob/master/package/system_messages/msg/BackendOutput.msg

A.11 DetectedObject.msg

Data definition of each object in the list of detected objects of the BackendOutput.msg. See https://gitlab.lrz.de/providentiaplusplus/toolchain/-/blob/master/package/system_messages/msg/DetectedObject.msg

A.12 OSI model

The Open Systems Interconnection model (OSI model) is a conceptual model that describes the universal standard of communication functions of a telecommunication system or computing system, without any regard to the system's underlying internal technology and specific protocol suites. Therefore, the objective is the inter-operability of all diverse communication systems containing standard communication protocols, through the encapsulation and de-encapsulation of data, for all networked communication.
The model partitions the flow of data in a communication system into seven abstraction layers, to describe networked communication from the physical implementation of transmitting bits across a communications medium to the highest-level representation of data of a distributed application. Each intermediate layer serves a class of functionality to the layer above it and is served by the layer below it. Classes of functionality are realized in all software development through all and any standardized communication protocols. See https://en.wikipedia.org/wiki/OSI_model

A.13 REST

Representational state transfer (REST) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasises the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems. See https://en.wikipedia.org/wiki/Representational_state_transfer

A.14 Comet

Comet is a web application model in which a long-held HTTPS request allows a web server to push data to a browser, without the browser explicitly requesting it. Comet is an umbrella term, encompassing multiple techniques for achieving this interaction. All these methods rely on features included by default in browsers, such as JavaScript, rather than on non-default plugins. The Comet approach differs from the original model of the web, in which a browser requests a complete web page at a time. See https://en.wikipedia.org/wiki/Comet_(programming)

A.15 Push technology

Push technology or server push, is a style of Internet-based communication where the request for a given transaction is initiated by the publisher or central server. It is contrasted with pull/get, where the request for the transmission of information is initiated by the receiver or client. See https://en.wikipedia.org/wiki/Push_technology

A.16 EventSource API

The EventSource interface is web content's interface to server-sent events. An EventSource instance opens a persistent connection to an HTTP server, which sends events in text/event-stream format. The connection remains open until closed by calling EventSource.close(). See https://developer.mozilla.org/en-US/docs/Web/API/EventSource

A.17 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or, formerly, Secure Sockets Layer (SSL). The protocol is therefore also referred to as HTTP over TLS, or HTTP over SSL. See https://en.wikipedia.org/wiki/HTTPS

A.18 SSL/TLS

Transport Layer Security (TLS), the successor of the now-deprecated Secure Sockets Layer (SSL), is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP, but its use in securing HTTPS remains the most publicly visible.
The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity through the use of certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.
See https://en.wikipedia.org/wiki/Transport_Layer_Security

A.19 Man-in-the-middle attack

In cryptography and computer security, a man-in-the-middle, monster-in-the-middle, machine-in-the-middle, monkey-in-the-middle, meddler-in-the-middle (MITM) or person-in-the-middle (PITM) attack is a cyberattack where the attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other, as the attacker has inserted themselves between the two parties.
See https://en.wikipedia.org/wiki/Man-in-the-middle_attack

A.20 SSH port forwarding

SSH port forwarding is a mechanism in SSH for tunneling application ports from the client machine to the server machine, or vice versa. It can be used for adding encryption to legacy applications, going through firewalls, and some system administrators and IT professionals use it for opening backdoors into the internal network from their home machines.
See https://www.ssh.com/academy/ssh/tunneling/example

A.21 Publish-subscribe pattern

The Publish/Subscribe pattern, sometimes known as pub/sub, is an architectural design pattern that enables publishers and subscribers to communicate with one another. In this arrangement, the publisher and subscriber rely on a message broker to send messages from the publisher to the subscribers. Messages (events) are sent out by the host (publisher) to a channel, which subscribers can join.

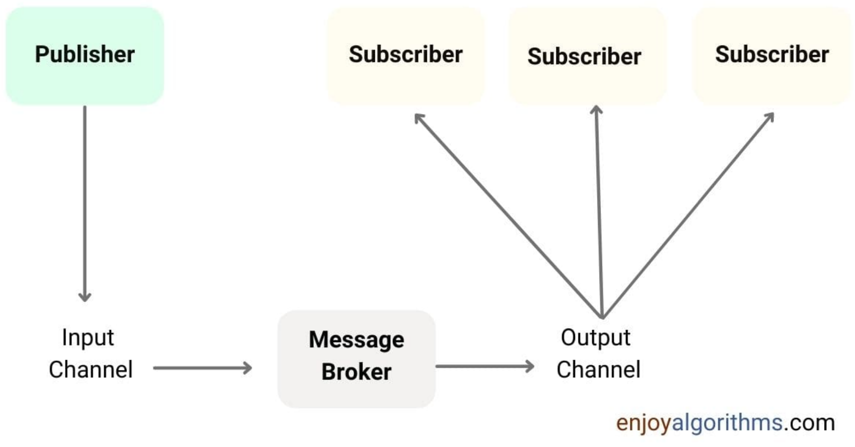


Figure A.1: Publisher-Subscriber (Pub-Sub) Design Pattern

See https://www.enjoyalgorithms.com/blog/publisher-subscriber-pattern

A.22 ROS Bag

A *bag* is a file format in ROS for storing ROS message data. Bags are typically created by a tool like `rosbag`, which subscribe to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.
See http://wiki.ros.org/Bags

A.23 UTM coordinate system

The Universal Transverse Mercator (UTM) is a map projection system for assigning coordinates to locations on the surface of the Earth. Like the traditional method of latitude and longitude, it is a horizontal position representation, which means it ignores altitude and treats the earth as a perfect ellipsoid. However, it differs from global latitude/longitude in that it divides earth into 60 zones and projects each to the plane as a basis for its coordinates.
See https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

A.24 PROJ

PROJ is a generic coordinate transformation software that transforms geospatial coordinates from one coordinate reference system (CRS) to another. This includes cartographic projections as well as geodetic transformations.
See https://proj.org/

A.25 Docker container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
See https://www.docker.com/resources/what-container/

A.26 Autobahn|Testsuite

The Autobahn|Testsuite provides a fully automated test suite to verify client and server implementations of The WebSocket Protocol for specification conformance and implementation robustness.
See https://github.com/crossbario/autobahn-testsuite
Report for μWebSockets.js: https://unetworking.github.io/uWebSockets.js/report.pdf

Bibliography

[Aar22] Aaron Kaefer Walter Zimmer, A. K. “Deep Traffic Scenario Mining, Detection, Classification and Generation on the Autonomous Driving Test Stretch using the CARLA Simulator”. In: *SUMO User Conference 2022* (2022), pp. 1–175.

[Ant18] Antwerp, U. of. *Antwerp Smart Highway*. Accessed: 05.05.2022. 2018. URL: https://www.uantwerpen.be/en/research-groups/idlab/infrastructure/smart-highway.

[Bak17] Baker, J. *Getting Started with Building Realtime API Infrastructure*. Accessed: 06.05.2022. 2017. URL: https://becominghuman.ai/getting-started-with-building-realtime-api-infrastructure-a19601fc794e.

[Ber] Berlin, T. *DIGINET-PS*. Accessed: 05.05.2022. URL: https://diginet-ps.de/en/intelligent-vehicles/.

[BR95] Braess, H. H. and Reichart, G. “PROMETHEUS: VISION DES "INTELLIGENTEN AUTOMOBILS" AUF "INTELLIGENTER STRASSE"? VERSUCH EINER KRITISCHEN WUERDIGUNG - TEILE 1 UND 2”. In: 1995.

[Com18] Commission, E. *GDPR and location data*. Accessed: 18.04.2022. 2018. URL: https://joinup.ec.europa.eu/collection/elise-european-location-interoperability-solutions-e-government/news/gdpr-and-location-data.

[Dia20a] Diaconu, A. *WebSockets and Android apps - client-side considerations*. Accessed: 18.04.2022. 2020. URL: https://ably.com/topic/websockets-android.

[Dia20b] Diaconu, A. *WebSockets and iOS: client-side engineering challenges*. Accessed: 18.04.2022. 2020. URL: https://ably.com/topic/websockets-ios.

[DOT15] DOT, N. *NYC Connected Vehicle Project*. Accessed: 05.05.2022. 2015. URL: https://cvp.nyc.

[ETSa] ETSI. *Intelligent Transport Systems (ITS); Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band*. Accessed: 06.05.2022. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.02.00_20/en_302663v010200a.pdf.

[ETSb] ETSI. *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Analysis of the Collective Perception Service (CPS); Release 2*. Accessed: 06.05.2022. URL: https://www.etsi.org/deliver/etsi_tr/103500_103599/103562/02.01.01_60/tr_103562v020101p.pdf.

[Fle+18] Fleck, T., Daaboul, K., Weber, M., Schörner, P., Wehmer, M., Doll, J., Orf, S., Sußmann, N., Hubschneider, C., Zofka, M. R., Kuhnt, F., Kohlhaas, R., Baumgart, I., Zöllner, R. D., and Zöllner, J. M. “Towards Large Scale Urban Traffic Reference Data: Smart Infrastructure in the Test Area Autonomous Driving Baden-Württemberg”. In: *IAS*. 2018.

[Hul20] Hultman, A. “100k secure WebSockets with Raspberry Pi 4: Practical benchmark of TLS 1.3 WebSockets on limited hardware”. In: (Apr. 2020). URL: https://medium.com/swlh/100k-secure-websockets-with-raspberry-pi-4-1ba5d2127a23.

[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*. 2019. DOI: 10.48550/ARXIV.1906.06789. URL: https://arxiv.org/abs/1906.06789.

[KS21] Kryzhanovska, A. and Sharapova, M. *BUILDING SCALABLE WEB APPLICATION FOR YOUR PROJECT: BEST PRINCIPLES AND PRACTICES*. Accessed: 06.05.2022. 2021. URL: https://gearheart.io/articles/how-build-scalable-web-applications/.

[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.

[MF11] Melnikov, A. and Fette, I. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[Men+17] Menouar, H., Guvenc, I., Akkaya, K., Uluagac, A. S., Kadri, A., and Tuncer, A. “UAV-Enabled Intelligent Transportation Systems for the Smart City: Applications and Challenges”. In: *IEEE Communications Magazine* 55.3 (2017), pp. 22–28. DOI: 10.1109/MCOM.2017.1600238CM.

[Naa22] Naanaa, M. *Bachelor thesis: Accident Prevention Frontend Framework to Support Autonomous Driving*. Mar. 2022.

[Ogu19] Ogundeyi, K. *WebSocket in real time application*. 2019. DOI: 10.4314/njt.v38i4.26. URL: https://www.ajol.info/index.php/njt/article/view/191780.

[Phe19] Phelps, J. *Backpressure explained — the resisted flow of data through software*. Accessed: 20.04.2022. Feb. 2019. URL: https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7.

[QA13] Qureshi, K. and Abdullah, H. “A Survey on Intelligent Transportation Systems”. In: *Middle-East Journal of Scientific Research* 15 (Jan. 2013), pp. 629–642. DOI: 10.5829/idosi.mejsr.2013.15.5.11215.

[Rib21] Ribeiro, E. *How to get updates from a server in real-time?* Accessed: 06.05.2022. Apr. 2021. URL: https://eduardocribeiro.com/blog/real-time-communication/.

[See+19] Seebacher, S., Datler, B., Erhart, J., Greiner, G., Harrer, M., Hrassnig, P., Präsent, A., Schwarzl, C., and Ullrich, M. “Infrastructure data fusion for validation and future enhancements of autonomous vehicles’ perception on Austrian motorways”. In: *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*. 2019, pp. 1–7. DOI: 10.1109/ICCVE45908.2019.8965142.

[Shl92] Shladover, S. “The California PATH Program of IVHS research and its approach to vehicle-highway automation”. In: *Proceedings of the Intelligent Vehicles ‘92 Symposium*. 1992, pp. 347–352. DOI: 10.1109/IVS.1992.252284.

[Tei12] Teixeira, P. *Professional Node.Js: Building Javascript Based Scalable Software*. 1st. GBR: Wrox Press Ltd., 2012. ISBN: 1118185463.

[Ver] Verkehrsverbund, K. *Test Area Autonomous Driving Baden-Württemberg*. Accessed: 05.05.2022. URL: https://taf-bw.de/en/the-test-field/definition-of-objectives.

[Wik] Wikipedia. *Node.js*. URL: https://en.wikipedia.org/wiki/Node.js#Native_bindings.